**Java Language Basics**

**Java Language**

In the Java programming language, a keyword is one of 50 reserved words that have a predefined meaning in the language; because of this, programmers cannot use keywords as names for variables, methods, classes, or as any other identifier.

Java key words - <https://en.wikipedia.org/wiki/List_of_Java_keywords>

**Variables**

As you learned in the previous lesson, an object stores its state in fields.

* int cadence = 0;
* int speed = 0;
* int gear = 1;

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". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

**Naming - Naming Conventions &**

**CamelCase**

Alternatively referred to as bicapitalisation, medial capitals, and Pascal case, the term CamelCase describes a compound word which uses capital letters to delimit the word parts. (The name refers to the internal capital letters, which resemble the humps on a camel's back.) For example, "ComputerHope", "FedEx", and "WordPerfect" are all examples of CamelCase. In computer programming CamelCase is often used as a naming convention for naming variables, arrays, and other elements.

 

**Hungarian Notation**

Hungarian notation. Hungarian notation is a naming convention in computer programming that indicates either the type of object or the way it should be used.

**Examples**

* bBusy : [boolean](https://en.wikipedia.org/wiki/Boolean_datatype%22%20%5Co%20%22Boolean%20datatype)
* chInitial : [char](https://en.wikipedia.org/wiki/Character_%28computing%29)
* cApples : count of items
* dwLightYears : double [word](https://en.wikipedia.org/wiki/Word_%28data_type%29) (Systems)
* fBusy : [flag](https://en.wikipedia.org/wiki/Boolean_data_type) (or [float](https://en.wikipedia.org/wiki/Floating-point))
* nSize : [integer](https://en.wikipedia.org/wiki/Integer_%28computer_science%29) (Systems) or count (Apps)
* iSize : [integer](https://en.wikipedia.org/wiki/Integer_%28computer_science%29) (Systems) or index (Apps)
* fpPrice: [floating-point](https://en.wikipedia.org/wiki/Floating-point)
* db[Pi](https://en.wikipedia.org/wiki/Pi) : [double](https://en.wikipedia.org/wiki/Double_precision) (Systems)

**Naming Identifiers/Variables**

Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "\_". The convention, however, is to always begin your variable names with a letter, not "$" or "\_". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "\_", this practice is discouraged. White space is not permitted.

Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields named cadence, speed, and gear, for example, are much more intuitive than abbreviated versions, such as s, c, and g. Also keep in mind that the name you choose must not be a keyword or reserved word.

If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names gearRatio and currentGear are prime examples of this convention. If your variable stores a constant value, such as static final int NUM\_GEARS = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

|  |  |
| --- | --- |
| Samples of acceptable variable names:   **YES** |  Samples of unacceptable variable names:   **NO** |
| Grade  | Grade(Test) |
| GradeOnTest | GradeTest#1 |
| Grade\_On\_Test | 3rd\_Test\_Grade |
| GradeTest | Grade Test   (has a space) |

**Primitive Data 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've already seen:

* int gear = 1;

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, divideUnsigned etc 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).

**String**

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, 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'll probably tend to think of it as such. You'll learn more about the String class in Simple Data Objects

**Default Values**

It's 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 chart 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.

**Literals**

You may have noticed that the new keyword isn't 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's possible to assign a literal to a variable of a primitive type:

* boolean result = **true**;
* char capitalC = '**C**';
* byte b = **100**;
* short s = **10000**;
* int i = **100000**;

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'll 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:

// The number 26, in decimal

int decVal = 26;

// The number 26, in hexadecimal

int hexVal = 0x1a;

// The number 26, in binary

int binVal = 0b11010;

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;

// same value as d1, but in scientific notation

double d2 = 1.234e2;

float f1 = 123.4f;

**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's also a special null literal that can be used as a value for any reference type. null may be assigned to any variable, except variables of primitive types. There's 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's 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 (of type Class) that represents the type itself.

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:

long creditCardNumber = 1234\_5678\_9012\_3456L;

long socialSecurityNumber = 999\_99\_9999L;

float pi = 3.14\_15F;

long hexBytes = 0xFF\_EC\_DE\_5E;

long hexWords = 0xCAFE\_BABE;

long maxLong = 0x7fff\_ffff\_ffff\_ffffL;

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 (which are highlighted) in numeric literals:

// Invalid: cannot put underscores

// adjacent to a decimal point

float pi1 = 3\_.1415F;

// Invalid: cannot put underscores

// adjacent to a decimal point

float pi2 = 3.\_1415F;

// Invalid: cannot put underscores

// prior to an L suffix

long socialSecurityNumber1 = 999\_99\_9999\_L;

// OK (decimal literal)

int x1 = 5\_2;

// Invalid: cannot put underscores

// At the end of a literal

int x2 = 52\_;

// OK (decimal literal)

int x3 = 5\_\_\_\_\_\_\_2;

// Invalid: cannot put underscores

// in the 0x radix prefix

int x4 = 0\_x52;

// Invalid: cannot put underscores

// at the beginning of a number

int x5 = 0x\_52;

// OK (hexadecimal literal)

int x6 = 0x5\_2;

// Invalid: cannot put underscores

// at the end of a number

int x7 = 0x52\_;

**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.

Illustration of an array as 10 boxes numbered 0 through 9; an index of 0 indicates the first element in the array

An array of 10 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 9th element, for example, would therefore be accessed at index 8.

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

class ArrayDemo {

 public static void main(String[] args) {

 // declares an array of integers

 int[] anArray;

 // allocates memory for 10 integers

 anArray = new int[10];

 // initialize first element

 anArray[0] = 100;

 // initialize second element

 anArray[1] = 200;

 // and so forth

 anArray[2] = 300;

 anArray[3] = 400;

 anArray[4] = 500;

 anArray[5] = 600;

 anArray[6] = 700;

 anArray[7] = 800;

 anArray[8] = 900;

 anArray[9] = 1000;

 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]);

 System.out.println("Element at index 3: "

 + anArray[3]);

 System.out.println("Element at index 4: "

 + anArray[4]);

 System.out.println("Element at index 5: "

 + anArray[5]);

 System.out.println("Element at index 6: "

 + anArray[6]);

 System.out.println("Element at index 7: "

 + anArray[7]);

 System.out.println("Element at index 8: "

 + anArray[8]);

 System.out.println("Element at index 9: "

 + anArray[9]);

 }

}

The output from this program is:

Element at index 0: 100

Element at index 1: 200

Element at index 2: 300

Element at index 3: 400

Element at index 4: 500

Element at index 5: 600

Element at index 6: 700

Element at index 7: 800

Element at index 8: 900

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. You will learn about the various looping constructs (for, while, and do-while) in the Control Flow section.

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

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 previously discussed in the naming section. 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;

short[] anArrayOfShorts;

long[] anArrayOfLongs;

float[] anArrayOfFloats;

double[] anArrayOfDoubles;

boolean[] anArrayOfBooleans;

char[] anArrayOfChars;

String[] anArrayOfStrings;

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

// this form is discouraged

float anArrayOfFloats[];

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

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

anArray = new int[10];

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

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

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

anArray[0] = 100; // initialize first element

anArray[1] = 200; // initialize second element

anArray[2] = 300; // and so forth

Each array element is accessed by its numerical index:

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

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

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

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

int[] anArray = {

 100, 200, 300,

 400, 500, 600,

 700, 800, 900, 1000

};

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:

class MultiDimArrayDemo {

 public static void main(String[] args) {

 String[][] names = {

 {"Mr. ", "Mrs. ", "Ms. "},

 {"Smith", "Jones"}

 };

 // Mr. Smith

 System.out.println(names[0][0] + names[1][0]);

 // Ms. Jones

 System.out.println(names[0][2] + names[1][1]);

 }

}

The output from this program is:

Mr. Smith

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);

Copying Arrays

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

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 char elements, spelling the word "decaffeinated." It uses the System.arraycopy method to copy a subsequence of array components into a second array:

class ArrayCopyDemo {

 public static void main(String[] args) {

 char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',

 'i', 'n', 'a', 't', 'e', 'd' };

 char[] copyTo = new char[7];

 System.arraycopy(copyFrom, 2, copyTo, 0, 7);

 System.out.println(new String(copyTo));

 }

}

The output from this program is:

caffein

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 copyOfRange method of the java.util.Arrays class, as you can see in the ArrayCopyOfDemo example. The difference is that using the copyOfRange method does not require you to create the destination array before calling the method, because the destination array is returned by the method:

class ArrayCopyOfDemo {

 public static void main(String[] args) {

 char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',

 'i', 'n', 'a', 't', 'e', 'd'};

 char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

 System.out.println(new String(copyTo));

 }

}

As you can see, the output from this program is the same (caffein), although it requires fewer lines of code. Note that the second parameter of the copyOfRange 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 character a).

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.

**Control Flow Statements**

The statements inside your source files are generally executed from top to bottom, in the order that they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.}

**Difference between variable and field**

If a class or instance variable changes, the state of the object changes. So we can say that class variables and instance variables are fields while local variables and parameter variables are not.

**Summary of Variables**

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.