**JAVA**

**JAVA:**

Java is a high -level, class based object oriented popular programming language, created in 1995.

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

First appeared in May 23, 1995, 28 years ago. As of September 2023, JAVA 21  is the latest version, which is also a Long Term Support (LTS) version. Java 8, 11, and 17 are previous LTS versions still officially supported.

**JAVA** is used for:

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

## Why to 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++ & C# it makes it easy for programmers to switch to Java or vice versa.

**JAVA Installation**

Few Systems already comes with JAVA to check it go to Command prompt and search for JAVA version.

Java -version.

## Setup for Windows

To install Java on Windows:

1. Go to "System Properties" (Can be found on Control Panel > System and Security > System > Advanced System Settings)
2. Click on the "Environment variables" button under the "Advanced" tab
3. Then, select the "Path" variable in System variables and click on the "Edit" button
4. Click on the "New" button and add the path where Java is installed, followed by **\bin**. By default, Java is installed in C:\Program Files\Java\jdk-11.0.1 (If nothing else was specified when you installed it). In that case, You will have to add a new path with: **C:\Program Files\Java\jdk-11.0.1\bin**  
   Then, click "OK", and save the settings
5. At last, open Command Prompt (cmd.exe) and type **java -version** to see if Java is running on your machine

## Java Quick start

In Java, every application begins with a class name, and that class must match the filename.

Let's create our first Java file, called Main.java, which can be done in any text editor (like Notepad).

The file should contain a "Hello World" message, which is written with the following code:

**Main.java**

public class Main {

public static void main(String[] args) {

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

}

}

Save the code in Notepad as "Main.java". Open Command Prompt (cmd.exe), navigate to the directory where you saved your file, and type "javac Main.java":

C:\Users\Your Name>javac Main.java

This will compile your code. If there are no errors in the code, the command prompt will take you to the next line. Now, type "java Main" to run the file:

C:\Users\Your Name>java Main

**The output should read:**

Hello World

 A class should always start with an uppercase first letter.

**Note:** Java is case-sensitive: "MyClass" and "myclass" has different meaning.

The name of the java file **must match** the class name. When saving the file, save it using the class name and add ".java" to the end of the filename. To run the example above on your computer, make sure that Java is properly installed.

## 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.

Every Java program has a class name which must match the filename, and that every program must contain the main() method.

## System.out.println()

## Public static void main(String[] args) {

## System.out.println(“Hello World”);

## }

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

**Note:** The curly braces {} marks the beginning and the end of a block of code.

System is a built-in Java class that contains useful members, such as out, which is short for "output". The println() method, short for "print line", is used to print a value to the screen (or a file).

Each code statement must end with a semicolon (;).

# **Java Output / Print**

**Print Text**

The println() method to output values or print text in Java:

System.out.prinln(“Hello World!”)

Println is used to add NEW line.

**Example**

public class Main {

public static void main(String[] args) {

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

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

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

}

}

**Output**

Hello World!

I am learning Java.

It is awesome!

## Double Quotes

In text, it must be wrapped inside double quotations marks "".

The result will be error if we forgot double quotes.

**Example**

public class Main {

public static void main(String[] args) {

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

}

}

**Output**

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**

public class Main {

public static void main(String[] args) {

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

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

}

}

**Output**

Hello World! I will print on the same line.

## Print Numbers

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

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

**Example**

public class Main {

public static void main(String[] args) {

System.out.println(3);

System.out.println(358);

System.out.println(50000);

}

}

**Output**

3

358

50000

mathematical calculations inside the println() method:

**Example**

public class Main {

public static void main(String[] args) {

System.out.println(3 + 3);

}

}

**Output**

6

**Example**

public class Main {

public static void main(String[] args) {

System.out.println(17 \* 3);

}

}

**Output**

51

## 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:

public class Main {

public static void main(String[] args) {

// This is a comment

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

}

}

**OUTPUT**

Hello World

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).

## Java Multi-line Comments

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

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

**Example**

public class Main {

public static void main(String[] args) {

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

to the screen, and it is amazing \*/

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

}

}

**OUTPUT**

Hello World.

## 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:

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:

Create a variable called name of type String and assign it the value “Gouthami”.

**Example**

public class Main {

public static void main(String[] args) {

String name = "Gouthami";

System.out.println(name);

}

}

**Output**

Gouthami

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

Create a variable called myNum of type int and assign it the value 17

**Example**

public class Main {

public static void main(String[] args) {

int myNum = 17;

System.out.println(myNum);

}

}

**Output**

17

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

**Example**

public class Main {

public static void main(String[] args) {

int myNum;

myNum = 17;

System.out.println(myNum);

}

}

**Output**

17

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

**Example**

public class Main {

public static void main(String[] args) {

int myNum = 15;

myNum = 20; // myNum is now 20

System.out.println(myNum);

}

}

**Output**

20

## 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**

public class Main {

public static void main(String[] args) {

final int myNum = 15;

myNum = 20; // will generate an error

System.out.println(myNum);

}

}

**Output**

Error

## Other Types

A demonstration of how to declare variables of other types:

int myNum = 6;

float myFloatNum = 17.99f;

char myLetter = ‘g’;

boolean myBool = false;

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**

public class Main {

public static void main(String[] args) {

String name = "Gouthami";

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

}

}

**Output**

Hello Gouthami

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

**Example**

public class Main {

public static void main(String[] args) {

String firstName = "Gouthami ";

String lastName = "Koyya";

String fullName = firstName + lastName;

System.out.println(fullName);

}

}

**Output**

Gouthami Koyya

For numeric values, the + character works as a mathematical [operator](https://www.w3schools.com/java/java_operators.asp) (notice that we use int (integer) variables here):

**Example**

public class Main {

public static void main(String[] args) {

int x = 5;

int y = 6;

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

}

}

**Output**

11

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**

## Declare Many Variables

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

**Example**

public class Main {

public static void main(String[] args) {

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

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

}

}

**Output**

61

## One Value to Multiple Variables

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

**Example**

public class Main {

public static void main(String[] args) {

int x, y, z;

x = y = z = 50;

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

}

}

**Output**

150

# **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**

public class Main {

public static void main(String[] args) {

// Good

int minutesPerHour = 60;

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

int m = 60;

System.out.println(minutesPerHour);

System.out.println(m);

}

}

**Output**

60

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

**EXAMPLE**

## public class Main {

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

## int myNum = 5; // integer (whole number)

## float myFloatNum = 5.99f; // floating point number

## char myLetter = 'G'; // character

## boolean myBool = true; // boolean

## String myText = "Hello"; // String

## System.out.println(myNum);

## System.out.println(myFloatNum);

## System.out.println(myLetter);

## System.out.println(myBool);

## System.out.println(myText);

## }

## }

**Output**

## 5

## 5.99

## G

## true

## Hello

**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](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp) 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)

## 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**

public class Main {

public static void main(String[] args) {

byte myNum = 100;

System.out.println(myNum);

}

}

**Output**

100

### **Short**

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

short myNum = 5000;

System.out.println(myNum);

}

}

**Output**

5000

### **Int**

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

int myNum = 100000;

System.out.println(myNum);

}

}

**Output**

100000

### **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. Should end the value with an "L":

**EXAMPLE**

public class Main {

public static void main(String[] args) {

long myNum = 15000000000L;

System.out.println(myNum);

}

}

**Output**

15000000000

## 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:

**EXAMPLE**

public class Main {

public static void main(String[] args) {

float myNum = 5.75f;

System.out.println(myNum);

}

}

**Output**

5.75

**Double Example**

public class Main {

public static void main(String[] args) {

double myNum = 19.99d;

System.out.println(myNum);

}

}

**Output**

19.99

### **Scientific Numbers**

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

float f1 = 35e3f;

double d1 = 12E4d;

System.out.println(f1);

System.out.println(d1);

}

}

**Output**

35000.0

120000.0

## 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**

public class Main {

public static void main(String[] args) {

boolean isJavaFun = true;

boolean isFishTasty = false;

System.out.println(isJavaFun);

System.out.println(isFishTasty);

}

}

**Output**

true

false

# **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**

public class Main {

public static void main(String[] args) {

char myGrade = 'B';

System.out.println(myGrade);

}

}

**Output**

B

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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

System.out.println(myVar1);

System.out.println(myVar2);

System.out.println(myVar3);

}

}

**Output**

A

B

C

## Strings

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

String greeting = "Hello World";

System.out.println(greeting);

}

}

**Output**

Hello World.

## 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](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp), [Classes,](https://www.w3schools.com/java/java_classes.asp)[Interface](https://www.w3schools.com/java/java_interface.asp), etc.

## 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);

System.out.println(myDouble);

}

}

**Output**

9

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; // Explicit casting: double to int

System.out.println(myDouble);

System.out.println(myInt);

}

}

**Output**

9.78

**9**

## 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**

public class Main {

public static void main(String[] args) {

int x = 100 + 50;

System.out.println(x);

}

}

**Output**

150

Java divides the operators into the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Bitwise operators

## Java Strings

## 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**

public class Main {

public static void main(String[] args) {

String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

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

}

}

**Output**

## The length of the txt string is 26

## More String Methods

**EXAMPLE**

public class Main {

public static void main(String[] args) {

String txt = "Hello World";

System.out.println(txt.toUpperCase());

System.out.println(txt.toLowerCase());

}

}

**Output**

HELLO WORLD

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**

public class Main {

public static void main(String[] args) {

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

System.out.println(txt.indexOf("locate"));

}

}

**Output**

7

## String Concatenation

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

**EXAMPLE**

public class Main {

public static void main(String args[]) {

String firstName = "Gouthami";

String lastName = "Koyya";

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

}

}

**Output**

Gouthami Koyya

## Adding Numbers and Strings

## Numbers are added Strings are concated.

**EXAMPLE**

## public class Main {

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

## int x = 10;

## int y = 20;

## int z = x + y;

## System.out.println(z);

## }

## }

**Output**

## 30

**EXAMPLE**

public class Main {

public static void main(String[] args) {

String x = "10";

String y = "20";

String z = x + y;

System.out.println(z);

}

}

**Output**

1020

## Strings - Special Characters

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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

System.out.println(txt);

}

}

**Output**

We are the so-called "Vikings" from the north

# **Java Math**

## Math.max(x,y)

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

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

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

System.out.println(Math.max(5, 10));

}

}

**Output**

10

## Math.min(x,y)

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

System.out.println(Math.min(5, 10));

}

}

**Output**

5

## Math.sqrt(x)

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

System.out.println(Math.sqrt(64));

}

}

**Output**

8

## The if Statement

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

if (20 > 18) {

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

}

}

}

**Output**

20 is greater than 18

## The else Statement

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

int time = 20;

if (time < 18) {

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

} else {

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

}

}

}

**Output**

Good evening

## The else if Statement

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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.");

}

}

}

**Output**

Good evening

## Short Hand If...Else

There is also a short-hand [if else](https://www.w3schools.com/java/java_conditions.asp), 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:

**EXAMPLE**

public class Main {

public static void main(String[] args) {

int time = 20;

String result;

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

System.out.println(result);

}

}

**Output**

Good evening

## 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:

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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;

}

}

}

**Output**

Thursday

## The default Keyword

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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");

}

}

}

**Output**

Looking forward to the Weekend

## Java While Loop

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

int i = 0;

while (i < 5) {

System.out.println(i);

i++;

}

}

}

**Output**

0

1

2

3

4

## 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.

**EXAMPLE**

public class Main {

public static void main(String[] args) {

int i = 0;

do {

System.out.println(i);

i++;

}

while (i < 5);

}

}

**Output**

0

1

2

3

4

## 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:

**EXAMPLE**

public class Main {

public static void main(String[] args) {

for (int i = 0; i <= 10; i = i + 2) {

System.out.println(i);

}

}

}

**Output**

0

2

4

6

8

10

## For-Each Loop

There is also a "for-each" loop, which is used exclusively to loop through elements in an [**array**](https://www.w3schools.com/java/java_arrays.asp):

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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

for (String i : cars) {

System.out.println(i);

}

}

}

**Output**

Volvo

BMW

Ford

Mazda

## Break and Continue in While Loop

You can also use break and continue in while loops:

**EXAMPLE**

public class Main {

public static void main(String[] args) {

int i = 0;

while (i < 10) {

System.out.println(i);

i++;

if (i == 4) {

break;

}

}

}

}

**Output**

0

1

2

3

**EXAMPLE**

public class Main {

public static void main(String[] args) {

int i = 0;

while (i < 10) {

if (i == 4) {

i++;

continue;

}

System.out.println(i);

i++;

}

}

}

**Output**

0

1

2

3

4

5

6

7

8

9

## 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**:

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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

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

}

}

**Output**

Volvo

## 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.

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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

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

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

}

}

}

**Output**

Volvo

BMW

Ford

Mazda

## Loop Through an Array with For-Each

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

**EXAMPLE**

public class Main {

public static void main(String[] args) {

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

for (String i : cars) {

System.out.println(i);

}

}

}

**Output**

Volvo

BMW

Ford

Mazda

## 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**

public class Main {

public static void main(String[] args) {

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

System.out.println(myNumbers[1][2]);

}

}

**Output**

7

## Change Element Values

You can also change the value of an element:

public class Main {

public static void main(String[] args) {

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

myNumbers[1][2] = 9;

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

}

}

**Output**

9

## 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]);

}

}

}

}

**Output**

1

2

3

4

5

6

7

## Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

**EXAMPLE**

public class Main {

static void myMethod() {

System.out.println("I just got executed!");

}

public static void main(String[] args) {

myMethod();

myMethod();

myMethod();

}

}

**Output**

I just got executed!

I just got executed!

I just got executed!

## Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

**EXAMPLE**

public class Main {

static void myMethod(String fname) {

System.out.println(fname + " Koyya");

}

public static void main(String[] args) {

myMethod("Gouthami");

myMethod("Bindu");

}

}

**Output**

Gouthami Koyya

Bindu Koyya

## Multiple Parameters

public class Main {

static void myMethod(String fname, int age) {

System.out.println(fname + " is " + age);

}

public static void main(String[] args) {

myMethod("Gouthami", 5);

myMethod("Bindu", 8);

}

}

**Output**

Gouthami is 5

Bindu is 8

## Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

**EXAMPLE**

public class Main {

static int myMethod(int x) {

return 5 + x;

}

public static void main(String[] args) {

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

}

}

**Output**

**8**

## A Method with If...Else

It is common to use if...else statements inside methods:

**EXAMPLE**

public class Main {

// Create a checkAge() method with an integer parameter called age

static void checkAge(int age) {

// If age is less than 18, print "access denied"

if (age < 18) {

System.out.println("Access denied - You are not old enough!");

// If age is greater than, or equal to, 18, print "access granted"

} else {

System.out.println("Access granted - You are old enough!");

}

}

public static void main(String[] args) {

checkAge(20); // Call the checkAge method and pass along an age of 20

}

}

**Output**

You are old enough!

## Method Overloading

With**method overloading**, multiple methods can have the same name with different parameters:

**EXAMPLE**

public class Main {

static int plusMethodInt(int x, int y) {

return x + y;

}

static double plusMethodDouble(double x, double y) {

return x + y;

}

public static void main(String[] args) {

int myNum1 = plusMethodInt(8, 5);

double myNum2 = plusMethodDouble(4.3, 6.26);

System.out.println("int: " + myNum1);

System.out.println("double: " + myNum2);

}

}

**Output**

Int :13

Double :10.5599999999

## Java Scope

In Java, variables are only accessible inside the region they are created. This is called **scope**.

## Method Scope

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared:

**EXAMPLE**

public class Main {

public static void main(String[] args) {

// Code here cannot use x

int x = 100;

// Code here can use x

System.out.println(x);

}

}

**Output**

100

## Block Scope

A block of code refers to all of the code between curly braces {}.

Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared:

**EXAMPLE**

public class Main {

public static void main(String[] args) {

// Code here CANNOT use x

{ // This is a block

// Code here CANNOT use x

int x = 100;

// Code here CAN use x

System.out.println(x);

} // The block ends here

// Code here CANNOT use x

}

}

**OUTPUT**

100

## Java Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

**Example**

public class Main {

public static void main(String[] args) {

int result = sum(10);

System.out.println(result);

}

public static int sum(int k) {

if (k > 0) {

return k + sum(k - 1);

} else {

return 0;

}

}

}

**Output**

55

## Halting Condition

Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself.

**Example**

public class Main {

public static void main(String[] args) {

int result = sum(5, 10);

System.out.println(result);

}

public static int sum(int start, int end) {

if (end > start) {

return end + sum(start, end - 1);

} else {

return end;

}

}

}

**Output**

4

**JAVA OOP**

OOPS stands for object-oriented programming. Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time

**Tip:** The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

## Java Classes/Objects

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes,** such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

## Create a Class

To create a class, use the keyword class:

## Create an Object

In Java, an object is created from a class.

To create an object of Main, specify the class name, followed by the object name, and use the keyword new:

**Example**

public class Main {

int x = 5;

public static void main(String[] args) {

Main myObj = new Main();

System.out.println(myObj.x);

}

}

**Output**

5

## Multiple Objects

You can create multiple objects of one class:

**Example**

public class Main {

int x = 5;

public static void main(String[] args) {

Main myObj1 = new Main();

Main myObj2 = new Main();

System.out.println(myObj1.x);

System.out.println(myObj2.x);

}

}

**Output**

5

5

## Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

* Main.java
* Second.java

public class Main {

int x = 5;

}

class Second {

public static void main(String[] args) {

Main myObj = new Main();

System.out.println(myObj.x);

}

}

**Output**

5

## Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax (.):

The following example will create an object of the Main class, with the name myObj. We use the x attribute on the object to print its value:

**Example**

public class Main {

int x = 5;

public static void main(String[] args) {

Main myObj = new Main();

System.out.println(myObj.x);

}

}

**Output**

5

## Modify Attributes

You can also modify attribute values:

**Example**

Set the value of x = 40

public class Main {

int x;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 40;

System.out.println(myObj.x);

}

}

**Output**

40

**Override existing values:**

Change the value of x to 20

public class Main {

int x = 17;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 20; // x is now 20

System.out.println(myObj.x);

}

}

**Output**

20

If you don't want the ability to override existing values, declare the attribute as **final:**

**Example**

public class Main {

final int x = 10;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 25; // will generate an error

System.out.println(myObj.x);

}

}

**Output**

Error

## Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

Change the value of x to 25 in myObj2, & leave x in myObj1 unchanged:

**Example**

public class Main {

int x = 5;

public static void main(String[] args) {

Main myObj1 = new Main();

Main myObj2 = new Main();

myObj2.x = 25;

System.out.println(myObj1.x);

System.out.println(myObj2.x);

}

}

**Output**

5

25

## Multiple Attributes

You can specify as many attributes as you want:

**Example**

public class Main {

String fname = "Gouthami";

String lname = "Koyya";

int age = 28;

public static void main(String[] args) {

Main myObj = new Main();

System.out.println("Name: " + myObj.fname + " " + myObj.lname);

System.out.println("Age: " + myObj.age);

}

}

**Output**

Name: Gouthami Koyya

Age: 28

## Java Class Methods

## Example

## public class Main {

## static void myMethod() {

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

## }

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

## myMethod();

## }

## }

**Output**

## Hello World!

## Static vs. Public

## Example

## public class Main {

## // Static method

## static void myStaticMethod() {

## System.out.println("Static methods can be called without creating objects");

## }

## // Public method

## public void myPublicMethod() {

## System.out.println("Public methods must be called by creating objects");

## }

## // Main method

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

## myStaticMethod(); // Call the static method

## Main myObj = new Main(); // Create an object of MyClass

## myObj.myPublicMethod(); // Call the public method

## }

## }

**Output**

Static methods can be called without creating objects

Public methods must be called by creating objects

## Access Methods With an Object

## Example

// Create a Main class

public class Main {

// Create a fullThrottle() method

public void fullThrottle() {

System.out.println("The car is going as fast as it can!");

}

// Create a speed() method and add a parameter

public void speed(int maxSpeed) {

System.out.println("Max speed is: " + maxSpeed);

}

// Inside main, call the methods on the myCar object

public static void main(String[] args) {

Main myCar = new Main(); // Create a myCar object

myCar.fullThrottle(); // Call the fullThrottle() method

myCar.speed(200); // Call the speed() method

}

}

**Output**

The car is going as fast as it can!

Max speed is: 200

## Using Multiple Classes

* Main.java
* Second.java

public class Main {

public void fullThrottle() {

System.out.println("The car is going as fast as it can!");

}

public void speed(int maxSpeed) {

System.out.println("Max speed is: " + maxSpeed);

}

}

class Second {

public static void main(String[] args) {

Main myCar = new Main(); // Create a myCar object

myCar.fullThrottle(); // Call the fullThrottle() method

myCar.speed(200); // Call the speed() method

}

}

**Output**

The car is going as fast as it can!

Max speed is: 200

## Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

## Example

// Create a Main class

public class Main {

int x;

// Create a class constructor for the Main class

public Main() {

x = 5;

}

public static void main(String[] args) {

Main myObj = new Main();

System.out.println(myObj.x);

}

}

**Output**

5

## Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

## Example

public class Main {

int x;

public Main(int y) {

x = y;

}

public static void main(String[] args) {

Main myObj = new Main(5);

System.out.println(myObj.x);

}

}

**Output**

5

You can have as many **parameters** as you want

## Example

//filename: Main.java

public class Main {

int modelYear;

String modelName;

public Main(int year, String name) {

modelYear = year;

modelName = name;

}

public static void main(String[] args) {

Main myCar = new Main(1969, "Mustang");

System.out.println(myCar.modelYear + " " + myCar.modelName);

}

}

**Output**

1969 Mustang

# **Java Modifiers**

We are familiar with the public keyword that appears in almost all of our examples:

Public class Main

The public keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

* **Access Modifiers** - controls the access level
* **Non-Access Modifiers** - do not control access level, but provides other functionality

## Final

If you don't want the ability to override existing attribute values, declare attributes as final:

## Example

public class Main {

final int x = 10;

final double PI = 3.14;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 50; // will generate an error

myObj.PI = 25; // will generate an error

System.out.println(myObj.x);

}

}

**Output**

Error

## Static

A static method means that it can be accessed without creating an object of the class, unlike public:

## Abstract

An abstract method belongs to an abstract class, and it does not have a body. The body is provided by the subclass:

## Example

// abstract class

abstract class Main {

public String fname = "Gouthami";

public int age = 28;

public abstract void study(); // abstract method

}

// Subclass (inherit from Main)

class Student extends Main {

public int graduationYear = 2018;

public void study() { // the body of the abstract method is provided here

System.out.println("Studying all day long");

}

}

class Second {

public static void main(String[] args) {

// create an object of the Student class (which inherits attributes and methods from Main)

Student myObj = new Student();

System.out.println("Name: " + myObj.fname);

System.out.println("Age: " + myObj.age);

System.out.println("Graduation Year: " + myObj.graduationYear);

myObj.study(); // call abstract method

}

}

Name: Gouthami

Age: 28

Graduation Year: 2018

Studying all day long

## Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare class variables/attributes as private
* provide public **get** and **set** methods to access and update the value of a private variable

## Get and Set

Private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods.

The get method returns the variable value, and the set method sets the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

MyClass.java

Person.java

Public class Main {

Public static void main(String[] args) {

Person myObj = new Person();

myObj.name = “Gouthami”

System.out.println(myObj.name);

}

}

Public class Persin {

Private String name;

//Getter

Public String getName() {

return name;

}

// Setter

Public void setName(String new Name) {

This.name = newname;

}

}

**Output**

Error

## Java Packages & API

A package in Java is used to group related classes. Think of it as **a folder in a file directory.** We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

* Built-in Packages (packages from the Java API)
* User-defined Packages (create your own packages)

## Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

## Import a Class

If you find a class you want to use, for example, the Scanner class, **which is used to get user input**, write the following code:

import java.util.Scanner; // import the Scanner class

class Main {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in);

String userName;

// Enter username and press Enter

System.out.println("Enter username");

userName = myObj.nextLine();

System.out.println("Username is: " + userName);

}

}

**Output**

Enter UserName

Gouthami

UserName is Gouthami

## Import a Package

There are many packages to choose from. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (\*). The following example will import ALL the classes in the java.util package:

## Example

import java.util.\*; // import the java.util package

class Main {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in);

String userName;

// Enter username and press Enter

System.out.println("Enter username");

userName = myObj.nextLine();

System.out.println("Username is: " + userName);

}

}

**Output**

Enter username

Gouthami

## User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

└── root

└── mypack

└── MyPackageClass.java

To create a package, use the package keyword:

## Example

My PackageClass.java

package mypack;

class MyPackageClass {

public static void main(String[] args) {

System.out.println("This is my package!");

}

}

**Output**

This is my package!

## Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

* **subclass** (child) - the class that inherits from another class
* **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

## Example

class Vehicle {

protected String brand = "Ford";

public void honk() {

System.out.println("");

}

}

class Car extends Vehicle {

private String modelName = "Mustang" Tuut, tuut!;

public static void main(String[] args) {

Car myFastCar = new Car();

myFastCar.honk();

System.out.println(myFastCar.brand + " " + myFastCar.modelName);

}

}

**Output**

Tuut, tuut!

Ford Mustang

## The final Keyword

If you don't want other classes to inherit from a class, use the final keyword:

## Example

final class Vehicle {

protected String brand = "Ford";

public void honk() {

System.out.println("Tuut, tuut!");

}

}

class Main extends Vehicle {

private String modelName = "Mustang";

public static void main(String[] args) {

Main myFastCar = new Main();

myFastCar.honk();

System.out.println(myFastCar.brand + " " + myFastCar.modelName);

}

}

**Output**

Error

## Java Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes

[**Inheritance**](https://www.w3schools.com/java/java_inheritance.asp) lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

## Example

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}

}

class Pig extends Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

class Dog extends Animal {

public void animalSound() {

System.out.println("The dog says: bow wow");

}

}

class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal();

Animal myPig = new Pig();

Animal myDog = new Dog();

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

**Output**

The animal makes a sound

The pig says: wee wee

The dog says: bow bow

## Java Inner Classes

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

## Example

class OuterClass {

int x = 10;

class InnerClass {

int y = 5;

}

}

public class Main {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

}

**Output**

15

## Private Inner Class

Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private

## Example

class OuterClass {

int x = 10;

private class InnerClass {

int y = 5;

}

}

public class Main {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

}

**Output**

Error

## Static Inner Class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

## Example

class OuterClass {

int x = 10;

static class InnerClass {

int y = 5;

}

}

public class Main {

public static void main(String[] args) {

OuterClass.InnerClass myInner = new OuterClass.InnerClass();

System.out.println(myInner.y);

}

}

**Output**

5

## Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

## Example

class OuterClass {

int x = 10;

class InnerClass {

public int myInnerMethod() {

return x;

}

}

}

public class Main {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.myInnerMethod());

}

}

**Output**

10

## Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or [**interfaces**](https://www.w3schools.com/java/java_interface.asp)

The abstract keyword is a non-access modifier, used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

## Example

// Abstract class

abstract class Animal {

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep() {

System.out.println("Zzz");

}

}

// Subclass (inherit from Animal)

class Pig extends Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

**Output**

The pig says: wee wee

Zzz

## Interfaces

Another way to achieve [abstraction](https://www.w3schools.com/java/java_abstract.asp) in Java, is with interfaces.

An interface is a completely "**abstract class**" that is used to group related methods with empty bodies:

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class:

## Example

interface Animal {

public void animalSound(); // interface method (does not have a body)

public void sleep(); // interface method (does not have a body)

}

class Pig implements Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

public void sleep() {

System.out.println("Zzz");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig();

myPig.animalSound();

myPig.sleep();

}

}

**Output**

The pig says: wee wee

Zzz

## Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

## Example

interface FirstInterface {

public void myMethod(); // interface method

}

interface SecondInterface {

public void myOtherMethod(); // interface method

}

// DemoClass "implements" FirstInterface and SecondInterface

class DemoClass implements FirstInterface, SecondInterface {

public void myMethod() {

System.out.println("Some text..");

}

public void myOtherMethod() {

System.out.println("Some other text...");

}

}

class Main {

public static void main(String[] args) {

DemoClass myObj = new DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

**Output**

Some text…

Some other text…

## Enums

An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

## Example

enum Level {

LOW,

MEDIUM,

HIGH

}

public class Main {

public static void main(String[] args) {

Level myVar = Level.MEDIUM;

System.out.println(myVar);

}

}

**Output**

MEDIUM

## Enum inside a Class

You can also have an enum inside a class:

## Example

public class Main {

enum Level {

LOW,

MEDIUM,

HIGH

}

public static void main(String[] args) {

Level myVar = Level.MEDIUM;

System.out.println(myVar);

}

}

**Output**

MEDIUM

## Enum in a Switch Statement

Enums are often used in switch statements to check for corresponding values:

## Example

enum Level {

LOW,

MEDIUM,

HIGH

}

public class Main {

public static void main(String[] args) {

Level myVar = Level.MEDIUM;

switch(myVar) {

case LOW:

System.out.println("Low level");

break;

case MEDIUM:

System.out.println("Medium level");

break;

case HIGH:

System.out.println("High level");

break;

}

}

}

**Output**

MEDIUM

## Loop Through an Enum

The enum type has a values() method, which returns an array of all enum constants. This method is useful when you want to loop through the constants of an enum:

## Example

enum Level {

LOW,

MEDIUM,

HIGH

}

public class Main {

public static void main(String[] args) {

for (Level myVar : Level.values()) {

System.out.println(myVar);

}

}

}

**Output**

LOW

MEDIUM

HIGH

## Java User Input

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read Strings:

import java.util.Scanner; // import the Scanner class

class Main {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in);

String userName;

// Enter username and press Enter

System.out.println("Enter username");

userName = myObj.nextLine();

System.out.println("Username is: " + userName);

}

}

**Output**

Enter username

Gouthami

## Java Dates

Java does not have a built-in Date class, but we can import the java.time package to work with the date and time API. The package includes many date and time classes.

## Display Current Date

To display the current date, import the java.time.LocalDate class, and use its now() method:

import java.time.LocalDate; // import the LocalDate class

public class Main {

public static void main(String[] args) {

LocalDate myObj = LocalDate.now(); // Create a date object

System.out.println(myObj); // Display the current date

}

}

**Output**

23-10-2023

## Display Current Date and Time

To display the current date and time, import the java.time.LocalDateTime class, and use its now() method:

import java.time.LocalDateTime; // import the LocalDateTime class

public class Main {

public static void main(String[] args) {

LocalDateTime myObj = LocalDateTime.now();

System.out.println(myObj);

}

}

## Formatting Date and Time

The "T" in the example above is used to separate the date from the time. You can use the DateTimeFormatter class with the ofPattern() method in the same package to format or parse date-time objects. The following example will remove both the "T" and nanoseconds from the date-time:

import java.time.LocalDateTime; // Import the LocalDateTime class

import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter class

public class Main {

public static void main(String[] args) {

LocalDateTime myDateObj = LocalDateTime.now();

System.out.println("Before formatting: " + myDateObj);

DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDate = myDateObj.format(myFormatObj);

System.out.println("After formatting: " + formattedDate);

}

}

## Java ArrayList

The ArrayList class is a resizable [array](https://www.w3schools.com/java/java_arrays.asp), which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want. The syntax is also slightly different:

## Add Items

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

}

}

**Output**

[Volvo,BMW,Ford,Mazda]

## Access an Item

To access an element in the ArrayList, use the get() method and refer to the index number:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars.get(0));

}

}

**Output**

Volvo

## Change an Item

To modify an element, use the set() method and refer to the index number:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

cars.set(0, "Opel");

System.out.println(cars);

}

}

[Opel, BMW,Ford,Mazda]

## Remove an Item

To remove an element, use the remove() method and refer to the index number:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

cars.remove(0);

System.out.println(cars);

}

}

**Output**

[BMW,Ford,Mazda]

## ArrayList Size

To find out how many elements an ArrayList have, use the size method:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars.size());

}

}

**Output**

4

## Loop Through an ArrayList

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

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

System.out.println(cars.get(i));

}

}

}

**Output**

Volvo

BMW

Ford

Mazda

## Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent [wrapper class](https://www.w3schools.com/java/java_wrapper_classes.asp): Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> myNumbers = new ArrayList<Integer>();

myNumbers.add(10);

myNumbers.add(15);

myNumbers.add(20);

myNumbers.add(25);

for (int i : myNumbers) {

System.out.println(i);

}

}

}

**Output**

10

15

20

25

## Sort an ArrayList

Another useful class in the java.util package is the Collections class, which include the sort() method for sorting lists alphabetically or numerically:

import java.util.ArrayList;

import java.util.Collections;

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

Collections.sort(cars);

for (String i : cars) {

System.out.println(i);

}

}

}

**Output**

BMW

Ford

Mazda

Volvo

## Java LinkedList

## import java.util.LinkedList;

## public class Main {

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

## LinkedList<String> cars = new LinkedList<String>();

## cars.add("Volvo");

## cars.add("BMW");

## cars.add("Ford");

## cars.add("Mazda");

## System.out.println(cars);

## }

## }

**Output**

Volvo

BMW

Ford

## Mazda

## Java HashMap

A HashMap however, store items in "**key**/**value**" pairs, and you can access them by an index of another type (e.g. a String).

One object is used as a key (index) to another object (value). It can store different types: String keys and Integer values, or the same type, like: String keys and String values:

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

System.out.println(capitalCities);

}

}

**Output**

**{**USA=Washington DC ,Norway= Oslo, England =London ,Germany=Berlin}

## Access an Item

To access a value in the HashMap, use the get() method and refer to its key:

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

System.out.println(capitalCities.get("England"));

}

}

**Output**

London

## Remove an Item

To remove an item, use the remove() method and refer to the key:

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

capitalCities.remove("England");

System.out.println(capitalCities);

}

}

**Output**

**{**USA=Washington DC, Norway= Oslo , Germany=Berlin}

**To remove all items, use the clear() method:**

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

capitalCities.clear();

System.out.println(capitalCities);

}

}

**Output**

{}

## HashMap Size

To find out how many items there are, use the size() method:

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

System.out.println(capitalCities.size());

}

}

**Output**

4

## Loop Through a HashMap

Loop through the items of a HashMap with a **for-each** loop.

**Note:** Use the keySet() method if you only want the keys, and use the values() method if you only want the values:

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

for (String i : capitalCities.keySet()) {

System.out.println(i);

}

}

}

USA

Norway

Germany

England

## Other Types

Keys and values in a HashMap are actually objects. In the examples above, we used objects of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent [wrapper class](https://www.w3schools.com/java/java_wrapper_classes.asp): Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

// Import the HashMap class

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

// Create a HashMap object called people

HashMap<String, Integer> people = new HashMap<String, Integer>();

// Add keys and values (Name, Age)

people.put("John", 32);

people.put("Steve", 30);

people.put("Angie", 33);

for (String i : people.keySet()) {

System.out.println("Name: " + i + " Age: " + people.get(i));

}

}

}

**Output**

Name : Angie Age : 33

Name : Steve Age : 30

Name : John Age : 33

## Java HashSet

A HashSet is a collection of items where every item is unique, and it is found in the java.util package:

## Add Items

The HashSet class has many useful methods. For example, to add items to it, use the add() method:

// Import the HashSet class

import java.util.HashSet;

public class Main {

public static void main(String[] args) {

HashSet<String> cars = new HashSet<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("BMW");

cars.add("Mazda");

System.out.println(cars);

}

}

**Output**

[Volvo, BMW, Ford, Mazda]

## Check If an Item Exists

To check whether an item exists in a HashSet, use the contains() method:

// Import the HashSet class

import java.util.HashSet;

public class Main {

public static void main(String[] args) {

HashSet<String> cars = new HashSet<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("BMW");

cars.add("Mazda");

System.out.println(cars.contains("Mazda"));

}

}

**Output**

True

## Remove an Item

To remove an item, use the remove() method:

// Import the HashSet class

import java.util.HashSet;

public class Main {

public static void main(String[] args) {

HashSet<String> cars = new HashSet<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("BMW");

cars.add("Mazda");

cars.remove("Volvo");

System.out.println(cars);

}

}

**Output**

[Mazda BMW, Ford]

To remove all items, use the **clear()** method:

// Import the HashSet class

import java.util.HashSet;

public class Main {

public static void main(String[] args) {

HashSet<String> cars = new HashSet<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("BMW");

cars.add("Mazda");

cars.clear();

System.out.println(cars);

}

}

**Output**

[]

## HashSet Size

To find out how many items there are, use the size method:

// Import the HashSet class

import java.util.HashSet;

public class Main {

public static void main(String[] args) {

HashSet<String> cars = new HashSet<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("BMW");

cars.add("Mazda");

System.out.println(cars.size());

}

}

**Output**

4

## Loop Through a HashSet

Loop through the items of an HashSet with a for-each loop:

// Import the HashSet class

import java.util.HashSet;

public class Main {

public static void main(String[] args) {

HashSet<String> cars = new HashSet<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("BMW");

cars.add("Mazda");

for (String i : cars) {

System.out.println(i);

}

}

}

**Output**

Volvo

Mazda

Ford

BMW

## Other Types

Items in an HashSet are actually objects. In the examples above, we created items (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent [wrapper class](https://www.w3schools.com/java/java_wrapper_classes.asp): Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

// Import the HashSet class

import java.util.HashSet;

public class Main {

public static void main(String[] args) {

// Create a HashSet object called numbers

HashSet<Integer> numbers = new HashSet<Integer>();

// Add values to the set

numbers.add(4);

numbers.add(7);

numbers.add(8);

// Show which numbers between 1 and 10 are in the set

for(int i = 1; i <= 10; i++) {

if(numbers.contains(i)) {

System.out.println(i + " was found in the set.");

} else {

System.out.println(i + " was not found in the set.");

}

}

}

}

**Output**

1 was not found in the set

2 was not found in the set

3 was not found in the set

4 was found in the set

5 was not found in the set

6 was not found in the set

7 was found in the set

8 was found in the set

9 was not found in the set

10 was not found in the set

## Java Iterator

An Iterator is an object that can be used to loop through collections, like [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) and [HashSet](https://www.w3schools.com/java/java_hashset.asp). It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the java.util package.

## Getting an Iterator

The iterator() method can be used to get an Iterator for any collection:

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

// Make a collection

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

// Get the iterator

Iterator<String> it = cars.iterator();

// Print the first item

System.out.println(it.next());

}

}

**Output**

Volvo

## Looping Through a Collection

To loop through a collection, use the hasNext() and next() methods of the Iterator:

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

// Make a collection

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

// Get the iterator

Iterator<String> it = cars.iterator();

// Loop through a collection

while(it.hasNext()) {

System.out.println(it.next());

}

}

}

**Output**

Volvo

BMW

Ford

Mazda

## Removing Items from a Collection

Iterators are designed to easily change the collections that they loop through. The remove() method can remove items from a collection while looping.

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(12);

numbers.add(8);

numbers.add(2);

numbers.add(23);

Iterator<Integer> it = numbers.iterator();

while(it.hasNext()) {

Integer i = it.next();

if(i < 10) {

it.remove();

}

}

System.out.println(numbers);

}

}

**Output**

[12 , 23]

## Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> myNumbers = new ArrayList<Integer>();

myNumbers.add(10);

myNumbers.add(15);

myNumbers.add(20);

myNumbers.add(25);

for (int i : myNumbers) {

System.out.println(i);

}

}

}

**Output**

10

15

20

25

## Creating Wrapper Objects

To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

public class Main {

public static void main(String[] args) {

Integer myInt = 5;

Double myDouble = 5.99;

Character myChar = 'A';

System.out.println(myInt);

System.out.println(myDouble);

System.out.println(myChar);

}

}

**Output**

5

5.99

A

Since you're now working with objects, you can use certain methods to get information about the specific object.

For example, the following methods are used to get the value associated with the corresponding wrapper object: intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

This example will output the same result as the example above:

public class Main {

public static void main(String[] args) {

Integer myInt = 5;

Double myDouble = 5.99;

Character myChar = 'A';

System.out.println(myInt.intValue());

System.out.println(myDouble.doubleValue());

System.out.println(myChar.charValue());

}

}

**Output**

5

5.99

A

Another useful method is the toString() method, which is used to convert wrapper objects to strings.

In the following example, we convert an Integer to a String, and use the length() method of the String class to output the length of the "string":

public class Main {

public static void main(String[] args) {

Integer myInt = 10330;

String myString = myInt.toString();

System.out.println(myString.length());

}

}

**Output**

5

## Java Exceptions

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

## Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

public class Main {

public static void main(String[] args) {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

}

}

**Output**

**Error**

If an error occurs, we can use try...catch to catch the error and execute some code to handle it:

public class Main {

public static void main(String[] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

}

}

}

**Output**

Something went wrong

## Finally

The finally statement lets you execute code, after try...catch, regardless of the result:

public class Main {

public static void main(String[] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

} finally {

System.out.println("The 'try catch' is finished.");

}

}

}

**Output**

Something went wrong.

The 'try catch' is finished.

## The throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

public class Main {

static void checkAge(int age) {

if (age < 18) {

throw new ArithmeticException("Access denied - You must be at least 18 years old.");

} else {

System.out.println("Access granted - You are old enough!");

}

}

public static void main(String[] args) {

checkAge(15);

}

}

**Output**

Exception

public class Main {

static void checkAge(int age) {

if (age < 18) {

throw new ArithmeticException("Access denied - You must be at least 18 years old.");

} else {

System.out.println("Access granted - You are old enough!");

}

}

public static void main(String[] args) {

checkAge(20);

}

}

**Output**

Access granted – You are old enough!

# **Java Regular Expressions**

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Java does not have a built-in Regular Expression class, but we can import the java.util.regex package to work with regular expressions. The package includes the following classes:

* Pattern Class - Defines a pattern (to be used in a search)
* Matcher Class - Used to search for the pattern
* PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Main {

public static void main(String[] args) {

Pattern pattern = Pattern.compile("w3schools", Pattern.CASE\_INSENSITIVE);

Matcher matcher = pattern.matcher("Visit W3Schools!");

boolean matchFound = matcher.find();

if(matchFound) {

System.out.println("Match found");

} else {

System.out.println("Match not found");

}

}

}

**Output**

Match found

## Flags

Flags in the compile() method change how the search is performed. Here are a few of them:

* Pattern.CASE\_INSENSITIVE - The case of letters will be ignored when performing a search.
* Pattern.LITERAL - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.

## Java Threads

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

## Creating a Thread

There are two ways to create a thread.

It can be created by extending the Thread class and overriding its run() method:

public class Main extends Thread {

public static void main(String[] args) {

Main thread = new Main();

thread.start();

System.out.println("This code is outside of the thread");

}

public void run() {

System.out.println("This code is running in a thread");

}

}

**Output**

This code is outside of the thread

This code is running in a thread

If the class **implements** the Runnable interface, the thread can be run by passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:

public class Main implements Runnable {

public static void main(String[] args) {

Main obj = new Main();

Thread thread = new Thread(obj);

thread.start();

System.out.println("This code is outside of the thread");

}

public void run() {

System.out.println("This code is running in a thread");

}

}

**Output**

This code is outside of the thread

This code is running in a thread

## Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

public class Main extends Thread {

public static int amount = 0;

public static void main(String[] args) {

Main thread = new Main();

thread.start();

System.out.println(amount);

amount++;

System.out.println(amount);

}

public void run() {

amount++;

}

}

**Output**

0

2

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the isAlive() method of the thread to check whether the thread has finished running before using any attributes that the thread can change.

public class Main extends Thread {

public static int amount = 0;

public static void main(String[] args) {

Main thread = new Main();

thread.start();

// Wait for the thread to finish

while(thread.isAlive()) {

System.out.println("Waiting...");

}

// Update amount and print its value

System.out.println("Main: " + amount);

amount++;

System.out.println("Main: " + amount);

}

public void run() {

amount++;

}

}

**Output**

Waiting …

Main: 1

Main: 2

## Java Lambda Expressions

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

## Using Lambda Expressions

Lambda expressions are usually passed as parameters to a function:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(5);

numbers.add(9);

numbers.add(8);

numbers.add(1);

numbers.forEach( (n) -> { System.out.println(n); } );

}

}

**Output**

5

9

8

1

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.

import java.util.ArrayList;

import java.util.function.Consumer;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(5);

numbers.add(9);

numbers.add(8);

numbers.add(1);

Consumer<Integer> method = (n) -> { System.out.println(n); };

numbers.forEach( method );

}

}

**Output**

5

9

8

1

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

interface StringFunction {

String run(String str);

}

public class Main {

public static void main(String[] args) {

StringFunction exclaim = (s) -> s + "!";

StringFunction ask = (s) -> s + "?";

printFormatted("Hello", exclaim);

printFormatted("Hello", ask);

}

public static void printFormatted(String str, StringFunction format) {

String result = format.run(str);

System.out.println(result);

}

}

**Output**

Hello!

Hello?

# **Java Files**

File handling is an important part of any application.

Java has several methods for creating, reading, updating, and deleting files.

## Java File Handling

The File class from the java.io package, allows us to work with files.

To use the File class, create an object of the class, and specify the filename or directory name:

import java.io.File; // Import the File class

File myObj = newFile(“filename.txt”); // Specify the filename

## Create a File

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists. Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

**Output**

import java.io.File;

import java.io.IOException;

public class CreateFile {

public static void main(String[] args) {

try {

File myObj = new File("filename.txt");

if (myObj.createNewFile()) {

System.out.println("File created: " + myObj.getName());

} else {

System.out.println("File already exists.");

}

} catch (IOException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

}

**Output**

Filecreated : filename.txt

**Example**

import java.io.File;

import java.io.IOException;

public class CreateFileDir {

public static void main(String[] args) {

try {

File myObj = new File("C:\\Users\\MyName\\filename.txt");

if (myObj.createNewFile()) {

System.out.println("File created: " + myObj.getName());

System.out.println("Absolute path: " + myObj.getAbsolutePath());

} else {

System.out.println("File already exists.");

}

} catch (IOException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

}

File created: filename.txt  
Absolute path: C:\Users\MyName\filename.txt

## Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

import java.io.FileWriter;

import java.io.IOException;

public class WriteToFile {

public static void main(String[] args) {

try {

FileWriter myWriter = new FileWriter("filename.txt");

myWriter.write("Files in Java might be tricky, but it is fun enough!");

myWriter.close();

System.out.println("Successfully wrote to the file.");

} catch (IOException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

}

**Output**

Successfully wrote to the file

## Read a File

In the following example, we use the Scanner class to read the contents of the text file we created in the previous chapter:

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ReadFile {

public static void main(String[] args) {

try {

File myObj = new File("filename.txt");

Scanner myReader = new Scanner(myObj);

while (myReader.hasNextLine()) {

String data = myReader.nextLine();

System.out.println(data);

}

myReader.close();

} catch (FileNotFoundException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

}

**Output**

Files in Java may be tricky but they are fun enough

## Get File Information

To get more information about a file, use any of the File methods:

import java.io.File;

public class GetFileInfo {

public static void main(String[] args) {

File myObj = new File("filename.txt");

if (myObj.exists()) {

System.out.println("File name: " + myObj.getName());

System.out.println("Absolute path: " + myObj.getAbsolutePath());

System.out.println("Writeable: " + myObj.canWrite());

System.out.println("Readable: " + myObj.canRead());

System.out.println("File size in bytes: " + myObj.length());

} else {

System.out.println("The file does not exist.");

}

}

}

File name: filename.txt  
Absolute path: C:\Users\MyName\filename.txt  
Writeable: true  
Readable: true  
File size in bytes: 0

## Delete a File

To delete a file in Java, use the delete() method:

import java.io.File;

public class DeleteFile {

public static void main(String[] args) {

File myObj = new File("filename.txt");

if (myObj.delete()) {

System.out.println("Deleted the file: " + myObj.getName());

} else {

System.out.println("Failed to delete the file.");

}

}

}

Deleted the file: filename.txth

## Delete a Folder

You can also delete a folder. However, it must be empty:

import java.io.File;

public class DeleteFolder {

public static void main(String[] args) {

File myObj = new File("C:\\Users\\MyName\\Test");

if (myObj.delete()) {

System.out.println("Deleted the folder: " + myObj.getName());

} else {

System.out.println("Failed to delete the folder.");

}

}

}

## Add Two Numbers

Learn how to add two numbers in Java:

public class Main {

public static void main(String[] args) {

int x = 5;

int y = 6;

int sum = x + y;

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

}

}

**Output**

11

## Add Two Numbers with User Input

Learn how to add two numbers with user input:

import java.util.Scanner; // Import the Scanner class

public class MyClass {

public static void main(String[] args) {

int x, y, sum;

Scanner myObj = new Scanner(System.in); // Create a Scanner object

System.out.println("Type a number:");

x = myObj.nextInt(); // Read user input

System.out.println("Type another number:");

y = myObj.nextInt(); // Read user input

sum = x + y;

System.out.println("Sum is: " + sum); // Output user input

}

}

**Output**

Type a number

## Reverse a String

You can easily reverse a string by characters with the following example:

public class Main {

public static void main(String[] args) {

String originalStr = "Hello";

String reversedStr = "";

System.out.println("Original string: " + originalStr);

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

reversedStr = originalStr.charAt(i) + reversedStr;

}

System.out.println("Reversed string: "+ reversedStr);

}

}

## Calculate the Sum of an Array

Get the sum of array elements:

public class Main {

public static void main(String[] args) {

int[] myArray = {1, 5, 10, 25};

int sum = 0;

int i;

// Loop through array elements and get the sum

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

sum += myArray[i];

}

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

}

}

**Output**

The sum is : 41

## Area of Rectangle

The area of a rectangle can be found by multiplying the length of the rectangle by the width:

public class Main {

public static void main(String[] args) {

int length = 5;

int width = 2;

int area = length \* width;

System.out.println("Area of rectangle: " + area);

}

}

**Output**

Area of rectangle: 10

## Check Whether a Number is Even or Odd

Find out if a number is even or odd:

public class Main {

public static void main(String[] args) {

int number = 5;

if (number % 2 == 0) {

System.out.println(number + " is even.");

} else {

System.out.println(number + " is odd.");

}

}

}

**Output**

5 is odd