## Chapter 1

## [Java Tutorial - Variables and Assignment Statements](http://javadevnotes.com/java-tutorial-variables-and-assignment-statements)

Variables are a fundamental component of any programming language. This tutorial will give a brief introduction to variables in Java.

### Introduction

Variables are container for data that are stored in computer's memory. When variables are created, memory space are reserved for storage.

Here is a sample code that uses a variable:

public class Sample {

public static void main(String[] args) {

int num = 10;

System.out.println("The number is: "+ num);

}

}

When we run this, we get the output:

The number is: 10

Here is the code section from above that declares a variable.

int num = 10;

* **int** is the type of the variable declared. In this case, *int* stands for integer. Integers are whole numbers (no decimal point) and can either be negative, positive, or zero.
* **num** is the name of the variable.
* **=** is the assignment operator. It tells to store the value 10 to the variable num. Hence, after the statement, num gets the value of 10.

### Different variables refers to different data

Consider this example:

public class Sample {

public static void main(String[] args) {

int a = 50;

int b = 75;

System.out.println("The first number is: "+ a);

System.out.println("The second number is: "+ b);

}

}

The output of this is:

The first number is: 50

The second number is: 75

* **a** is a variable of type **int** and assigned a value of 50
* **b** is another variable of type **int** and assigned a value of 75
* **a** and **b** are different from each other because they have different names

### Manipulating variables

The value stored to a variable can be updated. Consider this example:

public class Sample {

public static void main(String[] args) {

int num = 20;

num = num + 5;

System.out.println("The result is: "+ num);

}

}

The output of this is:

The result is: 25

* **num** is a variable of type **int** and assigned a value of 20
* **num**'s value is updated with the statement *num = num + 5*
  + num + 5 is evaluated first
  + num is 20, by adding 5 the outcome is 25
  + 25 is then assigned to num
  + after the statement, num gets the value 25

### Case Sensitivity

Consider this example:

public class Sample {

public static void main(String[] args) {

int a = 3;

int A = 7;

System.out.println("The first number is: "+ a);

System.out.println("The second number is: "+ A);

}

}

The output of this is:

The first number is: 3

The second number is: 7

* **a** and **A** are different from each other. This is because Java is case sensitive.
* **a** has the vale of 3
* **A** has the vale of 7

### Keywords and reserved words

In Java, there are words that have special meaning. They are usually called keywords or reserved words. Variables must **NOT** use keywords or reserved words as name.   
You can **NOT** use these as variable names:   
**abstract**, **assert**, **boolean**, **break**, **byte**, **case**, **catch**, **char**, **class**, **const**, **continue**, **default**, **do**, **double**, **else**, **enum**, **extends**, **final**, **finally**, **float**, **for**, **goto**, **if**, **implements**, **import**, **instanceof**, **int**, **interface**, **long**, **native**, **new**, **package**, **private**, **protected**, **public**, **return**, **short**, **static**, **strictfp**, **super**, **switch**, **synchronized**, **this**, **throw**, **throws**, **transient**, **try**, **void**, **volatile**, **while**

### Valid variable names

Here are the rules for naming variables:

* Must be one or more characters long
* First character must either be:
  + letter (a to z and A to Z)
  + underscore (**\_**)
  + dollar sign (**$**)
* Succeeding characters can be:
  + letter (a to z and A to ZZ)
  + underscore (**\_**)
  + dollar sign (**$**)
  + number (0 to 9)

**Sample of valid variable names:**

* num - starts with a letter and succeeding characters are letters
* \_test01 - starts with an underscore and succeeding characters are letters/numbers
* $hello - starts with a dollar sign and succeeding characters are letters

**Sample of invalid variable names:**

* 100percentile - can't start with a number
* test# - can't use # as part of a name
* byte - can't use a reserved word.

## Chapter 2

## [Java Tutorial - Primitive Types](http://javadevnotes.com/java-tutorial-primitive-types)

Primitive types are the most basic data types in Java that are used to store simple values. They can be used directly in programs or as a building block to create more complex data types.

### Primitive Types

* **byte** - 8 bit signed integer
* **short** - 16 bit signed integer
* **int** - 32 bit signed integer
* **long** - 64 bit signed integer
* **float** - 32 bit floating point number that can have decimal places
* **double** - 64 bit floating point number that can have decimal places
* **boolean** - represents logical operation that can either be true or false
* **char** - represents a single unicode (16 bit) character.

As you can observe, these types are for storing simple values. These can also be seen in other programming languages.

### byte, short, int, long

The first four data types (**byte**, **short**, **int**, and **long**) are very similar to each other.

* Represents integer (whole numbers) only. Values can be zero, negative, or positive numbers.
* Difference between the four data types is the number of bits needed to represent their values.
* **byte** uses 8 bit of memory
  + Minimum value is -128
  + Maximum value is 127
  + Ideal for representing small numbers as it consumes the least amount of memory
  + Example:
  + byte a = -10;
  + byte b = 0;

byte c = 100;

* **short** uses 16 bit of memory
  + Minimum value is -32,768
  + Maximum value is 32,767
  + Can be used if byte is not enough for the possible values to be assigned
  + Example:
  + short a = -19000;
  + short b = 0;

short c = 21000;

* **int** uses 32 bit of memory
  + Minimum value is -2,147,483,648
  + Maximum value is 2,147,483,647
  + Most commonly used data type for representing numbers
  + Example:
  + int a = -7111555;
  + int b = 0;

int c = 5200300;

* **long** uses 64 bit of memory
  + Minimum value is -9,223,372,036,854,775,808
  + Maximum value is 9,223,372,036,854,775,807
  + Not commonly used compared with others because it uses more memory. Useful when very large range of values is needed
  + Example:
  + long a = -9800900200l;
  + long b = 0;

long c = 5300100700l;

Note that you have to add **l** at the end of a very large number to tell Java it's a literal long value.

It is possible to assign values from different numeric data types. As long as the result is stored to a variable with enough number of bits. For example:

byte a = 8;

int b = a;

The above is acceptable beause **a** has 8 bits (byte) while **b** has 32 bits (int). The variable **b** here will also have the value of 8.   
This however is not acceptable.

int a = 8;

byte b = a;

Becase this is trying to store a variable with more bits (32 bits) to a variable with less bits(8 bits). The resulting variable will not be able to handle it.   
You can also mix numeric dat types in calculations. The result will follow the highest number of bits from all the numbers involved. Example:

byte a = 10;

short b = 5;

int c = 200;

int d = a + b + c;

The above code is acceptable. The result of **a + b + c** is automatically of **int** type because operand **c** has the highest number of bits.   
The example below is invalid:

byte a = 10;

short b = 5;

int c = 200;

short d = a + b + c;

Because **a + b + c** is int (32 bits) which can't be assigned to a short (16 bits).

### float and double

The data types **float** and **double** are also similar to each other.

* Represents numbers that can have a decimal place (E.g. 9.75)
* Uses IEEE 754 floating point standard
* Internal representation is similar to a scientific notation.
  + can have very wide range of values
  + trade off with precision. not ideal when accuracy is very important (E.g. money)
* **float** uses 32 bit of memory. More commonly used compared to **double**. Example:

float a = -10.15f;

Note that you have to add **f** at the end of a literal number to denote that it's a literal float.

* **double** uses 64 bit of memory. used for computations that needs higher precision than float. Example:

double a = -200.25d;

Note that you have to add **d** at the end of a literal number to denote that it's a literal double.

Since float has 32 bits and double has 64 bits, it is allowed to assign float values to a double, but it is not allowed to store double value to a float.

This is valid:

float a = 10;

double b = a;

This is invalid:

double b = 20;

float a = b;

Both float and double can store any of the numeric data types byte, short, int, and long. Example:

byte a = 10;

short b = 2000;

int c = 3000000;

long d = 4000000000l;

float f1 = a;

float f2 = b;

float f3 = c;

float f4 = d;

double d1 = a;

double d2 = b;

double d3 = c;

double d4 = d;

When computations contains either a float or a double operand, the result will automatically be either of the two - which ever has the highest number of bits.

This is valid because **a + b** will become float

int a = 5;

float b = 10;

float c = a + b;

This is invalid because **a + b** is float, which is not allowed to be assigned to an int.

int a = 5;

float b = 10;

int c = a + b;

This is valid because **a + b** will become double.

float a = 5;

double b = 10;

double c = a + b;

This is invalid because **a + b** is double (64 bits), which can't be assigned to a float (32 bits).

float a = 5;

double b = 10;

float c = a + b;

### boolean

Boolean is the smallest data type in Java.

* Needs only 1 bit to store the value
* Stores logical information
* Only two possible values: true an false
* Useful for storing flags or information that is answerable with yes/no in the real world.   
  Example :
* boolean isFullyPaid = false;
* boolean hasDriverLicense = true;

boolean isMarried = true;

### char

Char can store a single character.

* can store any English alphabet
* can store a number
* can store a special character. E.g. !, @, #, $, %, ^, &, \*, (, ), ¢, £, ¥
* can store unicode (16 bit) character.
  + Unicode is a computer industry standard for representing text/string data
  + supports most of the writing system in the world.   
    This means it can represent a character languages other than English
* Examples :
* char a = 'x';
* char b = '1';
* char c = '#';
* char d = '£';

char e = '?';

Note that literal char are enclosed with single quotes.

This statement is not valid because char can't have more than 1 character.

char a = 'ab';

## Chapter 3

[**Java Tutorial - Read Input From Console**](http://javadevnotes.com/java-tutorial-read-input-from-console)

**System.console().readLine()**

The simplest way to read user input is by using **System.console()**. Here is an example:

public class Sample {

public static void main(String[] args) {

System.out.print("Please enter your name: ");

String name = System.console().readLine();

System.out.println("Your name is: " + name);

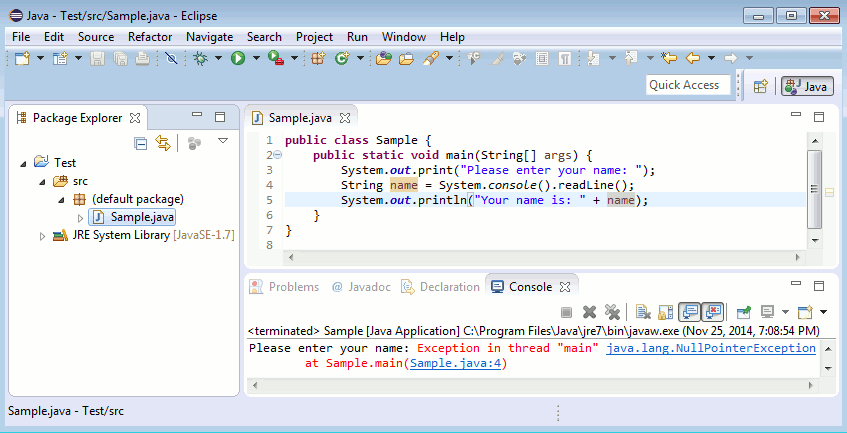
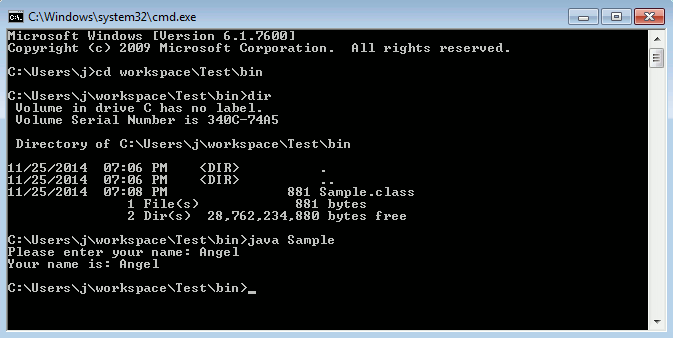
}

}

And this is a sample conversation with the user when the program is run:

Please enter your name: John

Your name is: John

The problem with using this method is that it will not run properly using an [IDE](http://javadevnotes.com/java-tutorial-setup-your-windows-development-environment/) (E.g. Eclipse). Here is a screenshot when you try to run this inside Eclipse:   
[](http://static.javadevnotes.com/wp-content/uploads/2014/11/zcona.png)  
If you wish to use this method, you can run this successfully using a command line. Here is an example:   
[](http://static.javadevnotes.com/wp-content/uploads/2014/11/zconb.png)  
An Eclipse project typically has src and bin subfolder. The former contains the source code (E.g. Test.java) while the latter contains the compiled **.class** files (E.g. Test.class). Just go to the bin folder and execute

java Sample

Where Sample is the name of the class you declared.

**java.util.Scanner and System.in**

Combining System.in and java.util.Scanner provides a way to read user input that can run inside an IDE. It also provides a way to read different data types.

* **read string input**:
* import java.util.Scanner;
* public class Test {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("What is your favorite color? ");
* String name = scanner.next();
* System.out.println("Your favorite color is: " + name);
* }

}

Sample output:

What is your favorite color? blue

Your favorite color is: blue

* **read byte input**:
* import java.util.Scanner;
* public class Test {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter a small number: ");
* byte number = scanner.nextByte();
* System.out.println("The number is: " + number);
* }

}

Sample output:

Enter a small number: 5

The number is: 5

* **read short input**:
* import java.util.Scanner;
* public class Test {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter a short integer: ");
* short number = scanner.nextShort();
* System.out.println("The number is: " + number);
* }

}

Sample output:

Enter a short integer: 1000

The number is: 1000

* **read int input**:
* import java.util.Scanner;
* public class Test {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter an integer: ");
* int number = scanner.nextInt();
* System.out.println("The number is: " + number);
* }

}

Sample output:

Enter an integer: 211555777

The number is: 211555777

* **read long input**:
* import java.util.Scanner;
* public class Test {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter a long number: ");
* long number = scanner.nextLong();
* System.out.println("The number is: " + number);
* }

}

Sample output:

Enter a long number: 12345678912

The number is: 12345678912

* **read float input**:
* import java.util.Scanner;
* public class Test {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter a float number: ");
* float number = scanner.nextFloat();
* System.out.println("The number is: " + number);
* }

}

Sample output:

Enter a float number: 1.25

The number is: 1.25

* **read double input**:
* import java.util.Scanner;
* public class Test {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter a double number: ");
* double number = scanner.nextDouble();
* System.out.println("The number is: " + number);
* }

}

Sample output:

Enter a float number: 55.11

The number is: 55.11

* **read boolean input**:
* import java.util.Scanner;
* public class Test {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter a boolean value (true or false): ");
* boolean bool = scanner.nextBoolean();
* System.out.println("You entered: " + bool);
* }

}

Sample output:

Enter a boolean value (true or false): true

You entered: true

**BufferedReader, InputStreamReader, and System.in**

Here is another example of getting user input using BufferedReader, InputStreamReader, and System.in. This way will also work inside an [IDE](http://javadevnotes.com/java-tutorial-setup-your-windows-development-environment/).

Example code:

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class Test {

public static void main(String[] args) throws IOException {

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

System.out.print("Please enter your name? ");

String name = reader.readLine();

System.out.println("Your name is: " + name);

}

}

And the behavior is the same as the samples above:

Please enter your name? James

Your name is: James

## Chapter - 4

## [Java Initialize Array Examples](http://javadevnotes.com/java-initialize-array-examples)

An array is a type of variable that can hold multiple values of similar data type. This is very useful for storing values when we don't know how many of them is needed, or when the number of values is very large. The first thing we need to know is how to initialize a Java Array. Below are some code samples on how to perform Java Array Initialization.

### Declare Array Without Initialization

Below is an example on how to declare an array in Java without performing initialization:

/\*\*

\* A Simple Example that Declares A Java Array.

\*/

public class DeclareJavaArray {

public static void main(String[] args) {

int[] testArray;

}

}

The code declares a variable testArray to be a one dimensional array of type int. Note that since the array was not initialized, the value of the variable is still null at this point of time. Below is another way of declaring an array:

/\*\*

\* A Simple Example that Declares A Java Array.

\*/

public class DeclareJavaArray {

public static void main(String[] args) {

int testArray[];

}

}

Notice that the square brackets is at the front of the variable. Both using the square brackets before (*int[] testArray;*) and after (*int testArray[];*) are valid.

### Initialize Java Array Using Assignment

When we declare an array, the initial value is null and has no size. Below is one simple way of initializing an array:

import java.util.Arrays;

/\*\*

\* A Simple Example that Initialise A Java Array Using Assignment.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

int[] testArray = new int[4];

System.out.println(Arrays.toString(testArray));

testArray[0] = 2;

testArray[1] = 3;

testArray[2] = 5;

testArray[3] = 7;

System.out.println(Arrays.toString(testArray));

}

}

The code *new int[4];* creates an array of int with size 4. Hence the variable *testArray* can hold up to 4 values of int, because we assigned to it the newly created array. We can then assign values to each item of the array. When we run the code, we will get this output:

[0, 0, 0, 0]

[2, 3, 5, 7]

Notice that the first line of output means that the array contains 0 as value to each of it's item. This is because the individual item of the array is un-initialized. And the primitive int has the default value 0. The second line of output reflects the values assigned to the variable items.

Consider the example below:

import java.util.Arrays;

/\*\*

\* A Simple Example that Initialise A Java Array Using Assignment.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

String[] testArray = new String[4];

System.out.println(Arrays.toString(testArray));

testArray[0] = "Apple";

testArray[1] = "Banana";

testArray[2] = "Coconut";

testArray[3] = "Dewberry";

System.out.println(Arrays.toString(testArray));

}

}

The output of the code will be:

[null, null, null, null]

[Apple, Banana, Coconut, Dewberry]

Notice that the first line outputs all nulls. This is because the code *new String[4]* just creates the array object. Since each item of the array are also objects, they are not yet created and has null value initially. Each item should be assigned a value to avoid having null values.

### Declare And Initialize Java Array In One Statement

If the size of the array you wish to initialize is fairly small and you know what values you want to assign, you may declare and initialize an array in one statement. Below shows an example on how to do it in 4 ways:

import java.util.Arrays;

/\*\*

\* A Simple Example that Declares And Initialise A Java Array In One Go.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

int[] testArray1 = {1, 2, 3, 4, 5};

int testArray2[] = {6, 7, 8, 9, 10};

int[] testArray3 = new int[] {11, 12, 13, 14, 15};

int testArray4[] = new int[] {16, 17, 18, 19, 20};

System.out.println(Arrays.toString(testArray1));

System.out.println(Arrays.toString(testArray2));

System.out.println(Arrays.toString(testArray3));

System.out.println(Arrays.toString(testArray4));

}

}

The code will have the output below when executed:

[1, 2, 3, 4, 5]

[6, 7, 8, 9, 10]

[11, 12, 13, 14, 15]

[16, 17, 18, 19, 20]

Again, this is only practical when the size of array is relatively small.

### Initialize Java Array In One Statement

Below is an example if we want to initialize a Java Array in one statement, but separate from the declaration:

import java.util.Arrays;

/\*\*

\* A Simple Example that Declares And Initialise A Java Array Separately.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

int[] testArray;

testArray = new int[] {5, 7, 11, 13, 17};

System.out.println(Arrays.toString(testArray));

}

}

Note that this code is valid:

testArray = new int[] {5, 7, 11, 13, 17};

While this code is **NOT VALID**:

testArray = {5, 7, 11, 13, 17};

### Initialize Java Array With Same Value

If we have a large Array and wants to initialize each item with the same value we may use the Arrays utility class to help us. Below is a simple example:

import java.util.Arrays;

/\*\*

\* A Simple Example that Initialise A Java Array With The Same Value.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

int[] testArray = new int[10];

Arrays.fill(testArray, 50);

System.out.println(Arrays.toString(testArray));

}

}

The critical code is this:

Arrays.fill(testArray, 50);

Since the value of each item in the array has been initialized with the value of 50, the output of the code will be:

[50, 50, 50, 50, 50, 50, 50, 50, 50, 50]

### Initialize Java Array Using Loops

We may use for loops to initialize individual items of an array. Below is a simple example of how to assign the numbers 1 to 10 to an array with 10 values.

import java.util.Arrays;

/\*\*

\* A Simple Example that Initialise A Java Array With Values 1 to 10.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

int[] testArray = new int[10];

for (int i=0; i<10; i++) {

testArray[i] = i + 1;

}

System.out.println(Arrays.toString(testArray));

}

}

The output of the code will be:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Below is a simple example that fills an array with the first 10 even numbers.

import java.util.Arrays;

/\*\*

\* A Simple Example that Initialise A Java Array With the first 10 even numbers.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

int[] testArray = new int[10];

for (int i=0; i<10; i++) {

testArray[i] = (i + 1) \* 2;

}

System.out.println(Arrays.toString(testArray));

}

}

Below is the output of the code:

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

Below is another example that fills an array with the first 10 prime numbers. The code is a bit long because it is meant to be a more readable algorithm.

import java.util.Arrays;

/\*\*

\* A Simple Example that Initialise A Java Array With the first 10 prime numbers.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

int[] testArray = new int[10];

int prime = 2;

for (int i = 0; i < 10; i++) {

boolean primeFound = false;

do {

int divisors = 0;

for (int j = 2; j < prime - 1; j++) {

if (prime % j == 0) {

divisors++;

}

}

if (divisors <= 1) {

primeFound = true;

} else {

prime++;

}

} while (!primeFound);

testArray[i] = prime;

prime++;

}

System.out.println(Arrays.toString(testArray));

}

}

And the output is below.

[2, 3, 4, 5, 7, 9, 11, 13, 17, 19]

### Initialize Java Array From Console

If we need to get the values of an array from the user, below is a simple example on how to initialize a Java Array with values coming from the console:

import java.util.Arrays;

/\*\*

\* A Simple Example that Initialise A Java Array With Values From Console.

\*/

public class InitializeJavaArray {

public static void main(String[] args) {

int[] testArray = new int[10];

for (int i = 0; i < 10; i++) {

String stringValue = System.console().readLine("Enter number: " );

testArray [ i ] = Integer.parseInt(stringValue);

}

System.out.println(Arrays.toString(testArray));

}

}

This code reads an input from the user via the console:

System.console().readLine("Enter number: " );

And we need to convert the String to an int via:

Integer.parseInt(stringValue);

## [Java Array Length Examples](http://javadevnotes.com/java-array-length-examples)

Dec 22, 2015 [Array](http://javadevnotes.com/category/array), [Core Java](http://javadevnotes.com/category/core-java), [Examples](http://javadevnotes.com/category/examples) 6 comments

An Array in Java can hold arbitrary number of elements, depending on how the array object was created. To perform operations on array, it is fundamental to know it's length. For example, if we wish to iterate to all the values in the array, the length of the Java array will determine how many times we need to loop. This post will show how to get the Java Array's length and show examples on how to use the value.

### Java Array Length

A Java Array Object does not have a method to get it's length. From the [Java Specification](http://docs.oracle.com/javase/specs/jls/se7/html/jls-10.html#jls-10.7), an Array has a "public final length" field that contains the number of it's components. The value can be a positive number or zero. Therefore, in order to get the Java Array Length, your code needs to access the array length attribute. Here is a simple example:

/\*\*

\* A Simple Example to get the Java array length.

\*/

public class JavaArrayLengthTest {

public static void main(String[] args) {

String[] testArray = { "Apple", "Banana", "Carrots" };

int arrayLength = testArray.length;

System.out.println("The length of the array is: " + arrayLength);

}

}

Since the array has three elements, the expected output of the program will be:

The length of the array is: 3

From the code sample, the array length is retrieved from the array length attribute:

int arrayLength = testArray.length;

But often in our code, we need to process an array where we are not sure how the array object was created. Here is a sample code where we write a function that receives an array and print the length of it.

/\*\*

\* A Simple Example that uses a function to print the Java array length.

\*/

public class JavaArrayLengthTest {

private static void printArrayLength(String[] anArray) {

if (anArray == null) {

System.out.println("The length of the array can't be determined.");

} else {

int arrayLength = anArray.length;

System.out.println("The length of the array is: " + arrayLength);

}

}

public static void main(String[] args) {

String[] testArray1 = { "Apple", "Banana", "Carrots" };

String[] testArray2 = { "A", "B" };

String[] testArray3 = { "1", "2", "3", "4" };

String[] testArray4 = { "Test" };

printArrayLength(null);

printArrayLength(testArray1);

printArrayLength(testArray2);

printArrayLength(testArray3);

printArrayLength(testArray4);

}

}

Notice that we check first if the array is null or not. This is because if we access the length field of a null object, a NullPointerException will be raised. Below is the result of the code when executed:

The length of the array can't be determined.

The length of the array is: 3

The length of the array is: 2

The length of the array is: 4

The length of the array is: 1

### Display All Values of an Array

We can use the length of an array to loop through all the elements of an array. We can simply use a for loop using the length as the exclusive upper bound. Here is a simple example:

/\*\*

\* A Simple Example that loops through all the values of an array using the

\* length field.

\*/

public class JavaArrayLengthTest {

private static void printArrayValues(String[] anArray) {

if (anArray == null) {

System.out.println("The array has no elements.");

} else {

int arrayLength = anArray.length;

for (int i = 0; i <= arrayLength - 1; i++) {

String value = anArray[i];

System.out.println("The array contains the value: " + value);

}

}

}

public static void main(String[] args) {

String[] testArray = { "Apple", "Banana", "Carrots" };

printArrayValues(null);

printArrayValues(testArray);

}

}

Since an array is 0 based, the index of the items range from 0 upto "arrayLength - 1", instead of 1 to arrayLength. Below is the result when the code is executed:

The array has no elements.

The array contains the value: Apple

The array contains the value: Banana

The array contains the value: Carrots

### Check if Array Contains Value

Below is another example of how useful an Array's length is. The code checks if an array contains a specific value. The length of the Java array is used to loop through all the items and check if the value is the same.

/\*\*

\* A Simple Example that uses Java Array Length to check if an array contains a

\* specific value.

\*/

public class JavaArrayLengthTest {

private static boolean arrayContainsValue(String[] anArray,

String valueToLookFor) {

if (anArray != null) {

int arrayLength = anArray.length;

for (int i = 0; i <= arrayLength - 1; i++) {

String value = anArray[i];

if (value.equals(valueToLookFor)) {

return true;

}

}

}

return false;

}

public static void main(String[] args) {

String[] testArray = { "Apple", "Banana", "Carrots" };

System.out.println(arrayContainsValue(testArray, "Banana"));

System.out.println(arrayContainsValue(testArray, "Grapes"));

}

}

Since "Banana" is in the array, the first statement is true. While the second statement is false because "Grapes" is not in the array. Below is the result when the code is run:

true

false

### Get Lowest Value in Array

Here is another example on how to use the length of an array to retrieve the lowest value in all the items of an array object:

/\*\*

\* A Simple Example that uses Java Array Length to determine the lowest value in

\* an array.

\*/

public class JavaArrayLengthTest {

private static int minValue(int[] anArray) {

int minValue = anArray[0];

int arrayLength = anArray.length;

for (int i = 1; i <= arrayLength - 1; i++) {

int value = anArray[i];

if (value < minValue) {

minValue = value;

}

}

return minValue;

}

public static void main(String[] args) {

int[] testArray = { 100, 50, 20, 99 };

System.out.println("The min value is: "+minValue(testArray));

}

}

The code assumes that the array is not null and there is at least one element. The result of the code is shown below:

The min value is: 20

### Get Highest Value in Array

Similarly, here is another example on how to use the length of an array to retrieve the highest value in all the items of an array object:

/\*\*

\* A Simple Example that uses Java Array Length to determine the highest value in

\* an array.

\*/

public class JavaArrayLengthTest {

private static int maxValue(int[] anArray) {

int maxValue = anArray[0];

int arrayLength = anArray.length;

for (int i = 1; i <= arrayLength - 1; i++) {

int value = anArray[i];

if (value > maxValue) {

maxValue = value;

}

}

return maxValue;

}

public static void main(String[] args) {

int[] testArray = { 100, 50, 20, 99 };

System.out.println("The max value is: "+maxValue(testArray));

}

}

Below is the rendered result:

The max value is: 100