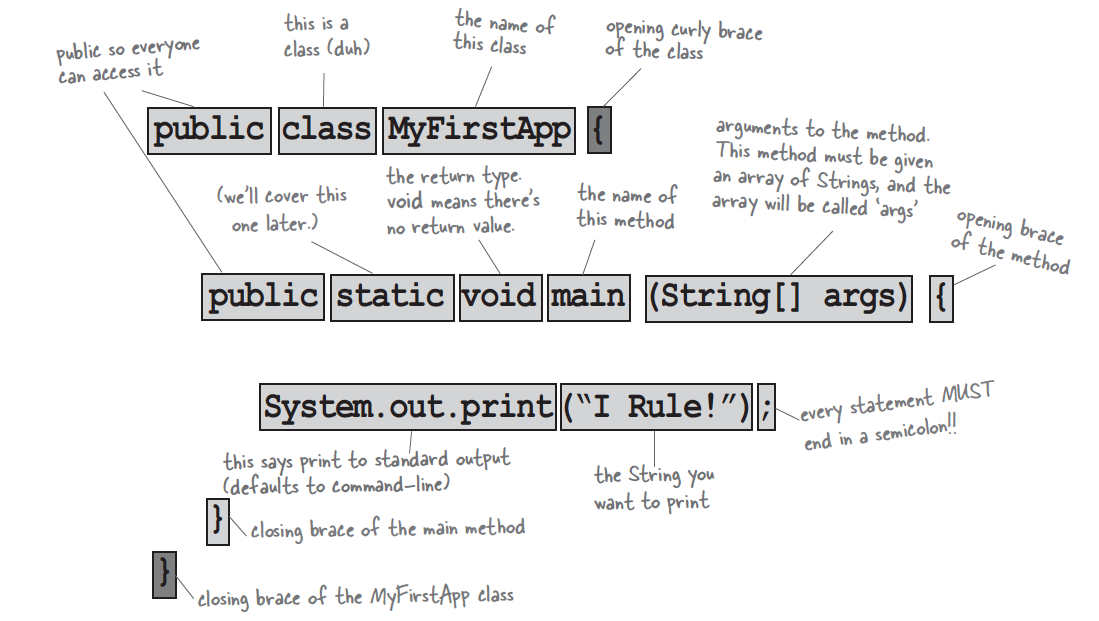
**Java**

Source -> Compiler -> Output -> Virtual Machine

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Source | ->> | Compiler | ->> | Output | ->> | Virtual Machine |
| JAVA code  Save as  Party.java | Compile the Party.java by running *javac.* | Save as Party.class | Run the program by JAVA Virtual Machine,  Ex: Mobile Phone, computer etc. |

****

## What is Data Types in Java?

**Data Types in Java** are defined as specifiers that allocate different sizes and types of values that can be stored in the variable or an identifier. Java has a rich set of data types. Data types in Java can be divided into two parts :

1. **Primitive Data Types**:- which include integer, character, Boolean, and float
2. **Non-primitive Data Types**:- which include classes, arrays and interfaces.

### 

There are 8 primitive types: byte, short, int, long, char, float, double, and Boolean.

### Example

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

float myFloatNum = 5.99f; // Floating point number

char myLetter = 'D'; // Character

boolean myBool = true; // Boolean

String myText = "Hello"; // String

|  |  |  |
| --- | --- | --- |
| **Data 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 |

**Example:**

class Demo {

public static void main(String args[]) {

byte x;

int a = 270;

double b = 128.128;

System.out.println("int converted to byte");

x = (byte) a;

System.out.println("a and x " + a + " " + x);

System.out.println("double converted to int");

a = (int) b;

System.out.println("b and a " + b + " " + a);

System.out.println("\ndouble converted to byte");

x = (byte)b;

System.out.println("b and x " + b + " " + x);

}

}

### 

### Types of Variables in Java

Different types of variables which are listed asfollows:

1. Local Variables
2. Instance Variables
3. Static Variables

**1. Local Variables**

1. A variable defined within a block or method or constructor is called a local variable.
2. These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
3. The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
4. Initialization of the local variable is mandatory before using it in the defined scope.

**2. Instance Variables**

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

* As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
* Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
* Initialization of an instance variable is not mandatory. Its default value is 0.
* Instance variables can be accessed only by creating objects.

**3. Static Variables**

Static variables are also known as class variables.

* These variables are declared similarly as instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor or block.
* Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
* Static variables are created at the start of program execution and destroyed automatically when execution ends.
* Initialization of a static variable is not mandatory. Its default value is 0.
* If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won’t halt the program. The compiler will replace the object name with the class name automatically.
* If we access a static variable without the class name, the compiler will automatically append the class name.

Example:

public class VariableExample{

int myVariable;

static int data = 30;

public static void main(String args[]){

int a = 100;

VariableExample obj = new VariableExample();

System.out.println("Value of instance variable myVariable: "+obj.myVariable);

System.out.println("Value of static variable data: "+VariableExample.data);

System.out.println("Value of local variable a: "+a);

}

}

## **Output**

Value of instance variable myVariable: 0

Value of static variable data: 30

Value of local variable a: 100

**Q. Difference between function and Method:**

**Ans.**

A **function** is a piece of code that is called by name. It can be passed data to operate on (i.e. the parameters) and can optionally return data (the return value). All data that is passed to a function is explicitly passed.

A **method** is a piece of code that is called by a name that is associated with an object. In most respects it is identical to a function except for two key differences:

1. A method is implicitly passed the object on which it was called.
2. A method is able to operate on data that is contained within the class (remembering that an object is an instance of a class - the class is the definition, the object is an instance of that data).

Simple way to remember:

* **F**unction → **F**ree (Free means it can be anywhere, no need to be in an object or class)
* **M**ethod → **M**ember (A member of an object or class)

**Q. Difference between Object and instance:**

**Ans.**

Class : A class is a blue print. Object : It is the copy of the class. Instance : Its a variable which is used to hold memory address of the object.

A very basic analytical example

Class House --> Blueprint of the house. But you can't live in the blue print. You need a physical House which is the instance of the class to live in. i.e., actual address of the object is instance. Instances represent objects.

## **What is statement in Java?**

In Java, a **statement** is an executable instruction that tells the compiler what to perform. It forms a complete command to be executed and can include one or more expressions. A sentence forms a complete idea that can include one or more clauses.

# Q: Expressions, Statements, and Blocks

Now that you understand variables and operators, it's time to learn about *expressions*, *statements*, and *blocks*. Operators may be used in building expressions, which compute values; expressions are the core components of statements; statements may be grouped into blocks.

## Expressions

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

int **cadence = 0**;

**anArray[0] = 100**;

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

int **result = 1 + 2**; // result is now 3

if (**value1 == value2**)

System.out.println(**"value1 == value2"**);

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

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

1 \* 2 \* 3

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

x + y / 100 // ambiguous

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

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

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

x + y / 100

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

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

## Statements

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

* Assignment expressions
* Any use of ++ or --
* Method invocations
* Object creation expressions

Such statements are called expression statements. Here are some examples of expression statements.

// assignment statement

aValue = 8933.234;

// increment statement

aValue++;

// method invocation statement

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

// object creation statement

Bicycle myBike = new Bicycle();

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

// declaration statement

double aValue = 8933.234;

Finally, control flow statements regulate the order in which statements get executed. You'll learn about control flow statements in the next section, [Control Flow Statements](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html)

## Blocks

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, [BlockDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/BlockDemo.java), illustrates the use of blocks:

class BlockDemo {

public static void main(String[] args) {

boolean condition = true;

if (condition) { **// begin block 1**

System.out.println("Condition is true.");

} **// end block one**

else { **// begin block 2**

System.out.println("Condition is false.");

} **// end block 2**

}

}

## **Difference between Constructor and Method**

The following table highlights the major differences between a Constructor and a Method −

| **Basis of Comparison** | **Constructor** | **Method** |
| --- | --- | --- |
| Invocation | The system invokes it implicitly. | It is called during program execution. |
| Inheritance | It is not possible for a subclass to inherit it. | It is anything that can be inherited by a subclass. |
| Uses | It is utilized in the process of initializing an object. | It is utilized for the purpose of demonstrating the operation of an object. |
| Return Type | There is no return type associated with it. | It is equipped with a return type. |
| Name | It is a common misconception that the name of the constructor does not have to match the name of the class. | It is a common misconception that the name of the method and the class cannot be the same. |

## **Conclusion**

Methods are a set of instructions that specify how a class's data can be utilised, and they are made up of one or more lines of code. Methods are analogous to a subprogram in the sense that they process data and return a value. It is possible to use the method name to invoke its execution at any point inside the body of a program.

Constructors are quite similar to methods, with the key difference being that constructors are invoked whenever an instance of an object is created. Constructors, as opposed to methods, are invoked in order to create and initialise objects that have not yet been created. In Java, constructors must be called with the same name as the name of the class in which they live, whereas methods can have any name and can be called directly either with a reference to the class or an object reference. Constructors are an exception to this rule.

**Loop:**

There are 3 looping constructs in java, *while*, *do-while* and *for*

**While Loop:**

While syntax:

while (*condition*) {

*// code block to be executed*

}

Example:

int i = 0;

while (i < 5) {

System.out.println(i);

i++;

}

public static void main(String[] arg) {  
 int x = 1;  
 System.*out*.println("Before the Loop");  
 while (x < 15) {  
 System.*out*.println("In the loop");  
 System.*out*.println("Value of x is: " + x);  
 x = x + 2;  
 }  
 System.*out*.println("This is after the loop");  
  
}

**Do-While:**

Syntax

do {

*// code block to be executed*

}

while (condition);

Example

int i = 0;  
do {

System.out.println(i);

i++;

}

while (i < 5);

**For loop:**

Syntax

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

*// code block to be executed*

}

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

**Statement 2** defines the condition for executing the code block.

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

Example:

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

System.out.println(i);

}

## **For-Each Loop**

here 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):

### Syntax

for (type variableName : arrayName) {

*// code block to be executed*

}

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

### Example

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

for (String i : cars) {

System.out.println(i);

}

# **Java Break and Continue**

## **Java Break**

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

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

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

### Example

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

if (i == 4) {

break;

}

System.out.println(i);

}

## **Java Continue**

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

This example skips the value of 4:

### Example

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

if (i == 4) {

continue;

}

System.out.println(i);

}

**Java Arrays:**

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

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

## **Access the Elements of an Array**

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

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

### Example

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

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

// Outputs Volvo

## **Change an Array Element**

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

### Example

cars[0] = "Opel";

### Example

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

cars[0] = "Opel";

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

// Now outputs Opel instead of Volvo

# **Java Arrays Loop**

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

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

### Example

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

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

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

}

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

### Syntax

for (type variable : arrayname) {

...

}

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

### Example

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

for (String i : cars) {

System.out.println(i);

}

# **Java Multi-Dimensional Arrays**

A multidimensional array is an array of arrays.

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

### Example

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

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

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

### Example

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

int x = myNumbers[1][2];

System.out.println(x); // Outputs 7

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]);

}  
 }

}

}

**Conditions: IF**

Syntax:

if (condition) {

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

}

Example

if (20 > 18) {

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

}

### Example

int x = 20;

int y = 18;

if (x > y) {

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

}

### If-else

### Syntax

if (condition) {

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

} else {

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

}

### Example

int time = 20;

if (time < 18) {

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

} else {

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

}

// Outputs "Good evening."

### Syntax

if (condition1) {

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

} else if (condition2) {

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

} else {

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

}

### Example

int time = 22;

if (time < 10) {

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

} else if (time < 20) {

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

} else {

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

}

// Outputs "Good evening."

Example: When you are asked to write a program to add to whole two numbers that will input from the terminal.

Solution:

import java.util.Scanner;

public class Exercise5 {

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.print("Type the first whole number and press Enter:");

String firstNum = sc.nextLine();

System.out.print("Type the second whole number and press Enter:");

String secondNum = sc.nextLine();

*//fix code below this line*

int sum = Integer.parseInt(firstNum) + Integer.parseInt(secondNum);

System.out.println(firstNum + " + " + secondNum + " = " + String.valueOf(sum));

*//fix code above this line*

}

}

Question:

**Problem**  
Use the variable x as you write this program. x will represent a string. Write a program that determines if x is a vowel (*a, e, i, o,* and *u* ). If yes, print \_ is a vowel, where the blank is the value of x. If no, print \_ is not a vowel, where the blank is the value of x.

Solution:

public class Exercise5 {

public static void main(String args[]) {

String x = args[0];

*//add code below this line*

if(x.equals("a")||x.equals("e")||x.equals("i")||x.equals("o")||x.equals("u")){

System.out.println(x +" is a vowel");

}

else{

System.out.println(x +" is not a vowel");

}

}

}

**Problem**  
Use the variable x as you write this program. x will represent a string. Write a program that determines if x is a primary color (red, blue, or yellow). If yes, print \_ is primary color, where the blank is the value of x. If no, print \_ is not a primary color, where the blank is the value of x.

Solution:

public class Exercise4 {

public static void main(String args[]) {

String x = args[0];

if( x.equals("yellow") || x.equals("red") || x.equals("blue") )

{System.out.println(x + " is a primary color");}

else

{System.out.println(x + " is not a primary color");}

}

}

Alternative solution:

switch (x) {

case "yellow":

case "red":

case "blue":

System.out.println(x + " is a primary color"); break;

default: System.out.println(x + " is not a primary color"); break;

}

**Problem**  
Use the variable x as you write this program. x will represent a positive integer. Write a program that determines if x is divisible by 5 and even. If yes, print \_ is divisible by 5 and even, where the blank is the value of x. If no, print \_ is not divisible by 5 or it is odd, where the blank is the value of x.

Solution:

if ( x % 5 == 0 && x % 2 == 0)

{ System.out.println(**String**.valueOf(x) + " is divisible by 5 and even"); }

else

{ System.out.println(**String**.valueOf(x) + " is not divisible by 5 or it is odd"); }

**Problem**  
Use the variable x as you write this program. x will represent a positive integer. Write a program that determines if x is divisible by 5. If yes, print \_ is divisible by 5, where the blank is the value of x. If no, print \_ is not divisible by 5, where the blank is the value of x.

Solution:

if ( x % 5 == 0 )

{ System.out.println(**String**.valueOf(x) + " is divisible by 5"); }

else

{ System.out.println(**String**.valueOf(x) + " is not divisible by 5"); }

**Problem**  
Use the variable x as you write this program. x will represent a positive integer. Write a program that determines if x is between 0 and 25 or between 75 and 100. If yes, print the message:\_ is between 0 and 25 or 75 and 100, where the blank would be the value of x. The program should do nothing if the value of x does not fit into either range.

Note: These are *inclusive* ranges - 0, 25, 75 and 100 should print out the message.

Solution:

if( (**x** >= 0 && **x** <=25) || (**x** >= 75 && **x** <=100) ){

System.out.println(String.valueOf(**x**) + " is between 0 and 25 or 75 and 100")*;*

*}*

Turtle Game:

Solution:

public class WhileLoopImages {

public static void main(String args[]) {

Turtle tina = new Turtle(0, 100); *//change parameters to make tina visible*

*//add code below this line*

tina.penColor("blue");

tina.shape("arrow");

tina.speed(200);

int i = 0;

while (i < 4) {

tina.forward(75);

tina.right(90);

tina.forward(25);

tina.right(90);

tina.forward(25);

tina.right(90);

tina.forward(25);

i++;

}

*//add code above this line*

}

}

This is another example:

public class WhileLoopImages {

public static void main(String args[]) {

Turtle tina = new Turtle(0, 100); *//change parameters to make tina visible*

*//add code below this line*

tina.penColor("red");

tina.shape("square");

tina.speed(10);

int i = 0;

while (i < 360) {

tina.forward(1);

tina.right(1);

i++;

}

*//add code above this line*

}

}

### Nested Loop Coding Challenge 2

Using nested loops, write some code that outputs the following:

<<<<<<<<<<

>>>>>>>>>>

<<<<<<<<<<

>>>>>>>>>>

<<<<<<<<<<

Solution:

public class NestedLoops {

public static void main(String args[]) {

*//add code below this line*

for(int row = 0; row < 5; row++){

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

if (row%2==0){

System.out.print("<");

} else{

System.out.print(">");

}

}

System.out.println(""); *//adds new line*

}

*//add code above this line*

}

}

Using nested loops, write some code that outputs the following:

1

22

333

4444

55555

Solution:

public class NestedLoops {

public static void main(String args[]) {

*//add code below this line*

for (int row = 1; row <= 5; row++) {

for (int col = 1; col <= row; col++) {

System.out.print(row);

}

System.out.println("");

}

System.out.println(""); *//adds new line*

}

*//add code above this line*

}

Problem:

Fill in the blanks such that the nested loop will print out:

&&

\*

\*

\*

&&

\*

\*

\*

&&

\*

\*

\*

&&

\*

\*

\*

Solution:

package codio;

public class Assignment1 {

public static void main(String[] args) {

for(int i = 0; i < 4; i++) {

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

for(int j = 0; j < 3; j++) {

System.out.println("\*");

}

}

}

}

Problem:

Rearrange the code blocks below to create a program that outputs the following:

Even Odd Even

Odd Even Odd

Even Odd Even

Odd Even Odd

Even Odd Even

package codio;

public class Assignment2 {

public static void main(String[] args) {

for(int row = 0; row < 5; row++){

for(int col = 0; col < 1; col++) {

if (row%2==0){

System.out.print("even odd even");

} else{

System.out.print("odd even odd");

}

}

System.out.println(""); //adds new line

}

}

}

Problem:

You are trying to produce a program that takes in two **integer** arguments from the user and then prints the **consecutive sum** of all numbers between those integers inclusively. For example, arguments 5 and 8 will result in a sum of 26 (5 + 6 + 7 + 8 = 26). If the integer arguments are the same value, the program will simply print that value as the sum.

Solution:

package codio;

public class Assignment4 {

public static void main(String[] args) {

int sum = 0;

int b=50;

int a=10;

while (a <= b) {

sum += a;

a++;

}

if (a == b) {

sum = a;

}

System.out.println(sum);

}

}

Problem:

For this assignment, you will use a nested loop to produce the unique output below.

....1

...2

..3

.4

5

Solution:

**for** (**int** i = 1; i < 6; i++) {

**for** (**int** j = 5 - i; j > 0; j--) {

System.out.print(".");

}

System.out.println(i);

}

# **Java Methods**

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

## **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

Create a method inside Main:

public class Main {

static void myMethod() {

// code to be executed

}

}

#### **Example Explained**

* myMethod() is the name of the method
* static means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
* void means that this method does not have a return value. You will learn more about return values later in this chapter

## **Call a Method**

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, myMethod() is used to print a text (the action), when it is called:

### Example

Inside main, call the myMethod() method:

public class Main {

static void myMethod() {

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

}

public static void main(String[] args) {

myMethod();

}

}

// Outputs "I just got executed!"

Add a fname parameter of type String to myMethod, and output "John Doe".

static void myMethod( ) {

System.out.println( + " Doe");

}

public static void main(String[] args) {

myMethod("John");

}

Insert the missing part to print the number 8 in main, by using a specific **keyword** inside myMethod:

static int myMethod(int x) {

return 5 + x;

}

public static void main(String[] args) {

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

}

# **Exercise:**

Follow the comments to insert the missing parts of the code below:

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

static void checkAge(int age ) {

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

if(age<18) {

System.out.println("Access denied");

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

} else {

System.out.println("Access granted");

}

}

public static void main(String[] args) {

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

checkAge(20);

}

In eclipse:

package practiceByLesson;

public class MethodPractice {

static void checkAge( int age) {

if(age<18) {

System.out.println("Access Denied");

}

else {

System.out.println("Access granted");

}

}

public static void main(String[] args) {

checkAge(20);

}

}

## **Multiple Parameters**

You can have as many parameters as you like:

### Example

public class Main {

static void myMethod(String fname, int age) {

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

}

public static void main(String[] args) {

myMethod("Liam", 5);

myMethod("Jenny", 8);

myMethod("Anja", 31);

}

}

// Liam is 5

// Jenny is 8

// Anja is 31

# **Java Method Overloading**

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

### Example

int myMethod(int x)

float myMethod(float x)

double myMethod(double x, double y)

Consider the following example, which has two methods that add numbers of different type:

### Example

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);

}

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

### Example

int myMethod(int x)

float myMethod(float x)

double myMethod(double x, double y)

Consider the following example, which has two methods that add numbers of different type:

### Example

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);

}

## **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);

}

}

## **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

}

}

## **Recursion Example**

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

### Example

Use recursion to add all of the numbers up to 10.

public class Main {

public static void main(String[] args) {

int result = sum(6);

System.out.println(result);

}

public static int sum(int k) {

if (k > 0) {

return k + sum(k - 1);

} else {

return 0;

}

}

}

### Example

Use recursion to add all of the numbers between 5 to 10.

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;

}

}

}

# **Java OOP**

## **Java - What is OOP?**

OOP 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 - What are Classes and Objects?**

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

## **class**

Fruit

## **objects**

Apple

Banana

Mango

Another example:

## **class**

Car

## **objects**

Volvo

Audi

Toyota

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

You will learn much more about [classes and objects](https://www.w3schools.com/java/java_classes.asp) in the next chapter.

# **Java Classes and Objects**

## **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:

### Main.java

Create a class named "Main" with a variable x:

public class Main {

int x = 5;

}

## **Create an Object**

In Java, an object is created from a class. We have already created the class named Main, so now we can use this to create objects.

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

### Example

Create an object called "myObj" and print the value of x:

public class Main {

int x = 5;

public static void main(String[] args) {

Main **myObj** = new Main();

System.out.println(myObj.x);

}

}

## **Multiple Objects**

You can create multiple objects of one class:

### Example

Create two objects of Main:

public class Main {

int x = 5;

public static void main(String[] args) {

Main **myObj1** = new Main(); // Object 1

Main **myObj2** = new Main(); // Object 2

System.out.println(myObj1.x);

System.out.println(myObj2.x);

}

}

## **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

#### **Main.java**

public class Main {

int x = 5;

}

#### **Second.java**

class Second {

public static void main(String[] args) {

Main **myObj** = new Main();

System.out.println(myObj.x);

}

}

When both files have been compiled:

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

Run the Second.java file:

C:\Users\Your Name>java Second

And the output will be:

5

## **Java Class Attributes**

In the previous chapter, we used the term "variable" for x in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class:

### Example

Create a class called "Main" with two attributes: x and y:

public class Main {

int x = 5;

int y = 3;

}

Another term for class attributes is **fields**.

## **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

Create an object called "myObj" and print the value of x:

public class Main {

int x = 5;

public static void main(String[] args) {

Main myObj = new Main();

System.out.println(myObj.x);

}

}

## **Modify Attributes**

You can also modify attribute values:

### Example

Set the value of x to 40:

public class Main {

int x;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 40;

System.out.println(myObj.x);

}

}

Or override existing values:

### Example

Change the value of x to 25:

public class Main {

int x = 10;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 25; // x is now 25

System.out.println(myObj.x);

}

}

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: cannot assign a value to a **final** variable

System.out.println(myObj.x);

}

}

## **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:

### Example

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

public class Main {

int x = 5;

public static void main(String[] args) {

Main myObj1 = new Main(); // Object 1

Main myObj2 = new Main(); // Object 2

myObj2.x = 25;

System.out.println(myObj1.x); // Outputs 5

System.out.println(myObj2.x); // Outputs 25

}

}

## **Multiple Attributes**

You can specify as many attributes as you want:

### Example

public class Main {

String fname = "John";

String lname = "Doe";

int age = 24;

public static void main(String[] args) {

Main myObj = new Main();

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

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

}

}

## **Java Class Methods**

You learned from the [Java Methods](https://www.w3schools.com/java/java_methods.asp) chapter that methods are declared within a class, and that they are used to perform certain actions:

### Example

Create a method named myMethod() in Main:

public class Main {

static void myMethod() {

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

}

}

myMethod() prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

### Example

Inside main, call myMethod():

public class Main {

static void myMethod() {

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

}

public static void main(String[] args) {

myMethod();

}

}

// Outputs "Hello World!"

## **Static vs. Public**

You will often see Java programs that have either static or public attributes and methods.

In the example above, we created a static method, which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:

### Example

An example to demonstrate the differences between static and public **methods**:

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

// myPublicMethod(); This would compile an error

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

myObj.myPublicMethod(); // Call the public method on the object

}

}

## **Access Methods With an Object**

### Example

Create a Car object named myCar. Call the fullThrottle() and speed() methods on the myCar object, and run the program:

// 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

}

}

// The car is going as fast as it can!

// Max speed is: 200

### Example explained

1) We created a custom Main class with the class keyword.

2) We created the fullThrottle() and speed() methods in the Main class.

3) The fullThrottle() method and the speed() method will print out some text, when they are called.

4) The speed() method accepts an int parameter called maxSpeed - we will use this in **8)**.

5) In order to use the Main class and its methods, we need to create an **object** of the Main Class.

6) Then, go to the main() method, which you know by now is a built-in Java method that runs your program (any code inside main is executed).

7) By using the new keyword we created an object with the name myCar.

8) Then, we call the fullThrottle() and speed() methods on the myCar object, and run the program using the name of the object (myCar), followed by a dot (.), followed by the name of the method (fullThrottle(); and speed(200);). Notice that we add an int parameter of **200** inside the speed() method.

### Remember that..

The dot (.) is used to access the object's attributes and methods.

To call a method in Java, write the method name followed by a set of parentheses **()**, followed by a semicolon (;).

A class must have a matching filename (Main and **Main.java**).

## **Using Multiple Classes**

Like we specified in the [Classes chapter](https://www.w3schools.com/java/java_classes.asp), it is a good practice to create an object of a class and access it in another class.

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:

* Main.java
* Second.java

#### **Main.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);

}

}

#### **Second.java**

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

}

}

When both files have been compiled:

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

Run the Second.java file:

C:\Users\Your Name>java Second

And the output will be:

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 constructor:

// Create a Main class

public class Main {

int x; // Create a class attribute

// Create a **class constructor** for the Main class

public Main() {

x = 5; // Set the initial value for the class attribute x

}

public static void main(String[] args) {

Main myObj = new Main(); // Create an object of class Main (This will **call the constructor**)

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

}

}

// Outputs 5

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

## **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);

}

}

// Outputs 5

You can have as many parameters as you want:

### Example

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);

}

}

// Outputs 1969 Mustang

# **Java Modifiers**

## **Modifiers**

By now, you are quite 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

## **Static**

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

### Example

An example to demonstrate the differences between static and public methods:

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

// myPublicMethod(); This would output an error

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

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

}

}

## **Abstract**

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

### Example

// Code from filename: Main.java

// abstract class  
abstract class Main {

public String fname = "John";

public int age = 24;

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");

}

}

// End code from filename: Main.java

// Code from filename: Second.java

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  
 }

}

## **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**

You learned from the previous chapter that 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:

### Example

public class Person {

private String name; // private = restricted access

// Getter

public String getName() {

return name;

}

// Setter

public void setName(String newName) {

this.name = newName;

}

}

#### **Example explained**

The get method returns the value of the variable name.

The set method takes a parameter (newName) and assigns it to the name variable. The this keyword is used to refer to the current object.

However, as the name variable is declared as private, we **cannot** access it from outside this class:

### Example

public class Main {

public static void main(String[] args) {

Person myObj = new Person();

myObj.name = "John"; // error

System.out.println(myObj.name); // error

}

}

Instead, we use the getName() and setName() methods to access and update the variable:

### Example

public class Main {

public static void main(String[] args) {

Person myObj = new Person();

myObj.setName("John"); // Set the value of the name variable to "John"

System.out.println(myObj.getName());

}

}

// Outputs "John"

## **Why Encapsulation?**

* Better control of class attributes and methods
* Class attributes can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
* Flexible: the programmer can change one part of the code without affecting other parts
* Increased security of data

# **Java Packages**

## **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 complete list can be found at Oracles website: <https://docs.oracle.com/javase/8/docs/api/>.

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:

### Syntax

import package.name.Class; // Import a single class

import package.name.\*; // Import the whole package

## **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:

### Example

import java.util.Scanner;

In the example above, java.util is a package, while Scanner is a class of 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 a complete line:

### Example

Using the Scanner class to get user input:

import java.util.Scanner;

class MyClass {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in);

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

String userName = myObj.nextLine();

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

}

}

## **Import a Package**

There are many packages to choose from. In the previous example, we used the Scanner class from the java.util package. 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.\*;

## **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:

### Example

└── root

└── mypack

└── MyPackageClass.java

To create a package, use the package keyword:

### MyPackageClass.java

package mypack;

class MyPackageClass {

public static void main(String[] args) {

System.out.println("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"; // Vehicle attribute

public void honk() { // Vehicle method

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

}

}

class Car extends Vehicle {

private String modelName = "Mustang"; // Car attribute

public static void main(String[] args) {

// Create a myCar object

Car myCar = new Car();

// Call the honk() method (from the Vehicle class) on the myCar object

myCar.honk();

// Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class

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

}

}

## **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);

}

}

// Outputs 15 (5 + 10)

## **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);

}

}

If you try to access a private inner class from an outside class, an error occurs:

Main.java:13: error: OuterClass.InnerClass has private access in OuterClass  
    OuterClass.InnerClass myInner = myOuter.new InnerClass();  
              ^

## **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:

### Example

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

class Main {

public static void main(String[] args) {

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

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

String userName = myObj.nextLine(); // Read user input

System.out.println("Username is: " + userName); // Output user input

}

}

## **Display Current Date**

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

### Example

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

}

}

## **Display Current Time**

To display the current time (hour, minute, second, and nanoseconds), import the java.time.LocalTime class, and use its now() method:

### Example

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

public class Main {

public static void main(String[] args) {

LocalTime myObj = LocalTime.now();

System.out.println(myObj);

}

}

## **Display Current Date and Time**

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

### Example

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);

}

}

## **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:

### Example

Create an ArrayList object called **cars** that will store strings:

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

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object

If you don't know what a package is, read our [Java Packages Tutorial](https://www.w3schools.com/java/java_packages.asp).

## **Add Items**

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

To add elements to the ArrayList, simply use the add() method. The add() method will add whatever element that is specified inside parentheses () to the end of the ArrayList by default. If an element is added to an empty ArrayList, that element will be the first and only element in the ArrayList.

o add an element to a *specific index* in the ArrayList, you can use the add() method with two parameters inside the parentheses (). The first parameter is the index where you want the element to be stored at and the second parameter is the element itself. For example, numbers.add(0, 12) will add the number 12 to index 0 causing 12 to become the first element in the ArrayList. This will cause all of the elements to the right of 12 to move up by 1 index number.

## Removing ArrayList Elements

To remove an element from an ArrayList, use the remove() method and specify the ArrayList index of the element you want to be removed as a parameter inside the parentheses (). Deleting an element will cause all elements to the right of that element to move down by 1 index number.

### Example

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);

}

}

## **Access an Item**

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

### Example

cars.get(0);

## **Change an Item**

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

### Example

cars.set(0, "Opel");

## **Remove an Item**

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

### Example

cars.remove(0);

To remove all the elements in the ArrayList, use the clear() method:

### Example

cars.clear();

## **ArrayList Size**

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

### Example

cars.size();

# ***Getting and Setting Elements***

## Getting ArrayList Elements

To get or access ArrayList elements, use the get() method and include the index as a parameter inside parentheses ().

## Setting ArrayList Elements

To set or modify ArrayList elements, use the set() method which includes two parameters within parentheses (). The first parameter specifies the ArrayList index and the second parameter specifies the element that will replace the current value at the index. For example, contact.set(2, "Email") will modify the element at index 2 and change it to Email.

Question:

Construct a program using the code blocks below so that the following will be printed:

Parker

Solution:

package secondCourse;

import java.util.ArrayList;

public class ArrayListLearning {

public static void main(String[] args) {

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

name.add("Peter");

name.set(0, "Parker");

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

}

}

Q: Write a program by using ArrayList to print the BMI

Solution:

package secondCourse;

import java.util.ArrayList;

public class Assignment3 {

public static void main( String[] args) {

ArrayList<Double> BMI = new ArrayList<Double>();

BMI.add(18.5);

BMI.add(25.0);

BMI.add(30.0);

for(Double d: BMI) {

System.out.println(d);

}

}

}

## **Q: Difference Between Array and ArrayList**

| **Method/Types** | **ArrayList** | **Array** |
| --- | --- | --- |
| Create | ArrayList<Type> var = new ArrayList<Type>() | type[] var = new type[num] or type[] var = {element, element…} |
| Find number of elements | var.size() | var.length |
| Access an element | var.get(index) | var[index] |
| Modify and element | var.set(index, element) | var[index] = element |
| Add an element | var.add(element) or var.add(index, element) | n/a |
| Remove an element | var.remove(index) | n/a |
| for loop | for (int i = 0; i < var.size(); i++) {System.out.println(var.get(i));} | for (int i = 0; i < var.length; i++) {System.out.println(var[i]);} |
| Enhanced for loop | for (Type i : var) {System.out.println(i)} | for (type i : var) {System.out.println(i)} |
| Common compatible types | Integer, Double, Boolean, Strings | int, double, boolean, Strings |

**Q. Finding a Minimum or Maximum value**

**Solution:**

**For Minimum:**

ArrayList<Integer> grades = **new** ArrayList<Integer>();

grades.add(72);

grades.add(84);

grades.add(63);

grades.add(55);

grades.add(98);

**int** min = grades.get(0); *//set min to the first element in the array*

**for** (**int** i : grades) { *//enhanced for loop*

**if** (i < min) { *//if element is less than min*

min = i; *//set min to element that is less*

}

}

*//elements are not modified so enhanced for loop can be used*

System.out.println("The lowest grade is " + min); *//print lowest element*

**For Maximum value:**

ArrayList<Integer> grades = **new** ArrayList<Integer>();

grades.add(72);

grades.add(84);

grades.add(63);

grades.add(55);

grades.add(98);

**int** max = grades.get(0);

**for** (**int** i : grades) {

**if** (i > max) {

max = i;

}

}

System.out.println("The highest grade is " + max);

### Q. Reversing the Order of Elements

import java.util.ArrayList;

public class ArrayListAlgorithms {

public static void main(String args[]) {

*//add code below this line*

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

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

letters.add("A");

letters.add("B");

letters.add("C");

letters.add("D");

letters.add("E");

int original = letters.size(); *//original size*

*//regular for loops needed to access element indices*

for (int i = letters.size() - 1; i >= 0; i--) {

letters.add(letters.get(i));

} *//add elements in reverse order to the ArrayList*

for (int j = 0; j < original; j++) {

letters.remove(0);

} *//remove all the original elements*

System.out.println(letters); *//print new ArrayList*

}

}

**Q.** On a student’s first four tests, they received scores in the following order: 68, 92, 100, and 88.

Fill in the blanks below with code so that the program will print You got a perfect score! if the student scored as least one 100 and will print Keep up the hard work! if they did not.

Solution:

public static void main(String[] args) {

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

scores.add(68);

scores.add(92);

scores.add(100);

scores.add(88);

String feedback = "";

for( int i: scores) {

if(i==100) {

feedback = "You got a perfect score!";

break;

}

else{

feedback = "Keep up the hard work!";

}

}

System.out.println(feedback);

}

}

## **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:

### Example

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));

}

}

}

You can also loop through an ArrayList with the **for-each** loop:

### Example

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 (String i : cars) {

System.out.println(i);

}

}

}

## **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:

### Example

Create an ArrayList to store numbers (add elements of type Integer):

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);

}

}

}

## **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:

### Example

Sort an ArrayList of Strings:

import java.util.ArrayList;

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

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); // Sort cars

for (String i : cars) {

System.out.println(i);

}

}

}

### Example

Sort an ArrayList of Integers:

import java.util.ArrayList;

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

public class Main {

public static void main(String[] args) {

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

myNumbers.add(33);

myNumbers.add(15);

myNumbers.add(20);

myNumbers.add(34);

myNumbers.add(8);

myNumbers.add(12);

Collections.sort(myNumbers); // Sort myNumbers

for (int i : myNumbers) {

System.out.println(i);

}

}

}

**Question: Complete the program below with code so that the program will print Even if the array element is an even number and Odd if it is odd.**

**Solution:**

package secondCourse;

public class Assignment1 {

public static void main(String[] args) {

int[ ] numbers = {15, 48, 22, 93, 75};

for(int i: numbers) {

if(i%2==0) {

System.out.println("Even");

}

else {

System.out.println("Odd");

}

}

}

}

**Q: Given the following output:**

First test: 84

Second test: 76

Third test: 97

Fill in the blanks below with code to produce the output mentioned above.

**Solution:**

package secondCourse;

public class Assignment2 {

public static void main(String[] args) {

String [] test = {"First Test : ", "Second Test: ", "Third Test: "};

int [] score = new int[3];

score[0]=84;

score[1]=76;

score[2]=97;

System.out.println(test[0]+score[0]);

System.out.println(test[1]+score[1]);

System.out.println(test[2]+score[2]);

}

}

## **Java LinkedList**

In the previous chapter, you learned about the [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) class. The LinkedList class is almost identical to the ArrayList:

### Example

// Import the LinkedList class

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);

}

}

## **Java HashMap**

In the [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) chapter, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (int type). 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:

### Example

Create a HashMap object called **capitalCities** that will store String **keys** and String **values**:

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

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

## **Add Items**

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

### Example

// Import the HashMap class

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

// Create a HashMap object called capitalCities

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

// Add keys and values (Country, City)

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

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

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

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

System.out.println(capitalCities);

}

}

## **Access an Item**

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

### Example

capitalCities.get("England");

## **Remove an Item**

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

### Example

capitalCities.remove("England");

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

### Example

capitalCities.clear();

## **HashMap Size**

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

### Example

capitalCities.size();

## **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:

### Example

// Print keys

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

System.out.println(i);

}

### Example

// Print values

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

System.out.println(i);

}

### Example

// Print keys and values

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

System.out.println("key: " + i + " value: " + capitalCities.get(i));

}

## **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:

### Example

Create a HashMap object called **people** that will store String **keys** and Integer **values**:

// 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("key: " + i + " value: " + people.get(i));

}

}

}

## **Java HashSet**

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

### Example

Create a HashSet object called **cars** that will store strings:

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

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

## **Add Items**

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

### Example

// 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);

}

}

**Note:** In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

## **Check If an Item Exists**

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

### Example

cars.contains("Mazda");

## **Remove an Item**

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

### Example

cars.remove("Volvo");

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

### Example

cars.clear();

## **HashSet Size**

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

### Example

cars.size();

## **Loop Through a HashSet**

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

### Example

for (String i : cars) {

System.out.println(i);

}

## **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:

### Example

Use a HashSet that stores Integer objects:

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.");

}

}

}

}

## **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:

### Example

// Import the ArrayList class and the Iterator class

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());

}

}

## **Looping Through a Collection**

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

### Example

while(it.hasNext()) {

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

}

## **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.

### Example

Use an iterator to remove numbers less than 10 from a collection:

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);

}

}

## **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:

### Example

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);

}

}

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:

### Example

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());

}

}

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":

### Example

public class Main {

public static void main(String[] args) {

Integer myInt = 100;

String myString = myInt.toString();

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

}

}

## **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:

### Syntax

try {

// *Block of code to try*

}

catch(Exception e) {

// *Block of code to handle errors*

}

Consider the following example:

This will generate an error, because **myNumbers[10]** does not exist.

public class Main {

public static void main(String[ ] args) {

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

System.out.println(myNumbers[10]); // error!

}

}

The output will be something like this:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10  
        at Main.main(Main.java:4)

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

### Example

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.");

}

}

}

The output will be:

Something went wrong.

## **Finally**

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

### Example

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.");

}

}

}

The output will be:

Something went wrong.  
The 'try catch' is finished.

**Use myObj to access and print the value of the x attribute of MyClass.**

public class MyClass {

int x = 5;

public static void main(String[] args) {

MyClass myObj = new MyClass();

System.out.println(myObj.x);

}

}

**Call myMethod on the object.**

public class MyClass {

public void myMethod() {

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

}

public static void main(String[] args) {

MyClass myObj = new MyClass();

myObj.MyMethod();

}

}

Create and call a **class constructor** of MyClass  
Follow the comments to insert the missing parts of the code below:

// Create a MyClass class

public class MyClass {

int x; // Create a class attribute x

// Create a class constructor for the MyClass class

public MyClass() {

x = 5; // Set the initial value for the class attribute x to **5**

}

public static void main(String[] args) {

// Create an **myObj** object of class MyClass (This will call the constructor)

MyClass myObj = new MyClass();

// Print the value of x

System.out.println(myObj.x);

}

}

The class below should not be inherited by other classes. Add the correct modifier:

Final class MyClass

The Car class should inherit the attributes and methods from the Vehicle class. Add the correct keyword to make this possible.

class Car Extends Vehicle

Insert the missing parts to handle the error in the code below.

try{

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

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

} catch (Exception e) {

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

}

Insert the missing keyword to execute code, after try..catch, regardless of the result.

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.");

}