

Java Language Basics

Contents

1	Creating Variables and Naming Them	5
1.1	Variables	5
2	Creating Primitive Type Variables in Your Programs	6
2.1	Primitive Types	6
3	Initializing a Variable with a Default Value	8
4	Creating Values with Literals	9
5	Integer Literals	10
6	Floating-Point Literals	10
7	Character and String Literals	11
8	Using Underscore Characters in Numeric Literals	12
9	Creating Arrays in Your Programs	13
9.1	Arrays	13
9.2	Declaring a Variable to Refer to an Array	15
9.3	Creating, Initializing, and Accessing an Array	16
9.4	Copying Arrays	19

9.5	Array Manipulations	20
9.6	Wrapping-up Variables and Arrays	22
10	Using the Var Type Identifier	22
10.1	The Var Keyword	22
10.2	Examples with Var	23
10.3	Restrictions on Using Var	23
11	Using Operators in Your Programs	25
11.1	Operators	25
11.2	The Simple Assignment Operator	26
11.3	The Arithmetic Operators	26
11.4	The Unary Operators	28
11.5	The Equality and Relational Operators	30
11.6	The Conditional Operators	31
11.7	The Type Comparison Operator Instanceof	33
11.8	Bitwise and Bit Shift Operators	34
12	Summary of Operators	35
12.1	Simple Assignment Operator	35
12.2	Arithmetic Operators	35
12.3	Unary Operators	35
12.4	Equality and Relational Operators	36
12.5	Conditional Operators	36

12.6 Type Comparison Operator	36
12.7 Bitwise and Bit Shift Operators	36
13 Expressions, Statements and Blocks	37
13.1 Expressions	37
13.2 Floating Point Arithmetic	38
13.3 Statements	39
13.4 Blocks	40
14 Control Flow Statements	40
14.1 The If-Then Statement	40
14.2 The If-Then-Else Statement	41
14.3 The While and Do-while Statements	42
14.4 The For Statement	44
14.5 The Break Statement	47
14.6 The Continue Statement	49
14.7 The Return Statement	51
14.8 The Yield Statement	51
15 Branching with Switch Statements	52
15.1 Using Switch Statements to Control the Flow of Your Program	52
15.2 Choosing Between Switch Statements and If-then-else Statements	55
15.3 Using String as a Type for the Case Labels	56
15.4 Null Selector Variables	58

16 Branching with Switch Expressions	58
16.1 Modifying the Switch Syntax	58
16.2 Producing a Value	60
16.3 Adding a Default Clause	61
16.4 Writing Colon Case in Switch Expressions	62
16.5 Dealing with Null Values	62

1 Creating Variables and Naming Them

1.1 Variables

An object stores its state in fields.



```
1 int cadence = 0;
   int speed = 0;
3 int gear = 1;
```

What are the rules and conventions for naming a field? Besides `int`, what other data types are there? Do fields have to be initialized when they are declared? Are fields assigned a default value if they are not explicitly initialized? We'll explore the answers to such questions in this section, but before we do, there are a few technical distinctions you must first become aware of. In the Java programming language, the terms "field" and "variable" are both used; this is a common source of confusion among new developers, since both often seem to refer to the same thing.

The Java programming language defines the following kinds of variables:

- **Instance Variables (Non-Static Fields)** Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword. Non-static fields are also known as **instance variables** because their values are unique to each instance of a class (to each object, in other words); the `currentSpeed` of one bicycle is independent from the `currentSpeed` of another.
- **Class Variables (Static Fields)** A class variable is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a `static` field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.
- **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in local variables. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword

designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

- **Parameters** You've already seen examples of parameters, both in the **Bicycle** class and in the main method of the "Hello World!" application. Recall that the signature for the main method is **public static void main(String[] args)**. Here, the **args** variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields".

Having said that, the remainder of this tutorial uses the following general guidelines when discussing fields and variables. If we are talking about "fields in general" (excluding local variables and parameters), we may simply say "fields". If the discussion applies to "all of the above", we may simply say "variables". If the context calls for a distinction, we will use specific terms (static field, local variables, etc.) as appropriate. You may also occasionally see the term "member" used as well. A type's fields, methods, and nested types are collectively called its *members*.

2 Creating Primitive Type Variables in Your Programs

You have already learned that objects store their state in fields. However, the Java programming language also uses the term **variable** as well. This section discusses this relationship, plus variable naming rules and conventions, basic data types (primitive types, character strings, and arrays), default values, and literals.

2.1 Primitive Types

The Java programming language is **statically-typed**, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you have already seen:



```
1 int gear = 1;
```

Doing so tells your program that a field named `gear` exists, holds numerical data, and has an initial value of `1`. A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to `int`, the Java programming language supports seven other primitive data types. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

- **byte**: The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of `int` where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
- **short**: The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
- **int**: By default, the `int` data type is a 32-bit signed two's complement integer, which has a minimum value of -2³¹ and a maximum value of 2³¹-1. In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 2³²-1. Use the `Integer` class to use `int` data type as an unsigned integer. See the section The **Number** Classes for more information. Static methods like `compareUnsigned()` have been added to the `Integer` class to support the arithmetic operations for unsigned integers.
- **long**: The `long` data type is a 64-bit two's complement integer. The signed long has a minimum value of -2⁶³ and a maximum value of 2⁶³-1. In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2⁶⁴-1. Use this data type when you need a range of values wider than those provided by `int`. The `Long` class also contains methods like `compareUnsigned()`, `divideUnsigned()` etc to support arithmetic operations for unsigned long.
- **float**: The `float` data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the **Floating-Point Types, Formats, and Values** section of the **Java Language Specification**. As with the recommendations for byte and short, use a `float` (instead of `double`) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use

the `java.math.BigDecimal` class instead. `Numbers` and `Strings` covers `BigDecimal` and other useful classes provided by the Java platform.

- **double**: The `double` data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the `Floating-Point Types, Formats, and Values` section of the `Java Language Specification`. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
- **boolean**: The `boolean` data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- **char**: The `char` data type is a single 16-bit Unicode character. It has a minimum value of `\u0000` (or 0) and a maximum value of `\uffff` (or 65,535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the `java.lang.String` class. Enclosing your character string within double quotes will automatically create a new `String` object; for example:



```
1 String s = "this is a string";
```

`String` objects are immutable, which means that once created, their values cannot be changed. The `String` class is not technically a primitive data type, but considering the special support given to it by the language, you will probably tend to think of it as such. You will learn more about the `String` class in the section `Strings`.

3 Initializing a Variable with a Default Value

It is not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad programming style.

The following table summarizes the default values for the above data types.

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	\u0000
String (or any object)	null
boolean	false

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

4 Creating Values with Literals

You may have noticed that the `new` keyword is not used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A literal is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it is possible to assign a literal to a variable of a primitive type:



```
1 boolean result = true;
   char capitalC = 'C';
3 byte b = 100;
   short s = 10000;
5 int i = 100000;
```

5 Integer Literals

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise it is of type `int`. It is recommended that you use the upper case letter `L` because the lower case letter `l` is hard to distinguish from the digit `1`.

Values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Values of type `long` that exceed the range of `int` can be created from long literals. Integer literals can be expressed by these number systems:

- Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day
- Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
- Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you will ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix `0x` indicates hexadecimal and `0b` indicates binary:



```
1 // The number 26, in decimal
  int decimalValue = 26;

3
  // The number 26, in hexadecimal
5 int hexadecimalValue = 0x1a;

7 // The number 26, in binary
  int binaryValue = 0b11010;
```

6 Floating-Point Literals

A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise its type is `double` and it can optionally end with the letter `D` or `d`.

The floating point types (`float` and `double`) can also be expressed using `E` or `e` (for scientific notation), `F` or `f` (32-bit float literal) and `D` or `d` (64-bit double literal; this is the default and by convention is omitted).



```
double d1 = 123.4;
2
// same value as d1, but in scientific notation
4 double d2 = 1.234e2;
float f1 = 123.4f;
```

7 Character and String Literals

Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as `\u0108` (capital C with circumflex), or `"S\u00ED Se\u00F1or"` (Sí Señor in Spanish). Always use 'single quotes' for char literals and "double quotes" for String literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in `char` or `String` literals.

The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

There is also a special `null` literal that can be used as a value for any reference type. The `null` literal may be assigned to any variable, except variables of primitive types. There is little you can do with a null value beyond testing for its presence. Therefore, `null` is often used in programs as a marker to indicate that some object is unavailable.

Finally, there is also a special kind of literal called a class literal, formed by taking a type name and appending `.class`; for example, `String.class`. This refers to the object that represents the type itself, of type `Class`.

8 Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (`_`) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:



```
1 long creditCardNumber = 1234_5678_9012_3456L;  
   long socialSecurityNumber = 999_99_9999L;  
3 float pi = 3.14_15F;  
   long hexBytes = 0xFF_EC_DE_5E;  
5 long hexWords = 0xCAFE_BABE;  
   long maxLong = 0x7fff_ffff_ffff_ffffL;  
7 byte nybbles = 0b0010_0101;  
   long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an **F** or **L** suffix
- In positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements in numeric literals:



```
// Invalid: cannot put underscores adjacent to a decimal point  
2 float pi1 = 3_.1415F;
```



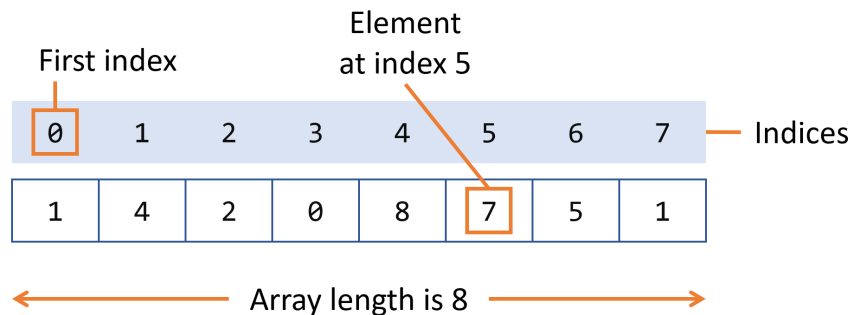
```
// Invalid: cannot put underscores adjacent to a decimal point
4 float pi2 = 3._1415F;
// Invalid: cannot put underscores prior to an L suffix
6 long socialSecurityNumber1 = 999_99_9999_L;

8 // OK (decimal literal)
  int x1 = 5_2;
10 // Invalid: cannot put underscores at the end of a literal
  int x2 = 52_;
12 // OK (decimal literal)
  int x3 = 5_2;
14
  // Invalid: cannot put underscores in the 0x radix prefix
16 int x4 = 0_x52;
  // Invalid: cannot put underscores at the beginning of a number
18 int x5 = 0x_52;
  // OK (hexadecimal literal)
20 int x6 = 0x5_2;
  // Invalid: cannot put underscores at the end of a number
22 int x7 = 0x52_;
```

9 Creating Arrays in Your Programs

9.1 Arrays

An **array** is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You have seen an example of arrays already, in the main method of the "Hello World!" application. This section discusses arrays in greater detail.



An array of 8 elements.

Each item in an array is called an **element**, and each element is accessed by its numerical index. As shown in the preceding illustration, numbering begins with 0. The 6th element, for example, would therefore be accessed at index 5.

The following program, **ArrayDemo**, creates an array of integers, puts some values in the array, and prints each value to standard output.



```
class ArrayDemo {
2  public static void main(String[] args) {
    // declares an array of integers
4  int [] anArray;

6  // allocates memory for 10 integers
    anArray = new int[10];

8

    // initialize first element
10   anArray[0] = 100;
    // initialize second element
12   anArray[1] = 200;
    // and so forth
14   anArray[2] = 300;
    anArray[3] = 400;
16   anArray[4] = 500;
    anArray[5] = 600;
18   anArray[6] = 700;
    anArray[7] = 800;
20   anArray[8] = 900;
    anArray[9] = 1000;
22 }
```



```
24 System.out.println("Element at index 0: " + anArray[0]);
    System.out.println("Element at index 1: " + anArray[1]);
    System.out.println("Element at index 2: " + anArray[2]);
26 System.out.println("Element at index 3: " + anArray[3]);
    System.out.println("Element at index 4: " + anArray[4]);
28 System.out.println("Element at index 5: " + anArray[5]);
    System.out.println("Element at index 6: " + anArray[6]);
30 System.out.println("Element at index 7: " + anArray[7]);
    System.out.println("Element at index 8: " + anArray[8]);
32 System.out.println("Element at index 9: " + anArray[9]);
    }
34 }
```

The output from this program is:

```
Command window
Element at index 0: 100
2 Element at index 1: 200
Element at index 2: 300
4 Element at index 3: 400
Element at index 4: 500
6 Element at index 5: 600
Element at index 6: 700
8 Element at index 7: 800
Element at index 8: 900
10 Element at index 9: 1000
```

In a real-world programming situation, you would probably use one of the supported looping constructs to iterate through each element of the array, rather than write each line individually as in the preceding example. However, the example clearly illustrates the array syntax.

9.2 Declaring a Variable to Refer to an Array

The preceding program declares an array (named `anArray`) with the following line of code:



```
// declares an array of integers  
2 int [] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as `type[]`, where `type` is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions. As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:



```
byte [] anArrayOfBytes;  
2 short [] anArrayOfShorts;  
long [] anArrayOfLongs;  
4 float [] anArrayOfFloats;  
double [] anArrayOfDoubles;  
6 boolean [] anArrayOfBooleans;  
char [] anArrayOfChars;  
8 String [] anArrayOfStrings;
```

You can also place the brackets after the array's name:



```
// this form is discouraged  
2 float anArrayOfFloats [];
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

9.3 Creating, Initializing, and Accessing an Array

One way to create an array is with the `new` operator. The next statement in the `ArrayDemo` program allocates an array with enough memory for 10 integer elements and assigns the

array to the `anArray` variable.



```
// create an array of integers
2 anArray = new int[10];
```

If this statement is missing, then the compiler prints an error like the following, and compilation fails:

Command window

ArrayDemo.java:4: Variable anArray may not have been initialized.

The next few lines assign values to each element of the array:



```
1 anArray[0] = 100; // initialize first element
  anArray[1] = 200; // initialize second element
3 anArray[2] = 300; // and so forth
```

Each array element is accessed by its numerical index:



```
1 System.out.println("Element 1 at index 0: " + anArray[0]);
  System.out.println("Element 2 at index 1: " + anArray[1]);
3 System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:



```
1 int [] anArray = {
    100, 200, 300,
3   400, 500, 600,
    700, 800, 900, 1000
5 };
```

Here the length of the array is determined by the number of values provided between braces and separated by commas.

You can also declare an array of arrays (also known as a multidimensional array) by using two or more sets of brackets, such as `String[][]` names. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following `MultiDimArrayDemo` program:



```
1 class MultiDimArrayDemo {  
    public static void main(String[] args) {  
2        String [][] names = {  
            {"Mr. ", "Mrs. ", "Ms. "},  
3            {"Smith", "Jones"}  
4        };  
5        // Mr. Smith  
        System.out.println(names[0][0] + names[1][0]);  
6        // Ms. Jones  
        System.out.println(names[0][2] + names[1][1]);  
7    }  
8 }
```

The output from this program is:

```
Command window  
Mr. Smith  
2 Ms. Jones
```

Finally, you can use the built-in `length` property to determine the size of any array. The following code prints the array's size to standard output:



```
System.out.println(anArray.length);
```

9.4 Copying Arrays

The `System` class has an `arraycopy()` method that you can use to efficiently copy data from one array into another:



```
1 public static void arraycopy(Object src, int srcPos,
                               Object dest, int destPos, int length)
```

The two `Object` arguments specify the array to copy from and the array to copy to. The three `int` arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, `ArrayCopyDemo`, declares an array of `String` elements. It uses the `System.arraycopy()` method to copy a subsequence of array components into a second array:



```
class ArrayCopyDemo {
2  public static void main(String[] args) {
    String[] copyFrom = {
4      "Affogato", "Americano", "Cappuccino", "Corretto", "Cortado",
      "Doppio", "Espresso", "Frappucino", "Freddo", "Lungo", "Macchiato",
6      "Marocchino", "Ristretto" };

8      String[] copyTo = new String[7];
      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
10     for (String coffee : copyTo) {
        System.out.print(coffee + " ");
12     }
    }
14 }
```

The output from this program is:

Command window

Cappuccino Corretto Cortado Doppio Espresso Frappucino Freddo

9.5 Array Manipulations

Arrays are a powerful and useful concept used in programming. Java SE provides methods to perform some of the most common manipulations related to arrays. For instance, the `ArrayCopyDemo` example uses the `arraycopy()` method of the `System` class instead of manually iterating through the elements of the source array and placing each one into the destination array. This is performed behind the scenes, enabling the developer to use just one line of code to call the method.

For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the `java.util.Arrays` class. For instance, the previous example can be modified to use the `java.util.Arrays` method of the `java.util.Arrays` class, as you can see in the `ArrayCopyOfDemo` example. The difference is that using the `java.util.Arrays` method does not require you to create the destination array before calling the method, because the destination array is returned by the method:



```
1 class ArrayCopyOfDemo {  
    public static void main(String[] args) {  
2        String[] copyFrom = {  
            "Affogato", "Americano", "Cappuccino", "Corretto", "Cortado",  
3            "Doppio", "Espresso", "Frappuccino", "Freddo", "Lungo", "Macchiato",  
4            "Marocchino", "Ristretto" };  
5  
6        String[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);  
7        for (String coffee : copyTo) {  
8            System.out.print(coffee + " ");  
9        }  
10    }  
11 }  
12 }  
13 }
```

As you can see, the output from this program is the same, although it requires fewer lines of code. Note that the second parameter of the `java.util.Arrays` method is the initial index of the range to be copied, inclusively, while the third parameter is the final index of the range to be copied, exclusively. In this example, the range to be copied does not include the array element at index 9 (which contains the string `Lungo`).

Some other useful operations provided by methods in the `java.util.Arrays` class are:

- Searching an array for a specific value to get the index at which it is placed (the

`binarySearch()` method).

- Comparing two arrays to determine if they are equal or not (the `equals()` method).
- Filling an array to place a specific value at each index (the `fill()` method).
- Sorting an array into ascending order. This can be done either sequentially, using the `sort()` method, or concurrently, using the `parallelSort()` method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.
- Creating a stream that uses an array as its source (the `stream()` method). For example, the following statement prints the contents of the `copyTo` array in the same way as in the previous example:



```
1 java.util.Arrays.stream(copyTo).map(coffee -> coffee + " ").forEach(  
    ↪ System.out::print);
```

See Aggregate Operations for more information about streams.

- Converting an array to a string. The `toString()` method converts each element of the array to a string, separates them with commas, then surrounds them with brackets. For example, the following statement converts the `copyTo` array to a string and prints it:



```
System.out.println(java.util.Arrays.toString(copyTo));
```

This statement prints the following:

```
Command window [v]

[Cappuccino, Corretto, Cortado, Doppio, Espresso, Frappuccino, Freddo  
]
```

9.6 Wrapping-up Variables and Arrays

The Java programming language uses both "fields" and "variables" as part of its terminology. Instance variables (non-static fields) are unique to each instance of a class. Class variables (static fields) are fields declared with the static modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated. Local variables store temporary state inside a method. Parameters are variables that provide extra information to a method; both local variables and parameters are always classified as "variables" (not "fields"). When naming your fields or variables, there are rules and conventions that you should (or must) follow.

The eight primitive data types are: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. The `java.lang.String` class represents character strings. The compiler will assign a reasonable default value for fields of the above types; for local variables, a default value is never assigned.

A literal is the source code representation of a fixed value. An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

10 Using the Var Type Identifier

10.1 The Var Keyword

Starting with Java SE 10, you can use the `var` type identifier to declare a local variable. In doing so, you let the compiler decide what is the real type of the variable you create. Once created, this type cannot be changed.

Consider the following example.



```
String message = "Hello world!";  
2 Path path = Path.of("debug.log");  
InputStream stream = Files.newInputStream(path);
```

In that case, having to declare the explicit types of the three variables `message`, `path` and `stream` is redundant.

With the `var` type identifier the previous code can be written as follow:



```
1 var message = "Hello world!";  
   var path = Path.of("debug.log");  
3 var stream = Files.newInputStream(path);
```

10.2 Examples with Var

The following example shows you how you can use the `var` type identifier to make your code simpler to read. Here the `strings` variable is given the type `List<String>`, and the `element` variable the type `String`.



```
1 var list = List.of("one", "two", "three", "four");  
   for (var element: list) {  
3     System.out.println(element);  
   }
```

On this example, the `path` variable is of type `Path`, and the `stream` variable is of type `InputStream`.



```
   var path = Path.of("debug.log");  
2 try (var stream = Files.newInputStream(path)) {  
    // process the file  
4 }
```

Note that on the two previous examples, you have used `var` to declare a variable in a `for` statement and in a `try-with-resources` statement. These two statements are covered later in this tutorial.

10.3 Restrictions on Using Var

There are restrictions on the use of the `var` type identifier.

1. You can only use it for **local variables** declared in methods, constructors and initializer blocks.
2. **var** cannot be used for fields, not for method or constructor parameters.
3. The compiler must be able to choose a type when the variable is declared. Since **null** has no type, the variable must have an initializer.

Following these restrictions, the following class does not compile, because using **var** as a type identifier is not possible for a field or a method parameter.



```
public class User {  
2   private var name = "Sue";  
  
4   public void setName(var name) {  
        this.name = name;  
6   }  
}
```

The same goes for the following code.

In that case, the compiler cannot guess the real type of **message** because it lacks an initializer.



```
1 public String greetings(int message) {  
    var greetings;  
3   if (message == 0) {  
        greetings = "morning";  
5   } else {  
        greetings = "afternoon";  
7   }  
    return "Good " + greetings;  
9 }
```


11 Using Operators in Your Programs

11.1 Operators

Now that you have learned how to declare and initialize variables, you probably want to know how to *do something* with them. Learning the operators of the Java programming language is a good place to start. Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

In general-purpose programming, certain operators tend to appear more frequently than

others; for example, the assignment operator `=` is far more common than the unsigned right shift operator `>>>`. With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

11.2 The Simple Assignment Operator

One of the most common operators that you'll encounter is the simple assignment operator `=`. You saw this operator in the `Bicycle` class; it assigns the value on its right to the operand on its left:



```
1 int cadence = 0;
   int speed = 0;
3 int gear = 1;
```

This operator can also be used on objects to assign object references, as discussed in the section [Creating Objects](#).

11.3 The Arithmetic Operators

The Java programming language provides operators that perform addition, subtraction, multiplication, and division. There is a good chance you will recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is `%`, which divides one operand by another and returns the remainder as its result.

Operator	Description
<code>+</code>	Additive operator (also used for String concatenation)
<code>-</code>	Subtraction operator
<code>*</code>	Multiplication operator
<code>/</code>	Division operator
<code>%</code>	Remainder operator

The following program, `ArithmeticDemo`, tests the arithmetic operators.



```
1 class ArithmeticDemo {
    public static void main (String[] args) {
2         int result = 1 + 2;
          // result is now 3
3         System.out.println("1 + 2 = " + result);
          int original_result = result;
4
5         result = result - 1;
          // result is now 2
6         System.out.println(original_result + " - 1 = " + result);
          original_result = result;
7
8         result = result * 2;
          // result is now 4
9         System.out.println(original_result + " * 2 = " + result);
          original_result = result;
10
11        result = result / 2;
          // result is now 2
12        System.out.println(original_result + " / 2 = " + result);
          original_result = result;
13
14        result = result + 8;
          // result is now 10
15        System.out.println(original_result + " + 8 = " + result);
          original_result = result;
16
17        result = result % 7;
          // result is now 3
18        System.out.println(original_result + " % 7 = " + result);
19    }
20 }
```

This program prints the following:

Command window

```
1 + 2 = 3
2 3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
```

```
6 10 % 7 = 3
```

You can also combine the arithmetic operators with the simple assignment operator to create compound assignments. For example, `x += 1;` and `x = x + 1;` both increment the value of `x` by 1.

The `+` operator can also be used for concatenating (joining) two strings together, as shown in the following `ConcatDemo` program:



```
class ConcatDemo {
2  public static void main(String[] args){
    String firstString = "This is";
4    String secondString = " a concatenated string.";
    String thirdString = firstString+secondString;
6    System.out.println(thirdString);
    }
8 }
```

By the end of this program, the variable `thirdString` contains `This is a concatenated string.`, which gets printed to standard output.

11.4 The Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

The following program, `UnaryDemo`, tests the unary operators:



```
class UnaryDemo {
2   public static void main(String[] args) {
        int result = +1;
4       // result is now 1
        System.out.println(result);

6       result--;
8       // result is now 0
        System.out.println(result);

10      result++;
12     // result is now 1
        System.out.println(result);

14     result = -result;
16     // result is now -1
        System.out.println(result);

18     boolean success = false;
20     // false
        System.out.println(success);
22     // true
        System.out.println(!success);
24 }
}
```

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `result++`; and `++result`; will both end in result being incremented by one. The only difference is that the prefix version (`++result`) evaluates to the incremented value, whereas the postfix version (`result++`) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

The following program, `PrePostDemo`, illustrates the prefix/postfix unary increment operator:



```
1 class PrePostDemo {  
    public static void main(String [] args){  
3         int i = 3;  
            i++;  
5         // prints 4  
            System.out.println(i);  
7         ++i;  
            // prints 5  
9         System.out.println(i);  
            // prints 6  
11        System.out.println(++i);  
            // prints 6  
13        System.out.println(i++);  
            // prints 7  
15        System.out.println(i);  
    }  
17 }
```

11.5 The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use `==`, not `=`, when testing if two primitive values are equal.

Operator	Description
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to
<code><</code>	less than
<code><=</code>	less than or equal to

The following program, `ComparisonDemo`, tests the comparison operators:



```

1 class ComparisonDemo {
    public static void main(String [] args){
3     int value1 = 1;
        int value2 = 2;
5     if(value1 == value2)
        System.out.println("value1 == value2");
7     if(value1 != value2)
        System.out.println("value1 != value2");
9     if(value1 > value2)
        System.out.println("value1 > value2");
11    if(value1 < value2)
        System.out.println("value1 < value2");
13    if(value1 <= value2)
        System.out.println("value1 <= value2");
15 }
    }

```

Running this program produces the following output:

```

Command window
value1 != value2
2 value1 < value2
value1 <= value2

```

11.6 The Conditional Operators

The `&&` and `||` operators perform Conditional-AND and Conditional-OR operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

Operator	Description
<code>&&</code>	Conditional-AND
<code> </code>	Conditional-OR
<code>? :</code>	Ternary (shorthand for if-then-else statement)

The following program, `ConditionalDemo1`, tests these operators:



```
1 class ConditionalDemo1 {  
    public static void main(String [] args){  
3         int value1 = 1;  
         int value2 = 2;  
5         if ((value1 == 1) && (value2 == 2))  
            System.out.println("value1 is 1 AND value2 is 2");  
7         if ((value1 == 1) || (value2 == 1))  
            System.out.println("value1 is 1 OR value2 is 1");  
9     }  
}
```

Another conditional operator is `?:`, which can be thought of as shorthand for an **if-then-else** statement (discussed in the Control Flow Statements section). This operator is also known as the *ternary* operator because it uses three operands. In the following example, this operator should be read as: "If someCondition is true, assign the value of value1 to result. Otherwise, assign the value of value2 to result."

The following program, `ConditionalDemo2`, tests the `?:` operator:



```
class ConditionalDemo2 {  
2     public static void main(String [] args){  
         int value1 = 1;  
4         int value2 = 2;  
         int result;  
6         boolean someCondition = true;  
         result = someCondition ? value1 : value2;  
8  
         System.out.println(result);  
10    }  
}
```

Because `someCondition` is true, this program prints "1" to the screen. Use the `?:` operator instead of an **if-then-else** statement if it makes your code more readable; for example, when the expressions are compact and without side-effects (such as assignments).

11.7 The Type Comparison Operator Instanceof

The `instanceof` operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

The following program, `InstanceofDemo`, defines a parent class (named `Parent`), a simple interface (named `MyInterface`), and a child class (named `Child`) that inherits from the parent and implements the interface.



```
1 class InstanceofDemo {
    public static void main(String[] args) {
2         Parent obj1 = new Parent();
3         Parent obj2 = new Child();
4
5         System.out.println("obj1 instanceof Parent: "
6             + (obj1 instanceof Parent));
7         System.out.println("obj1 instanceof Child: "
8             + (obj1 instanceof Child));
9         System.out.println("obj1 instanceof MyInterface: "
10            + (obj1 instanceof MyInterface));
11        System.out.println("obj2 instanceof Parent: "
12            + (obj2 instanceof Parent));
13        System.out.println("obj2 instanceof Child: "
14            + (obj2 instanceof Child));
15        System.out.println("obj2 instanceof MyInterface: "
16            + (obj2 instanceof MyInterface));
17    }
18 }
19
20
21 class Parent {}
22 class Child extends Parent implements MyInterface {}
23 interface MyInterface {}
```

The following program produces the following output:

Command window

```
1 obj1 instanceof Parent: true
  obj1 instanceof Child: false
3 obj1 instanceof MyInterface: false
```

```
obj2 instanceof Parent: true
5 obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

When using the `instanceof` operator, keep in mind that `null` is not an instance of anything.

11.8 Bitwise and Bit Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator `^` inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is `00000000` would change its pattern to `11111111`.

The signed left shift operator `<<` shifts a bit pattern to the left, and the signed right shift operator `>>` shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator `>>>` shifts a zero into the leftmost position, while the leftmost position after `>>` depends on sign extension.

The bitwise `&` operator performs a bitwise AND operation.

The bitwise `^` operator performs a bitwise exclusive OR operation.

The bitwise `|` operator performs a bitwise inclusive OR operation.

The following program, `BitDemo`, uses the bitwise AND operator to print the number "2" to standard output.



```
class BitDemo {
2  public static void main(String[] args) {
    int bitmask = 0x000F;
4    int val = 0x2222;
```



```
// prints "2"
6 System.out.println(val & bitmask);
    }
8 }
```

12 Summary of Operators

12.1 Simple Assignment Operator

Operator	Description
=	Simple assignment operator

12.2 Arithmetic Operators

Operator	Description
+	Additive operator (also used for String concatenation)
−	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

12.3 Unary Operators

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
−	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

12.4 Equality and Relational Operators

Operator	Description
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

12.5 Conditional Operators

Operator	Description
&&	Conditional-AND
	Conditional-OR
? :	Ternary (shorthand for if-then-else statement)

12.6 Type Comparison Operator

Operator	Description
instanceof	Compares an object to a specified type

12.7 Bitwise and Bit Shift Operators

Operator	Description
~	Unary bitwise complement
<<	Signed left shift
>>	Signed right shift
>>>	Unsigned right shift
&	Bitwise AND
^	Bitwise exclusive OR
	Bitwise inclusive OR

13 Expressions, Statements and Blocks

13.1 Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value. You have already seen examples of expressions, illustrated in code below:



```
int cadence = 0;
2 anArray[0] = 100;
  System.out.println("Element 1 at index 0: " + anArray[0]);
4
int result = 1 + 2; // result is now 3
6 if (value1 == value2) {
  System.out.println("value1 == value2");
8 }
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression `cadence = 0` returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `cadence` is an `int`. As you can see from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here is an example of a compound expression:



```
1 * 2 * 3
```

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, no matter in which order you apply the multiplications. However, this is not true of all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:



```
1 x + y / 100    // ambiguous
```

You can specify exactly how an expression will be evaluated using balanced parenthesis: (and). For example, to make the previous expression unambiguous, you could write the following:



```
1 (x + y) / 100  // unambiguous, recommended
```

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:



```
1 x + y / 100    // ambiguous  
  
3 x + (y / 100)  // unambiguous, recommended
```

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and to maintain.

13.2 Floating Point Arithmetic

Floating point arithmetic is a special world in which common operations may behave unexpectedly. Consider the following code.



```
1 double d1 = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;  
  System.out.println("d1 == 1 ? " + (d1 == 1.0));
```

You would probably expect that it prints **true**. Due to the way floating point addition is conducted and rounded, it prints **false**.

13.3 Statements

Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

- Assignment expressions
- Any use of `++` or `--`
- Method invocations
- Object creation expressions
- Such statements are called expression statements. Here are some examples of expression statements.



```
// assignment statement
2 aValue = 8933.234;

4 // increment statement
  aValue++;

6
// method invocation statement
8 System.out.println("Hello World!");

10 // object creation statement
   Bicycle myBike = new Bicycle();
```

In addition to expression statements, there are two other kinds of statements: declaration statements and control flow statements. A declaration statement declares a variable. You have seen many examples of declaration statements already:



```
// declaration statement
2 double aValue = 8933.234;
```

Finally, control flow statements regulate the order in which statements get executed. You will learn about control flow statements in the next section, Control Flow Statements.

13.4 Blocks

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, BlockDemo, illustrates the use of blocks:



```
class BlockDemo {
2  public static void main(String[] args) {
    boolean condition = true;
4    if (condition) { // begin block 1
        System.out.println("Condition is true.");
6    } // end block one
    else { // begin block 2
8        System.out.println("Condition is false.");
    } // end block 2
10 }
}
```

14 Control Flow Statements

14.1 The If-Then Statement

The **if-then** statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to **true**. For example, the **Bicycle** class could allow the brakes to decrease the bicycle's speed only if the bicycle is already in motion. One possible implementation of the **applyBrakes()** method could be as follows:



```
1 void applyBrakes() {
    // the "if" clause: bicycle must be moving
3  if (isMoving){
    // the "then" clause: decrease current speed
5    currentSpeed--;
    }
7 }
```


If this test evaluates to **false** (meaning that the bicycle is not in motion), control jumps to the end of the **if-then** statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:



```
1 void applyBrakes() {  
    // same as above, but without braces  
3  if (isMoving)  
    currentSpeed -- ;  
5 }
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you will just get the wrong results.

14.2 The If-Then-Else Statement

The **if-then-else** statement provides a secondary path of execution when an "if" clause evaluates to **false**. You could use an **if-then-else** statement in the **applyBrakes()** method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.



```
1 void applyBrakes() {  
    if (isMoving) {  
3     currentSpeed -- ;  
    } else {  
5     System.err.println("The bicycle has already stopped!");  
    }  
7 }
```

The following program, **IfElseDemo**, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.



```
1 class IfElseDemo {  
    public static void main(String[] args) {  
3         int testscore = 76;  
        char grade;  
5  
        if (testscore >= 90) {  
7            grade = 'A';  
        } else if (testscore >= 80) {  
9            grade = 'B';  
        } else if (testscore >= 70) {  
11           grade = 'C';  
        } else if (testscore >= 60) {  
13           grade = 'D';  
        } else {  
15           grade = 'F';  
        }  
17        System.out.println("Grade = " + grade);  
    }  
19 }
```

The output from the program is:

```
Command window  
1 Grade = C
```

You may have noticed that the value of `testscore` can satisfy more than one expression in the compound statement: `76 >= 70` and `76 >= 60`. However, once a condition is satisfied, the appropriate statements are executed (`grade = 'C'`;) and the remaining conditions are not evaluated.

14.3 The While and Do-while Statements

The `while` statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:



```
1 while (expression) {  
    statement(s)  
3 }
```

The **while** statement evaluates expression, which must return a **boolean** value. If the expression evaluates to **true**, the **while** statement executes the **statement(s)** in the while block. The **while** statement continues testing the expression and executing its block until the expression evaluates to **false**. Using the **while** statement to print the values from 1 through 10 can be accomplished as in the following **WhileDemo** program:



```
1 class WhileDemo {  
    public static void main(String [] args){  
3         int count = 1;  
        while (count < 11) {  
5             System.out.println("Count is: " + count);  
            count++;  
7         }  
    }  
9 }
```

You can implement an infinite loop using the **while** statement as follows:



```
1 while (true){  
    // your code goes here  
3 }
```

The Java programming language also provides a do-while statement, which can be expressed as follows:



```
1 do {  
    statement(s)  
3 } while (expression);
```

The difference between **do-while** and **while** is that **do-while** evaluates its expression at the

bottom of the loop instead of the top. Therefore, the statements within the **do** block are always executed at least once, as shown in the following **DoWhileDemo** program:



```
1 class DoWhileDemo {  
    public static void main(String [] args){  
3        int count = 1;  
        do {  
5            System.out.println("Count is: " + count);  
            count++;  
7        } while (count < 11);  
    }  
9 }
```

14.4 The For Statement

The **for** statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the **for** statement can be expressed as follows:



```
1 for (initialization; termination; increment) {  
    statement(s)  
3 }
```

When using this version of the for statement, keep in mind that:

- The *initialization* expression initializes the loop; it is executed once, as the loop begins.
- When the *termination* expression evaluates to **false**, the loop terminates.
- The *increment expression* is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.

The following program, **ForDemo**, uses the general form of the **for** statement to print the numbers 1 through 10 to standard output:



```
1 class ForDemo {  
    public static void main(String[] args){  
3        for(int i = 1; i < 11; i++){  
            System.out.println("Count is: " + i);  
5        }  
    }  
7 }
```

The output of this program is:

Command window

```
1 Count is: 1  
Count is: 2  
3 Count is: 3  
Count is: 4  
5 Count is: 5  
Count is: 6  
7 Count is: 7  
Count is: 8  
9 Count is: 9  
Count is: 10
```

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the **for** statement, so it can be used in the termination and increment expressions as well. If the variable that controls a **for** statement is not needed outside of the loop, it is best to declare the variable in the initialization expression. The names **i**, **j**, and **k** are often used to control **for** loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the **for** loop are optional; an infinite loop can be created as follows:



```
// infinite loop  
2 for ( ; ; ) {  
    // your code goes here  
4 }
```

The **for** statement also has another form designed for iteration through Collections and

arrays. This form is sometimes referred to as the *enhanced* for statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:



```
int [] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

The following program, **EnhancedForDemo**, uses the *enhanced* for to loop through the array:



```
1 class EnhancedForDemo {  
    public static void main(String [] args){  
3         int [] numbers =  
            {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
5         for (int item : numbers) {  
            System.out.println("Count is: " + item);  
7         }  
    }  
9 }
```

In this example, the variable **item** holds the current value from the numbers array. The output from this program is the same as before:

Command window

```
1 Count is: 1  
Count is: 2  
3 Count is: 3  
Count is: 4  
5 Count is: 5  
Count is: 6  
7 Count is: 7  
Count is: 8  
9 Count is: 9  
Count is: 10
```

We recommend using this form of the **for** statement instead of the general form whenever possible.

14.5 The Break Statement

The **break** statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the **switch** statement. You can also use an unlabeled **break** to terminate a **for**, **while**, or **do-while** loop, as shown in the following **BreakDemo** program:



```
class BreakDemo {
2  public static void main(String[] args) {
    int[] arrayOfInts =
4      {32, 87, 3, 589,
        12, 1076, 2000,
6        8, 622, 127};
    int searchfor = 12;

8
    int i;
10   boolean foundIt = false;

12   for (i = 0; i < arrayOfInts.length; i++) {
        if (arrayOfInts[i] == searchfor) {
14             foundIt = true;
                break;
16         }
    }

18
    if (foundIt) {
20         System.out.println("Found " + searchfor + " at index " + i);
    } else {
22         System.out.println(searchfor + " not in the array");
    }

24 }
}
```

This program searches for the number 12 in an array. The **break** statement, terminates the **for** loop when that value is found. Control flow then transfers to the statement after the **for** loop. This program's output is:

Command window

```
1 Found 12 at index 4
```

An unlabeled **break** statement terminates the innermost **switch**, **for**, **while**, or **do-while** state-

ment, but a labeled `break` terminates an outer statement. The following program, `Break-WithLabelDemo`, is similar to the previous program, but uses nested `for` loops to search for a value in a two-dimensional array. When the value is found, a labeled `break` terminates the outer `for` loop (labeled "search"):



```
1 class BreakWithLabelDemo {
    public static void main(String[] args) {
2         int [][] arrayOfInts = {
3             {32, 87, 3, 589},
4             {12, 1076, 2000, 8},
5             {622, 127, 77, 955}
6         };
7         int searchfor = 12;
8
9         int i;
10        int j = 0;
11        boolean foundIt = false;
12
13        search:
14        for (i = 0; i < arrayOfInts.length; i++) {
15            for (j = 0; j < arrayOfInts[i].length;
16                j++) {
17                if (arrayOfInts[i][j] == searchfor) {
18                    foundIt = true;
19                    break search;
20                }
21            }
22        }
23
24        if (foundIt) {
25            System.out.println("Found " + searchfor + " at " + i + ", " + j);
26        } else {
27            System.out.println(searchfor + " not in the array");
28        }
29    }
30 }
31 }
```

This is the output of the program.




```
Command window
1 Found 12 at 1, 0
```

The **break** statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

14.6 The Continue Statement

The **continue** statement skips the current iteration of a **for**, **while**, or **do-while** loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop. The following program, **ContinueDemo**, steps through a **String**, counting the occurrences of the letter **p**. If the current character is not a **p**, the **continue** statement skips the rest of the loop and proceeds to the next character. If it is a **p**, the program increments the letter count.



```
1 class ContinueDemo {
    public static void main(String[] args) {
2        String searchMe = "peter piper picked a " + "peck of pickled peppers"
        ↵ ;
3        int max = searchMe.length();
4        int numPs = 0;

5

6        for (int i = 0; i < max; i++) {
7            // interested only in p's
8            if (searchMe.charAt(i) != 'p')
9                continue;

10
11            // process p's
12            numPs++;
13        }
14        System.out.println("Found " + numPs + " p's in the string.");
15    }
16 }
17 }
```

Here is the output of this program:

Command window

```
1 Found 9 p's in the string.
```

To see this effect more clearly, try removing the `continue` statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 `p`'s instead of 9.

A labeled `continue` statement skips the current iteration of an outer loop marked with the given label. The following example program, `ContinueWithLabelDemo`, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, `ContinueWithLabelDemo`, uses the labeled `test` of `continue` to skip an iteration in the outer loop.



```
1 class ContinueWithLabelDemo {
    public static void main(String[] args) {
2         String searchMe = "Look for a substring in me";
3         String substring = "sub";
4         boolean foundIt = false;
5
6         int max = searchMe.length() - substring.length();
7
8         test:
9         for (int i = 0; i <= max; i++) {
10             int n = substring.length();
11             int j = i;
12             int k = 0;
13             while (n-- != 0) {
14                 if (searchMe.charAt(j++) != substring.charAt(k++)) {
15                     continue test;
16                 }
17             }
18             foundIt = true;
19             break test;
20         }
21         System.out.println(foundIt ? "Found it" : "Didn't find it");
22     }
23 }
```

Here is the output from this program.



14.7 The Return Statement

The next branching statements is the **return** statement. The **return** statement exits from the current method, and control flow returns to where the method was invoked. The **return** statement has two forms: one that returns a value, and one that does not. To return a value, simply put the value (or an expression that calculates the value) after the **return** keyword.



```
1 return ++count;
```

The data type of the returned value must match the type of the method's declared **return** value. When a method is declared **void**, use the form of **return** that doesn't return a value.



```
1 return;
```

The Classes and Objects section will cover everything you need to know about writing methods.

14.8 The Yield Statement

The last branching statement is the **yield** statement. The **yield** statement exits from the current **switch** expression it is in. A **yield** statement is always followed by an expression that must produce a value. This expression must not be **void**. The value of this expression is the value produced by the enclosing **switch** expression.

Here is an example of a **yield** statement.



```
1 class Test {  
    enum Day {  
3        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
    }  
5  
    public String calculate(Day d) {  
7        return switch (d) {  
            case SATURDAY, SUNDAY -> "week-end";  
9            default -> {  
                int remainingWorkDays = 5 - d.ordinal();  
11               yield remainingWorkDays;  
            }  
13        };  
    }  
15 }
```

15 Branching with Switch Statements

15.1 Using Switch Statements to Control the Flow of Your Program

The `switch` statement is one of the five control flow statements available in the Java language. It allows for any number of execution path. A `switch` statement takes a selector variable as an argument and uses the value of this variable to choose the path that will be executed.

You must choose the type of your selector variable among the following types:

- `byte`, `short`, `char`, and `int` primitive data types
- `Character`, `Byte`, `Short`, and `Integer` wrapper types
- enumerated types
- the `String` type.

It is worth noting that the following primitive types cannot be used for the type of your selector variable: `boolean`, `long`, `float`, and `double`.

Let us see a first example of a **switch** statement.



```
1 int quarter = ...; // any value

3 String quarterLabel = null;
  switch (quarter) {
5     case 0:
        quarterLabel = "Q1 - Winter";
7         break;
        case 1:
9             quarterLabel = "Q2 - Spring";
            break;
11         case 2:
            quarterLabel = "Q3 - Summer";
13             break;
            case 3:
15                 quarterLabel = "Q3 - Summer";
                    break;
17         default:
            quarterLabel = "Unknown quarter";
19     };
```

The body of a **switch** statement is known as a **switch** block. A statement in the **switch** block can be labeled with one or more **case** or **default** labels. The **switch** statement evaluates its expression, then executes all statements that follow the matching **case** label.

You may have noticed the use of the **break** keyword. Each **break** statement terminates the enclosing **switch** statement. Control flow continues with the first statement following the **switch** block. The **break** statements are necessary because without them, statements in **switch** blocks fall through. All statements after the matching **case** label are executed in sequence, regardless of the expression of subsequent **case** labels, until a **break** statement is encountered.

The following code uses fall through to fill the **futureMonths** list.



```
1 int month = 8;
  List<String> futureMonths = new ArrayList<>();
3
  switch (month) {
```



```
5  case 1:
    futureMonths.add("January");
7  case 2:
    futureMonths.add("February");
9  case 3:
    futureMonths.add("March");
11 case 4:
    futureMonths.add("April");
13 case 5:
    futureMonths.add("May");
15 case 6:
    futureMonths.add("June");
17 case 7:
    futureMonths.add("July");
19 case 8:
    futureMonths.add("August");
21 case 9:
    futureMonths.add("September");
23 case 10:
    futureMonths.add("October");
25 case 11:
    futureMonths.add("November");
27 case 12:
    futureMonths.add("December");
29     break;
    default:
31     break;
}
```

Technically, the final `break` is not required because flow falls out of the `switch` statement. Using a `break` is recommended so that modifying the code is easier and less error prone.

The `default` section handles all values that are not explicitly handled by one of the `case` sections.

The following code example, shows how a statement can have multiple `case` labels. The code example calculates the number of days in a particular month:



```
int month = 2;
2 int year = 2021;
```



```
int numDays = 0;

4
switch (month) {
6   case 1: case 3: case 5:    // January March May
    case 7: case 8: case 10:  // July August October
8   case 12:
        numDays = 31;
10    break;
    case 4: case 6:          // April June
12    case 9: case 11:        // September November
        numDays = 30;
14    break;
    case 2: // February
16    if (((year % 4 == 0) &&
        !(year % 100 == 0))
18        || (year % 400 == 0))
        numDays = 29;
20    else
        numDays = 28;
22    break;
    default:
24    System.out.println("Invalid month.");
        break;
26 }
```

This code has one statement for more than one `case`.

15.2 Choosing Between Switch Statements and If-then-else Statements

Deciding whether to use `if-then-else` statements or a `switch` statement is based on readability and the expression that the statement is testing. An `if-then-else` statement can test expressions based on ranges of values or conditions, whereas a `switch` statement tests expressions based only on a single integer, enumerated value, or `String` object.

For instance, the following code could be written with a `switch` statement.



```
int month = ...; // any month
2 if (month == 1) {
    System.out.println("January");
4 } else if (month == 2) {
    System.out.println("February");
6 } ... // and so on
```

On the other hand the following could not be written with a **switch** statement, because **switch** statements do not support labels of type **boolean**.



```
int temperature = ...; // any temperature
2 if (temperature < 0) {
    System.out.println("Water is ice");
4 } else if (temperature < 100){
    System.out.println("Water is liquid , known as water");
6 } else {
    System.out.println("Water is vapor");
8 }
```

15.3 Using String as a Type for the Case Labels

In Java SE 7 and later, you can use a **String** object in the **switch** statement's expression. The following code example displays the number of the month based on the value of the **String** named month.



```
String month = ...; // any month
2 int monthNumber = -1;

4 switch (month.toLowerCase()) {
    case "january":
6         monthNumber = 1;
        break;
8     case "february":
        monthNumber = 2;
```




```
10     break;
    case "march":
12         monthNumber = 3;
        break;
14     case "april":
        monthNumber = 4;
16         break;
        case "may":
18             monthNumber = 5;
            break;
20     case "june":
        monthNumber = 6;
22         break;
        case "july":
24             monthNumber = 7;
            break;
26     case "august":
        monthNumber = 8;
28         break;
        case "september":
30             monthNumber = 9;
            break;
32     case "october":
        monthNumber = 10;
34         break;
        case "november":
36             monthNumber = 11;
            break;
38     case "december":
        monthNumber = 12;
40         break;
        default:
42             monthNumber = 0;
            break;
44 }
```

The `String` in the `switch` expression is compared with the expressions associated with each `case` label as if the `String.equals()` method were being used. In order for this example to accept any month regardless of case, `month` is converted to lowercase (with the `toLowerCase()` method), and all the strings associated with the `case` labels are in lowercase.

15.4 Null Selector Variables

The selector variable of a `switch` statement can be an object, so this object can be null. You should protect your code from null selector variables, because in this case the switch statement will throw a `NullPointerException`.

16 Branching with Switch Expressions

16.1 Modifying the Switch Syntax

In Java SE 14 you can use another, more convenient syntax for the `switch` keyword: the `switch` expression.

Several things have motivated this new syntax.

1. The default control flow behavior between switch labels is to fall through. This syntax is error-prone and leads to bugs in applications.
2. The `switch` block is treated as one block. This may be an impediment in the case where you need to define a variable only in one particular `case`.
3. The `switch` statement is a statement. In the examples of the previous sections, a variable is given a value in each `case`. Making it an expression could lead to better and more readable code.

The syntax covered in the previous section, known as *switch statement* is still available in Java SE 14 and its semantics did not change. Starting with Java SE 14 a new syntax for the `switch` is available: the *switch expression*.

This syntax modifies the syntax of the switch label. Suppose you have the following *switch statement* in your application.



```
int day = ...; // any day
2 int len = 0;
  switch (day) {
4   case MONDAY:
```



```
case FRIDAY:
6 case SUNDAY:
    len = 6;
8 break;
case TUESDAY:
10 len = 7;
    break;
12 case THURSDAY:
case SATURDAY:
14 len = 8;
    break;
16 case WEDNESDAY:
    len = 9;
18 break;
}
20 System.out.println("len = " + len);
```

With the *switch expression* syntax, you can now write it in the following way.



```
int day = ...; // any day
2 int len =
switch (day) {
4 case MONDAY, FRIDAY, SUNDAY -> 6;
case TUESDAY -> 7;
6 case THURSDAY, SATURDAY -> 8;
case WEDNESDAY -> 9;
8 }
System.out.println("len = " + len);
```

The syntax of switch label is now **case L ->**. Only the code to the right of the label is executed if the label is matched. This code may be a single expression, a block, or a throw statement. Because this code is one block, you can define variables in it that are local to this particular block.

This syntax also supports multiple constants per case, separated by commas, as shown on the previous example.

16.2 Producing a Value

This switch statement can be used as an expression. For instance, the example of the previous section can be rewritten with a switch statement in the following way.



```
1 int quarter = ...; // any value

3 String quarterLabel =
    switch (quarter) {
5     case 0 -> "Q1 - Winter";
6     case 1 -> "Q2 - Spring";
7     case 2 -> "Q3 - Summer";
8     case 3 -> "Q3 - Summer";
9     default -> "Unknown quarter";
    };
```

If there is only one statement in the `case` block, the value produced by this statement is returned by the `switch` expression.

The syntax in the case of a block of code is a little different. Traditionally, the `return` keyword is used to denote the value produced by a block of code. Unfortunately this syntax leads to ambiguity in the case of the switch statement. Let us consider the following example. This code does not compile, it is just there as an example.



```
// Be careful, this code does NOT compile!
2 public String convertToLabel(int quarter) {
    String quarterLabel =
4     switch (quarter) {
        case 0 -> {
6         System.out.println("Q1 - Winter");
            return "Q1 - Winter";
8         };
        default -> "Unknown quarter";
10    };
    }
12    return quarterLabel;
}
```

The block of code executed in the case where `quarter` is equal to 0 needs to return a value. It uses the `return` keyword to denote this value. If you take a close look at this code, you see that there are two `return` statements: one in the `case` block, and another one in the method block. This is where the ambiguity lies: one may be wondering what is the semantics of the first `return`. Does it mean that the program exits the method with this value? Or does it leave the `switch` statement? Such ambiguities lead to poor readability and error-prone code.

A new syntax has been created to solve this ambiguity: the `yield` statement. The code of the previous example should be written in the following way.



```
1 public String convertToLabel(int quarter) {  
    String quarterLabel =  
3     switch (quarter) {  
        case 0 -> {  
5             System.out.println("Q1 - Winter");  
             yield "Q1 - Winter";  
7         };  
        default -> "Unknown quarter";  
9     };  
    }  
11    return quarterLabel;  
}
```

The `yield` statement is a statement that can be used in any `case` block of a `switch` statement. It comes with a value, that becomes the value of the enclosing `switch` statement.

16.3 Adding a Default Clause

Default clauses allow your code to handle cases where the selector value does not match any `case` constant.

The cases of a switch expression must be exhaustive. For all possible values, there must be a matching switch label. Switch statements are not required to be exhaustive. If the selector target does not match any switch label, this switch statement will not do anything, silently. This may be a place for bugs to hide in your application, something you want to avoid.

In most of the cases, exhaustiveness can be achieved using a `default` clause; however, in the case of an `enum switch` expression that covers all known constants, you do not need to add

this **default** clause.

There is still a case that needs to be dealt with. What would happen if someone adds an enumerated value in an enumeration, but forget to update the switch statements on this enumeration? To handle this case, the compiler adds a **default** clause for you in exhaustive switch statements. This **default** clause will never be executed in normal cases. It will be only if an enumerated value has been added, and will throw an **IncompatibleClassChangeError**.

Handling exhaustiveness is a feature of **switch** expressions that is not provided by traditional **switch** statements and that is used in other cases than **switch** on enumerated values.

16.4 Writing Colon Case in Switch Expressions

A **switch** expression can also use a traditional **case** block with **case L:**. In this case the fall through semantics does apply. Values are yielded using the **yield** statement.



```
1  int quarter = ...; // any value
2
3  String quarterLabel =
4  switch (quarter) {
5      case 0 : yield "Q1 - Winter";
6      case 1 : yield "Q2 - Spring";
7      case 2 : yield "Q3 - Summer";
8      case 3 : yield "Q3 - Summer";
9      default: System.out.println("Unknown quarter");
10     yield "Unknown quarter";
11 };
```

16.5 Dealing with Null Values

So far, **switch** statements do not accept null selector values. If you try to **switch** on a null value you will get a **NullPointerException**.

Java SE 17 has a preview feature that enhances **switch** expressions to allow for null values, so you can expect this situation to change.