Contents

What is Java? 8
Why Use Java? 8
Java Syntax 9
Java Syntax 9
The main Method 9
System.out.println() 9
Java Output / Print 10
Print Text 10
Example 10
Example 10
Double Quotes 10
Example 10
The Print() Method 10
Example 10
Java Output Numbers 11
Print Numbers 11
Example 11
Example 11
Example 11
Java Comments 12
Java Comments 12
Single-line Comments 12
Example 12
Example 12
Java Multi-line Comments 12
Example 12
Variables 13
Java Variables 13
Declaring (Creating) Variables 13
Syntax 13
Example 13
Example 13
Example 14
Example 14
Final Variables 14

Example 14
Other Types 14
Example 14
Java Print Variables 15
Display Variables 15
Example 15
Example 15
Example 15
Java Declare Multiple Variables 16
Example 16
One Value to Multiple Variables 16
Example 16
Java Identifiers 17
Identifiers 17
Example 17
Java Data Types 18
Java Data Types 18
Example 18
Primitive Data Types 18
Java Numbers 20
Numbers 20
Integer Types 20
Byte 20
Example 20
Short 20
Example 20
Int 20
Example 21
Long 21
Example 21
Floating Point Types 21
Float Example 21
Double Example 21
Scientific Numbers 21
Example 21
Java Boolean Data Types 23

Boolean Types 23
Example 23
Java Characters 24
Characters 24
Example 24
Example 24
Strings 24
Example 24
Java Non-Primitive Data Types 26
Non-Primitive Data Types 26
Java Type Casting 27
Java Type Casting 27
Widening Casting 27
Example 27
Narrowing Casting 27
Example 27
Java Operators 29
Java Operators 29
Example 29
Example 29
Arithmetic Operators 29
Java Assignment Operators 30
Example 30
Example 30
Java Comparison Operators 31
Example 31
Java Logical Operators 32
Java Strings 33
Java Strings 33
Example 33
String Length 33
Example 33
More String Methods 33
Example 33
Finding a Character in a String 33
Example 33

Java String Concatenation 35
String Concatenation 35
Example 35
Example 35
Java Numbers and Strings 36
Adding Numbers and Strings 36
Example 36
Example 36
Example 36
Java Special Characters 37
Strings - Special Characters 37
Example 37
Example 37
Example 37
Java Math 39
Math.max(*x,y*) 39
Example 39
Math.min(*x,y*) 39
Example 39
Math.sqrt(*x*) 39
Example 39
Math.abs(*x*) 39
Example 39
Random Numbers 40
Example 40
Example 40
Java Booleans 41
Java Booleans 41
Boolean Values 41
Example 41
Boolean Expression 41
Example 41
Example 42
Example 42
Example 42
Real Life Example 42

Example 42
Example 43
Java If ... Else 44
Java Conditions and If Statements 44
The if Statement 44
Syntax 44
Example 44
Example 45
The else Statement 45
Syntax 45
Example 45
The else if Statement 46
Syntax 46
Example 46
Java Short Hand If...Else (Ternary Operator) 48
Short Hand If...Else 48
Syntax 48
Example 48
Example 48
Java Switch 49
Java Switch Statements 49
Syntax 49
Example 49
The break Keyword 50
The default Keyword 51
Example 51
Java While Loop 52
Loops 52
Java While Loop 52
Syntax 52
Example 52
The Do/While Loop 52
Syntax 53
Example 53
Java For Loop 54
Java For Loop 54

Syntax 54
Example 54
Another Example 54
Example 55
Nested Loops 55
Example 55
Java For Each Loop 56
For-Each Loop 56
Syntax 56
Example 56
Java Break and Continue 57
Java Break 57
Example 57
Java Continue 57
Example 57
Break and Continue in While Loop 58
Break Example 58
Continue Example 58
Java Arrays 59
Java Arrays 59
Access the Elements of an Array 59
Example 59
Change an Array Element 59
Example 59
Example 60
Array Length 60
Example 60
Java Arrays Loop 61
Loop Through an Array 61
Example 61
Loop Through an Array with For-Each 61
Syntax 61
Example 61
Java Multi-Dimensional Arrays 63
Multidimensional Arrays 63
Example 63

Access Elements 63
Example 63
Change Element Values 63
Example 63
Loop Through a Multi-Dimensional Array 64
Example 64

```
public class Main {

  public static void main(String[] args) {

    System.out.println("Hello World");

  }

}
```

What is Java?

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than **3 billion** devices run Java.

It is used for:

- Mobile applications (specially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection
- And much, much more!

Why Use Java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming language in the world
- It has a large demand in the current job market
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has a huge community support (tens of millions of developers)
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As Java is close to C++ and C#, it makes it easy for programmers to switch to Java or vice versa

Java Syntax

Java Syntax

In the previous chapter, we created a Java file called `Main.java`, and we used the following code to print "Hello World" to the screen:

Main.java

```
public class Main {

  public static void main(String[] args) {

    System.out.println("Hello World");

  }

}
```

The main Method

The `main()` method is required and you will see it in every Java program:

```
public static void main(String[] args)
```

Any code inside the `main()` method will be executed. Don't worry about the keywords before and after main.

For now, just remember that every Java program has a `class` name which must match the filename, and that every program must contain the `main()` method.

System.out.println()

Inside the `main()` method, we can use the `println()` method to print a line of text to the screen:

```
public static void main(String[] args) {

  System.out.println("Hello World");

}
```

Java Output / Print

Print Text

You learned from the previous chapter that you can use the `println()` method to output values or print text in Java:

Example

```
System.out.println("Hello World!");
```

You can add as many `println()` methods as you want. Note that it will add a new line for each method:

Example

```
System.out.println("Hello World!");

System.out.println("I am learning Java.");

System.out.println("It is awesome!");
```

Double Quotes

When you are working with text, it must be wrapped inside double quotations marks `""`.

If you forget the double quotes, an error occurs:

Example

```
System.out.println("This sentence will work!");

System.out.println(This sentence will produce an error);
```

The Print() Method

There is also a `print()` method, which is similar to `println()`.

The only difference is that it does not insert a new line at the end of the output:

Example

```
System.out.print("Hello World! ");

System.out.print("I will print on the same line.");
```

Java Output Numbers

Print Numbers

You can also use the `println()` method to print numbers.

However, unlike text, we don't put numbers inside double quotes:

Example

```
System.out.println(3);

System.out.println(358);

System.out.println(50000);
```

You can also perform mathematical calculations inside the `println()` method:

Example

```
System.out.println(3 + 3);
```

Example

```
System.out.println(2 * 5);
```

Java Comments

Java Comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line Comments

Single-line comments start with two forward slashes (`//`).

Any text between `//` and the end of the line is ignored by Java (will not be executed).

This example uses a single-line comment before a line of code:

Example

```
// This is a comment

System.out.println("Hello World");
```

This example uses a single-line comment at the end of a line of code:

Example

```
System.out.println("Hello World"); // This is a comment
```

Java Multi-line Comments

Multi-line comments start with `/*` and ends with `*/`.

Any text between `/*` and `*/` will be ignored by Java.

This example uses a multi-line comment (a comment block) to explain the code:

Example

```
/* The code below will print the words Hello World

to the screen, and it is amazing */

System.out.println("Hello World");
```

Variables

Java Variables

Variables are containers for storing data values.

In Java, there are different **types** of variables, for example:

- `String` - stores text, such as "Hello". String values are surrounded by double quotes
- `int` - stores integers (whole numbers), without decimals, such as 123 or -123
- `float` - stores floating point numbers, with decimals, such as 19.99 or -19.99
- `char` - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- `boolean` - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, you must specify the type and assign it a value:

Syntax

```
type variableName = value;
```

Where *type* is one of Java's types (such as `int` or `String`), and *variableName* is the name of the variable (such as **x** or **name**). The **equal sign** is used to assign values to the variable.

To create a variable that should store text, look at the following example:

Example

Create a variable called **name** of type `String` and assign it the value "**John**":

```
String name = "John";

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

To create a variable that should store a number, look at the following example:

Example

Create a variable called **myNum** of type `int` and assign it the value **15**:

```
int myNum = 15;

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

You can also declare a variable without assigning the value, and assign the value later:

Example

```
int myNum;

myNum = 15;

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

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

Example

Change the value of `myNum` from `15` to `20`:

```
int myNum = 15;

myNum = 20;  // myNum is now 20

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

Final Variables

If you don't want others (or yourself) to overwrite existing values, use the `final` keyword (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

Example

```
final int myNum = 15;

myNum = 20;  // will generate an error: cannot assign a value to a final variable
```

Other Types

A demonstration of how to declare variables of other types:

Example

```
int myNum = 5;

float myFloatNum = 5.99f;

char myLetter = 'D';

boolean myBool = true;

String myText = "Hello";
```

Java Print Variables

Display Variables

The `println()` method is often used to display variables.

To combine both text and a variable, use the `+` character:

Example

```
String name = "John";

System.out.println("Hello " + name);
```

You can also use the `+` character to add a variable to another variable:

Example

```
String firstName = "John ";

String lastName = "Doe";

String fullName = firstName + lastName;

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

For numeric values, the `+` character works as a mathematical operator (notice that we use `int` (integer) variables here):

Example

```
int x = 5;

int y = 6;

System.out.println(x + y); // Print the value of x + y
```

From the example above, you can expect:

- x stores the value 5
- y stores the value 6
- Then we use the `println()` method to display the value of x + y, which is **11**

Java Declare Multiple Variables

Declare Many Variables

To declare more than one variable of the **same type**, you can use a comma-separated list:

Example

Instead of writing:

```
int x = 5;

int y = 6;

int z = 50;

System.out.println(x + y + z);
```

You can simply write:

```
int x = 5, y = 6, z = 50;

System.out.println(x + y + z);
```

One Value to Multiple Variables

You can also assign the **same value** to multiple variables in one line:

Example

```
int x, y, z;

x = y = z = 50;

System.out.println(x + y + z);
```

Java Identifiers

Identifiers

All Java **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

Example

```
// Good

int minutesPerHour = 60;



// OK, but not so easy to understand what m actually is

int m = 60;
```

The general rules for naming variables are:

- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names can also begin with $ and _
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Java keywords, such as `int` or `boolean`) cannot be used as names

Java Data Types

Java Data Types

As explained in the previous chapter, a variable in Java must be a specified data type:

Example

```
int myNum = 5;               // Integer (whole number)

float myFloatNum = 5.99f;    // Floating point number

char myLetter = 'D';         // Character

boolean myBool = true;       // Boolean

String myText = "Hello";     // String
```

Data types are divided into two groups:

- Primitive data types - includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`
- Non-primitive data types - such as `String`, Arrays and Classes (you will learn more about these in a later chapter)

Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647

`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
`boolean`	1 bit	Stores true or false values
`char`	2 bytes	Stores a single character/letter or ASCII values

Java Numbers

Numbers

Primitive number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `byte`, `short`, `int` and `long`. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. There are two types: `float` and `double`.

Even though there are many numeric types in Java, the most used for numbers are `int` (for whole numbers) and `double` (for floating point numbers). However, we will describe them all as you continue to read.

Integer Types

Byte

The `byte` data type can store whole numbers from -128 to 127. This can be used instead of `int` or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example

```
byte myNum = 100;

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

Short

The `short` data type can store whole numbers from -32768 to 32767:

Example

```
short myNum = 5000;

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

Int

The `int` data type can store whole numbers from -2147483648 to 2147483647. In general, the `int` data type is the preferred data type when we create variables with a numeric value.

Example

```
int myNum = 100000;

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

Long

The `long` data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

Example

```
long myNum = 15000000000L;

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

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

The `float` and `double` data types can store fractional numbers. Note that you should end the value with an "f" for floats and "d" for doubles:

Float Example

```
float myNum = 5.75f;

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

Double Example

```
double myNum = 19.99d;

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

Use `float` or `double`?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of `float` is only six or seven decimal digits, while `double` variables have a precision of about 15 digits. Therefore it is safer to use `double` for most calculations.

Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

Example

```
float f1 = 35e3f;

double d1 = 12E4d;

System.out.println(f1);

System.out.println(d1);
```

Java Boolean Data Types

Boolean Types

Very often in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, Java has a `boolean` data type, which can only take the values `true` or `false`:

Example

```
boolean isJavaFun = true;

boolean isFishTasty = false;

System.out.println(isJavaFun);     // Outputs true

System.out.println(isFishTasty);   // Outputs false
```

Boolean values are mostly used for conditional testing.

Java Characters

Characters

The `char` data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

Example

```
char myGrade = 'B';

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

Alternatively, if you are familiar with ASCII values, you can use those to display certain characters:

Example

```
char myVar1 = 65, myVar2 = 66, myVar3 = 67;

System.out.println(myVar1);

System.out.println(myVar2);

System.out.println(myVar3);
```

Tip: A list of all ASCII values can be found in our ASCII Table Reference.

Strings

The `String` data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
String greeting = "Hello World";

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

The String type is so much used and integrated in Java, that some call it "the special **ninth** type".

A String in Java is actually a **non-primitive** data type, because it refers to an object. The String object has methods that are used to perform certain operations on strings. **Don't worry if you don't understand the term "object" just yet**. We will learn more about strings and objects in a later chapter.

Java Non-Primitive Data Types

Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for `String`).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be `null`.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.

Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc. You will learn more about these in a later chapter.

Java Type Casting

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  `byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`
- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  `double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example

```
public class Main {

  public static void main(String[] args) {

    int myInt = 9;

    double myDouble = myInt; // Automatic casting: int to double


    System.out.println(myInt);      // Outputs 9

    System.out.println(myDouble);   // Outputs 9.0

  }

}
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example

```
public class Main {
```

```
  public static void main(String[] args) {

    double myDouble = 9.78d;

    int myInt = (int) myDouble; // Manual casting: double to int


    System.out.println(myDouble);   // Outputs 9.78

    System.out.println(myInt);      // Outputs 9

  }
}
```

Java Operators

Java Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + **operator** to add together two values:

Example

```
int x = 100 + 50;
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1 = 100 + 50;         // 150 (100 + 50)

int sum2 = sum1 + 250;       // 400 (150 + 250)

int sum3 = sum2 + sum2;      // 800 (400 + 400)
```

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y

*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x = 10;
```

The **addition assignment** operator (+=) adds a value to a variable:

Example

```
int x = 10;

x += 5;
```

A list of all assignment operators:

Operator	**Example**	**Same As**
=	x = 5	x = 5
+=	x += 3	x = x + 3

-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Java Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either `true` or `false`. These values are known as *Boolean values*, and you will learn more about them in the Booleans and If..Else chapter.

In the following example, we use the **greater than** operator (`>`) to find out if 5 is greater than 3:

Example

```
int x = 5;

int y = 3;

System.out.println(x > y); // returns true, because 5 is higher than 3
```

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Java Logical Operators

You can also test for `true` or `false` values with logical operators.

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Java Strings

Java Strings

Strings are used for storing text.

A `String` variable contains a collection of characters surrounded by double quotes:

Example

Create a variable of type `String` and assign it a value:

```
String greeting = "Hello";
```

String Length

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the `length()` method:

Example

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

System.out.println("The length of the txt string is: " + txt.length());
```

More String Methods

There are many string methods available, for example `toUpperCase()` and `toLowerCase()`:

Example

```
String txt = "Hello World";

System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"

System.out.println(txt.toLowerCase());   // Outputs "hello world"
```

Finding a Character in a String

The `indexOf()` method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

Example

```
String txt = "Please locate where 'locate' occurs!";
```

```
System.out.println(txt.indexOf("locate")); // Outputs 7
```

Java counts positions from zero.
0 is the first position in a string, 1 is the second, 2 is the third ...

Java String Concatenation

String Concatenation

The + operator can be used between strings to combine them. This is called **concatenation**:

Example

```
String firstName = "John";

String lastName = "Doe";

System.out.println(firstName + " " + lastName);
```

Note that we have added an empty text (" ") to create a space between firstName and lastName on print.

You can also use the `concat()` method to concatenate two strings:

Example

```
String firstName = "John ";

String lastName = "Doe";

System.out.println(firstName.concat(lastName));
```

Java Numbers and Strings

Adding Numbers and Strings

WARNING!

Java uses the + operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

Example

```
int x = 10;

int y = 20;

int z = x + y;  // z will be 30 (an integer/number)
```

If you add two strings, the result will be a string concatenation:

Example

```
String x = "10";

String y = "20";

String z = x + y;  // z will be 1020 (a String)
```

If you add a number and a string, the result will be a string concatenation:

Example

```
String x = "10";

int y = 20;

String z = x + y;  // z will be 1020 (a String)
```

Java Special Characters

Strings - Special Characters

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

```
String txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

The sequence \" inserts a double quote in a string:

Example

```
String txt = "We are the so-called \"Vikings\" from the north.";
```

The sequence \' inserts a single quote in a string:

Example

```
String txt = "It\'s alright.";
```

The sequence \\ inserts a single backslash in a string:

Example

```
String txt = "The character \\ is called backslash.";
```

Other common escape sequences that are valid in Java are:

Code	**Result**
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed

Java Math

The Java Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.max(*x*,*y*)

The `Math.max(x,y)` method can be used to find the highest value of *x* and *y*:

Example

```
Math.max(5, 10);
```

Math.min(*x*,*y*)

The `Math.min(x,y)` method can be used to find the lowest value of *x* and *y*:

Example

```
Math.min(5, 10);
```

Math.sqrt(*x*)

The `Math.sqrt(x)` method returns the square root of *x*:

Example

```
Math.sqrt(64);
```

Math.abs(*x*)

The `Math.abs(x)` method returns the absolute (positive) value of *x*:

Example

```
Math.abs(-4.7);
```

Random Numbers

`Math.random()` returns a random number between 0.0 (inclusive), and 1.0 (exclusive):

Example

```
Math.random();
```

To get more control over the random number, for example, if you only want a random number between 0 and 100, you can use the following formula:

Example

```
int randomNum = (int)(Math.random() * 101);  // 0 to 100
```

Java Booleans

Java Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, Java has a `boolean` data type, which can store `true` or `false` values.

Boolean Values

A boolean type is declared with the `boolean` keyword and can only take the values `true` or `false`:

Example

```
boolean isJavaFun = true;

boolean isFishTasty = false;

System.out.println(isJavaFun);     // Outputs true

System.out.println(isFishTasty);   // Outputs false
```

However, it is more common to return boolean values from boolean expressions, for conditional testing (see below).

Boolean Expression

A Boolean expression returns a boolean value: `true` or `false`.

This is useful to build logic, and find answers.

For example, you can use a comparison operator, such as the **greater than** (`>`) operator, to find out if an expression (or a variable) is true or false:

Example

```
int x = 10;
```

```
int y = 9;

System.out.println(x > y); // returns true, because 10 is higher than 9
```

Or even easier:

Example

```
System.out.println(10 > 9); // returns true, because 10 is higher than 9
```

In the examples below, we use the **equal to** (==) operator to evaluate an expression:

Example

```
int x = 10;

System.out.println(x == 10); // returns true, because the value of x is equal to
10
```

Example

```
System.out.println(10 == 15); // returns false, because 10 is not equal to 15
```

Real Life Example

Let's think of a "real life example" where we need to find out if a person is old enough to vote.

In the example below, we use the >= comparison operator to find out if the age (25) is **greater than** OR **equal to** the voting age limit, which is set to 18:

Example

```
int myAge = 25;

int votingAge = 18;

System.out.println(myAge >= votingAge);
```

Cool, right? An even better approach (since we are on a roll now), would be to wrap the code above in an if...else statement, so we can perform different actions depending on the result:

Example

Output "Old enough to vote!" if `myAge` is **greater than or equal to** `18`. Otherwise output "Not old enough to vote.":

```
int myAge = 25;

int votingAge = 18;



if (myAge >= votingAge) {

  System.out.println("Old enough to vote!");

} else {

  System.out.println("Not old enough to vote.");

}
```

Booleans are the basis for all Java comparisons and conditions.

You will learn more about conditions (`if...else`) in the next chapter.

Java If ... Else

Java Conditions and If Statements

You already know that Java supports the usual logical conditions from mathematics:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

The if Statement

Use the `if` statement to specify a block of Java code to be executed if a condition is `true`.

Syntax

```
if (condition) {

  // block of code to be executed if the condition is true

}
```

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is `true`, print some text:

Example

```
if (20 > 18) {

  System.out.println("20 is greater than 18");

}
```

We can also test variables:

Example

```
int x = 20;

int y = 18;

if (x > y) {

  System.out.println("x is greater than y");

}
```

Example explained

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the > operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is `false`.

Syntax

```
if (condition) {

  // block of code to be executed if the condition is true

} else {

  // block of code to be executed if the condition is false

}
```

Example

```
int time = 20;

if (time < 18) {

  System.out.println("Good day.");

} else {

  System.out.println("Good evening.");
```

```
}

// Outputs "Good evening."
```

Example explained

In the example above, time (20) is greater than 18, so the condition is `false`. Because of this, we move on to the `else` condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

The else if Statement

Use the `else if` statement to specify a new condition if the first condition is `false`.

Syntax

```
if (condition1) {

  // block of code to be executed if condition1 is true

} else if (condition2) {

  // block of code to be executed if the condition1 is false and condition2 is
true

} else {

  // block of code to be executed if the condition1 is false and condition2 is
false

}
```

Example

```
int time = 22;

if (time < 10) {

  System.out.println("Good morning.");

} else if (time < 18) {

  System.out.println("Good day.");

} else {

  System.out.println("Good evening.");

}
```

```
// Outputs "Good evening."
```

Example explained

In the example above, time (22) is greater than 10, so the **first condition** is `false`. The next condition, in the `else if` statement, is also `false`, so we move on to the `else` condition since **condition1** and **condition2** is both `false` - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

Java Short Hand If...Else (Ternary Operator)

Short Hand If...Else

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands.

It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements:

Syntax

```
variable = (condition) ? expressionTrue :  expressionFalse;
```

Instead of writing:

Example

```
int time = 20;

if (time < 18) {

  System.out.println("Good day.");

} else {

  System.out.println("Good evening.");

}
```

You can simply write:

Example

```
int time = 20;

String result = (time < 18) ? "Good day." : "Good evening.";

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

Java Switch

Java Switch Statements

Instead of writing **many** `if..else` statements, you can use the `switch` statement.

The `switch` statement selects one of many code blocks to be executed:

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The `switch` expression is evaluated once.
- The value of the expression is compared with the values of each `case`.
- If there is a match, the associated block of code is executed.
- The `break` and `default` keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;

switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
```

```
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

The break Keyword

When Java reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

The `default` keyword specifies some code to run if there is no case match:

Example

```
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

Note that if the `default` statement is used as the last statement in a switch block, it does not need a break.

Java While Loop

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

Java While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

Syntax

```
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {

  // code block to be executed

}

while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
int i = 0;
do {

  System.out.println(i);

  i++;

}

while (i < 5);
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

Java For Loop

Java For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

Syntax

```
for (statement 1; statement 2; statement 3) {

  // code block to be executed

}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++) {

  System.out.println(i);

}
```

Example explained

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Another Example

This example will only print even values between 0 and 10:

Example

```
for (int i = 0; i <= 10; i = i + 2) {
  System.out.println(i);
}
```

Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

```
// Outer loop
for (int i = 1; i <= 2; i++) {
  System.out.println("Outer: " + i); // Executes 2 times

  // Inner loop
  for (int j = 1; j <= 3; j++) {
    System.out.println(" Inner: " + j); // Executes 6 times (2 * 3)
  }
}
```

Java For Each Loop

For-Each Loop

There is also a "**for-each**" loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

Note: Don't worry if you don't understand the example above. You will learn more about Arrays in the Java Arrays chapter.

Java Break and Continue

Java Break

You have already seen the `break` statement used in an earlier chapter. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example stops the loop when i is equal to 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  System.out.println(i);
}
```

Java Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  System.out.println(i);
}
```

Break and Continue in While Loop

You can also use `break` and `continue` in while loops:

Break Example

```
int i = 0;
while (i < 10) {
  System.out.println(i);
  i++;
  if (i == 4) {
    break;
  }
}
```

Continue Example

```
int i = 0;
while (i < 10) {
  if (i == 4) {
    i++;
    continue;
  }
  System.out.println(i);
  i++;
}
```

Java Arrays

Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

```
String[] cars;
```

We have now declared a variable that holds an array of strings. To insert values to it, you can place the values in a comma-separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You can access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[0]);

// Outputs Volvo
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

cars[0] = "Opel";

System.out.println(cars[0]);

// Now outputs Opel instead of Volvo
```

Array Length

To find out how many elements an array has, use the `length` property:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars.length);

// Outputs 4
```

Java Arrays Loop

Loop Through an Array

You can loop through the array elements with the `for` loop, and use the `length` property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.length; i++) {

  System.out.println(cars[i]);

}
```

Loop Through an Array with For-Each

There is also a "**for-each**" loop, which is used exclusively to loop through elements in arrays:

Syntax

```
for (type variable : arrayname) {

  ...

}
```

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (String i : cars) {

  System.out.println(i);

}
```

The example above can be read like this: **for each** `String` element (called **i** - as in **i**ndex) in **cars**, print out the value of **i**.

If you compare the `for` loop and **for-each** loop, you will see that the **for-each** method is easier to write, it does not require a counter (using the length property), and it is more readable.

Java Multi-Dimensional Arrays

Multidimensional Arrays

A multidimensional array is an array of arrays.

Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns.

To create a two-dimensional array, add each array within its own set of **curly braces**:

Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

myNumbers is now an array with two arrays as its elements.

Access Elements

To access the elements of the **myNumbers** array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

System.out.println(myNumbers[1][2]); // Outputs 7
```

Remember that: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change Element Values

You can also change the value of an element:

Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

myNumbers[1][2] = 9;
```

```
System.out.println(myNumbers[1][2]); // Outputs 9 instead of 7
```

Loop Through a Multi-Dimensional Array

We can also use a `for loop` inside another `for loop` to get the elements of a two-dimensional array (we still have to point to the two indexes):

Example

```
public class Main {

  public static void main(String[] args) {

    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

    for (int i = 0; i < myNumbers.length; ++i) {

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

        System.out.println(myNumbers[i][j]);

      }
    }

  }

}
```