ADVANCE PROGRAMMING (JAVA LANGUAGE) PART 1

By

Dr. Amal Sufuih Ajrash

Course Goal

This course is a training on programming in the Java language for students who have completed the programming concepts course or have some experience in the field of programming. Students will create Java applications with a focus on correct object-oriented programming techniques that will later become familiar with object-oriented design, including creating classes in Java and using existing classes as stipulated in the current version of the Java API.

Good luck

Advance Programming Course Syllabus

1. An Overview of Java.
2. Concept of Class and Object.
3. Data Types, Variables, and Arrays.
4. Operators.
5. Control Statements.
6. Inheritance.
7. Method Overriding.
8. Packages and Interfaces.
9. Object Class.
10. Exception Handling.
11. Java's Built-in Exceptions.
12. Multithreaded Programming.
13. Internet Addressing.

Write Once, Run Anywhere

Java Programming Fundamentals

1- What is Java?

Java is a popular programming language, created in 1995. It is a powerful general-purpose programming language. According to Oracle, the company that owns Java, Java runs on 3 billion devices worldwide, which makes Java one of the most popular programming languages.

It is used for:

- Mobile applications (specially Android apps) and desktop applications.
- Web applications
- Games
- Database connection and big data processing
- And much, much more!

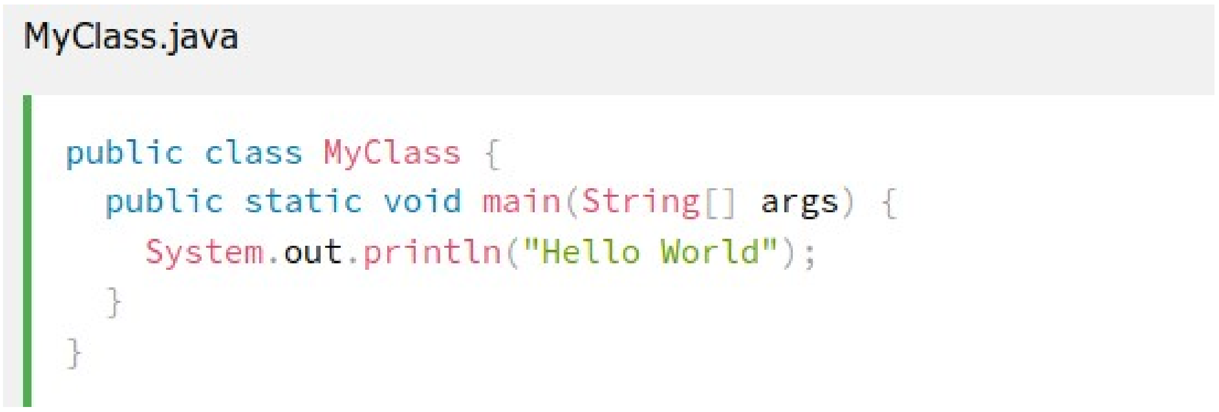
Our Java tutorial will guide you to learn Java one step at a time.

2- Why Use Java?

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

3- Java Quickstart

In Java, every application begins with a class name, and that class must match the filename. Let's create our first Java file, called MyClass.java. The file should contain a "Hello World" message and print it to the screen, which is written with the following code:



MyClass.java

```java
public class MyClass {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

When you run the program, the output will be:

```
Hello World
```

Example explained

`Public class MyClass { ... }`

In Java, every application begins with a class definition. Every line of code must be inside a `class`. In our example, we named the class **MyClass**. 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.

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

This is the main method. Every application in Java must contain the main method.

The Java compiler starts executing the code from the main method. The main method must be inside the class definition.

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

The following code prints the string inside quotation marks `Hello World` to standard output (your screen). Notice, this statement is inside the main function, which is inside the class definition.

4- Java Variables and (Primitive) Data Types

In this tutorial, you will learn about variables, how to create them, and different data types that Java programming language supports for creating variables.

Java Variables

A variable is a location in memory (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier).

How to declare variables in Java?

Here's an example to declare a variable in Java.

```
int price = 80;
```

Here, *price* is a variable of *int* data type and is assigned value 80. Meaning, the *price* variable can store integer values. You will learn about Java data types in detail later in the article.

In the example, we have assigned value to the variable during declaration. However, it's not mandatory. You can declare variables without assigning the value, and later you can store the value as you wish. For example,

```
int price;
price = 80;
```

The value of a variable can be changed in the program, hence the name 'variable'. For example,

```
int price = 80;
... .. ...
price = 90;
```

Java is a statically-typed language. It means that all variables must be declared before they can be used.

Also, you cannot change the data type of a variable in Java within the same scope. So, you cannot do something like this.

```
int price = 80;
... .. ...
float price;
```

Java Primitive Data Types

In Java all variables must be declared before they can be used.

```
int price;
```

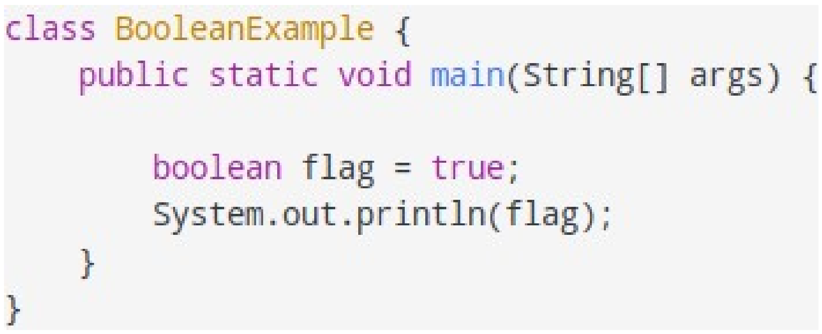
Here, *price* is a variable, and the data type of the variable is `int`. The *int* data type determines that the *price* variable can only contain integers.

In simple terms, a variable's data type determines the values a variable can store. There are 8 data types predefined in Java programming language, known as primitive data types.

Primitive Data Types

boolean

- The `boolean` data type has two possible values, either `true` or `false`.
- Default value: `false`.
- They are usually used for true/false conditions. For example,

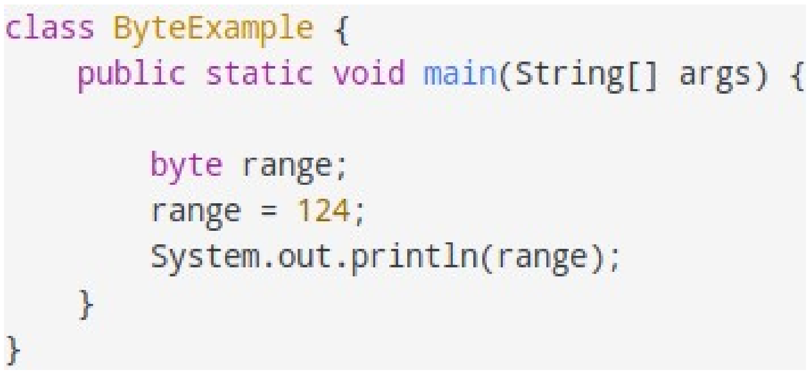


```
class BooleanExample {
    public static void main(String[] args) {

        boolean flag = true;
        System.out.println(flag);
    }
}
```

Output: true

byte

- The `byte` data type can have values from -128 to 127 (1 byte)
- It's used instead of `int` or other integer data types to save memory if it's certain that the value of a variable will be within [-128, 127].
- Default value: 0
- Example:

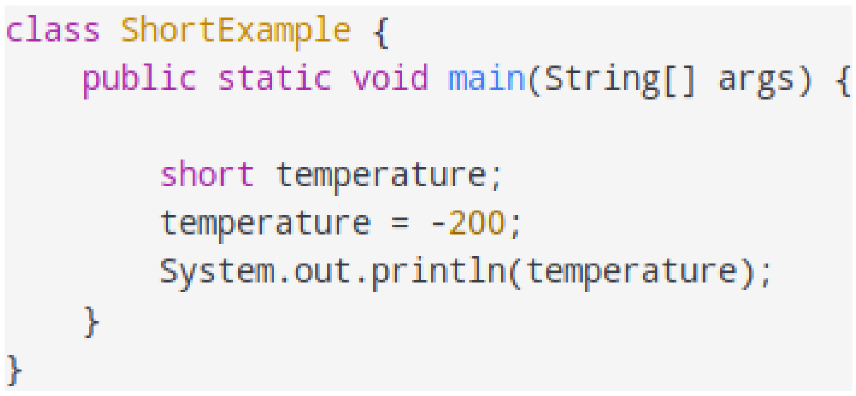


```
class ByteExample {
    public static void main(String[] args) {

        byte range;
        range = 124;
        System.out.println(range);
    }
}
```

Output: 124

short

- The `short` data type can have values from -32768 to 32767 (2 bytes)
- It's used instead of other integer data types to save memory if it's certain that the value of the variable will be within [-32768, 32767].
- Default value: 0
- Example:

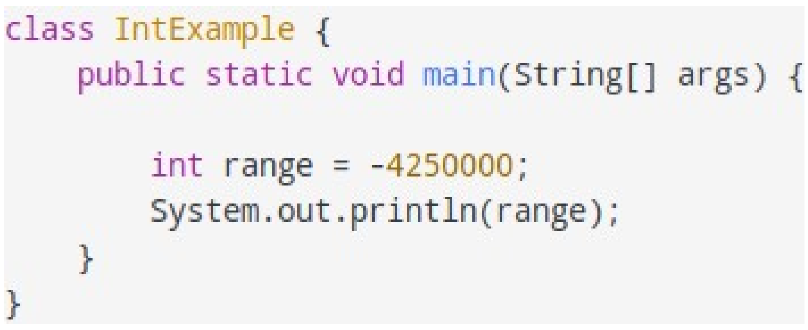


```
class ShortExample {
    public static void main(String[] args) {

        short temperature;
        temperature = -200;
        System.out.println(temperature);
    }
}
```

Output: -200

int

- The `int` data type can have values from -2,147,483,648 to 2,147,483,647 (4 bytes)
- Default value: 0
- Example:



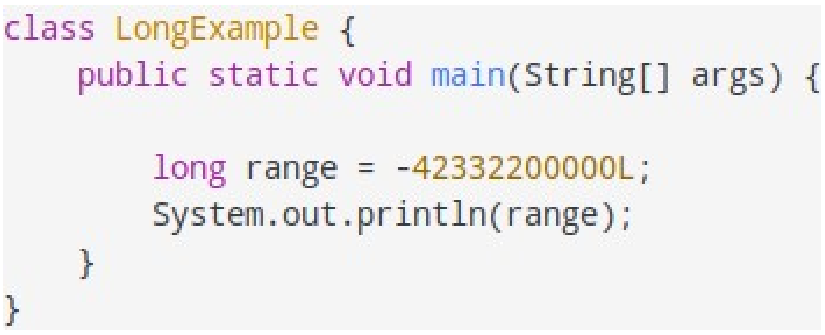
```
class IntExample {
    public static void main(String[] args) {

        int range = -4250000;
        System.out.println(range);
    }
}
```

Output: -4250000

long

- The `long` data type can have values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (8 bytes).
- Default value: 0

- Example:

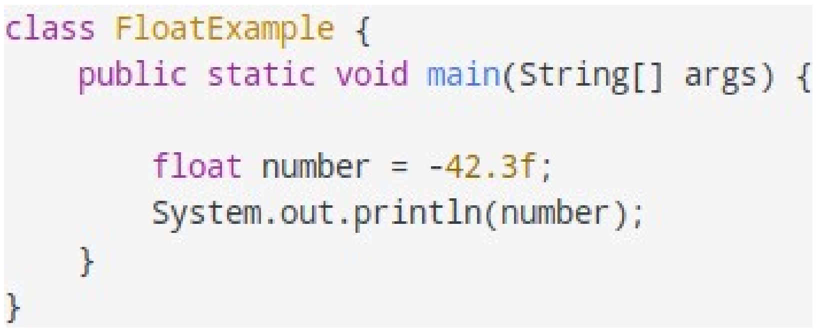


```
class LongExample {
    public static void main(String[] args) {

        long range = -42332200000L;
        System.out.println(range);
    }
}
```

Output: -42332200000

float

- Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits (4 bytes)
- Default value: 0.0 (0.0f)
- Example:



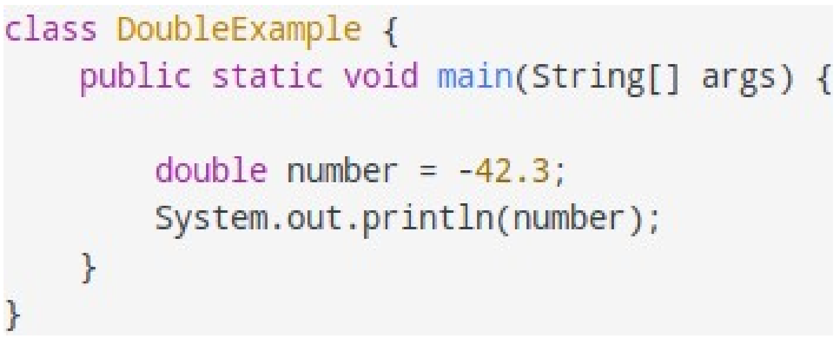
```
class FloatExample {
    public static void main(String[] args) {

        float number = -42.3f;
        System.out.println(number);
    }
}
```

Output: `-42.3`

Notice that, we have used `-42.3f` instead of `-42.3.` It's because `-42.3` is a `double` literal. To tell the compiler to treat `-42.3` as `float` rather than `double`, you need to use *f* or *F*.

double

- Stores fractional numbers. Sufficient for storing 15 decimal digits (8 bytes)
- Default value: 0.0 (0.0d)
- Example:

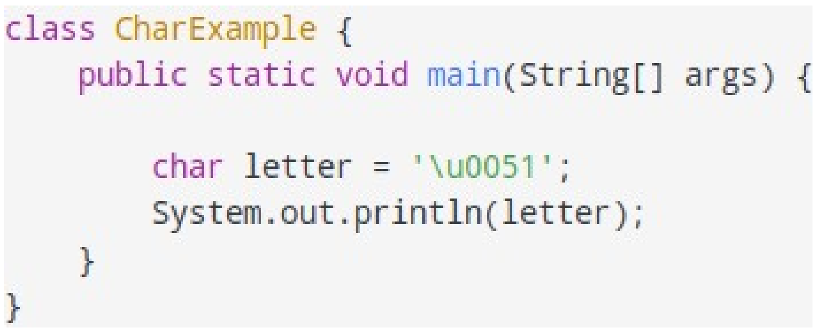


```
class DoubleExample {
    public static void main(String[] args) {

        double number = -42.3;
        System.out.println(number);
    }
}
```

Output: - 42.3

char

- It's a 16-bit Unicode character (2 bytes)
- The minimum value of the char data type is `'\u0000'` (0). The maximum value of the char data type is `'\uffff'`.
- Stores a single character/letter or ASCII values
- Example:

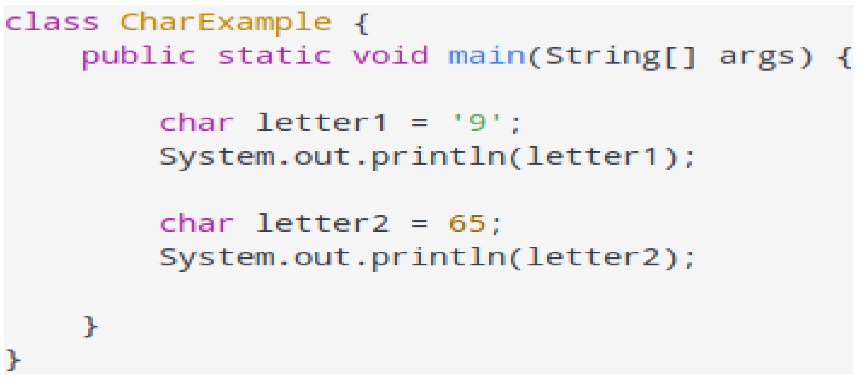


```
class CharExample {
    public static void main(String[] args) {

        char letter = '\u0051';
        System.out.println(letter);
    }
}
```

Output:`Q`

You get the output *Q* because the Unicode value of *Q* is `'\u0051'`.

Here is another example:



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

        char letter1 = '9';
        System.out.println(letter1);

        char letter2 = 65;
        System.out.println(letter2);

    }
}
```

Output:

```
9
A
```

When you print *letter1*, you will get *9* because *letter1* is assigned character `'9'.`

When you print *letter2*, you get *A* because the ASCII value of `'A'` is 65.

String
Java also provides support for character strings via `java.lang.String` class. Here's how you can create a String object in Java:

```
myString = "Programming is awesome";
```

5- Java Operators

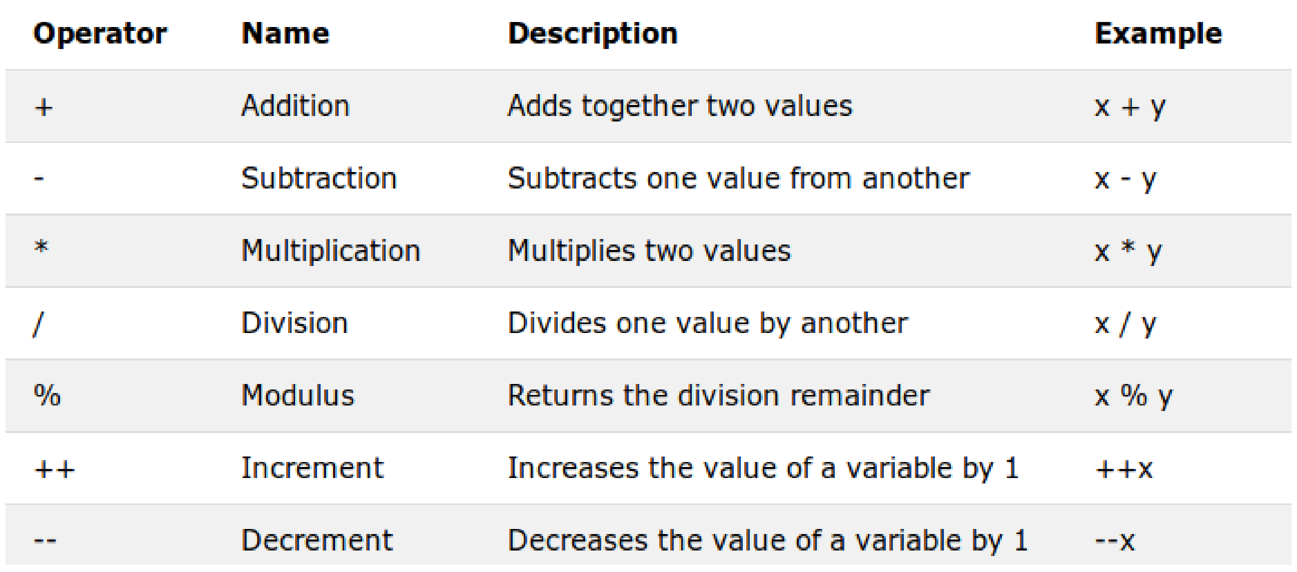
Operators are special symbols (characters) that carry out operations on operands (variables and values). For example, `+` is an operator that performs addition.

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Relational operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.



Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Java Assignment Operators

Assignment operators are used to assign values to variables.

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

Example

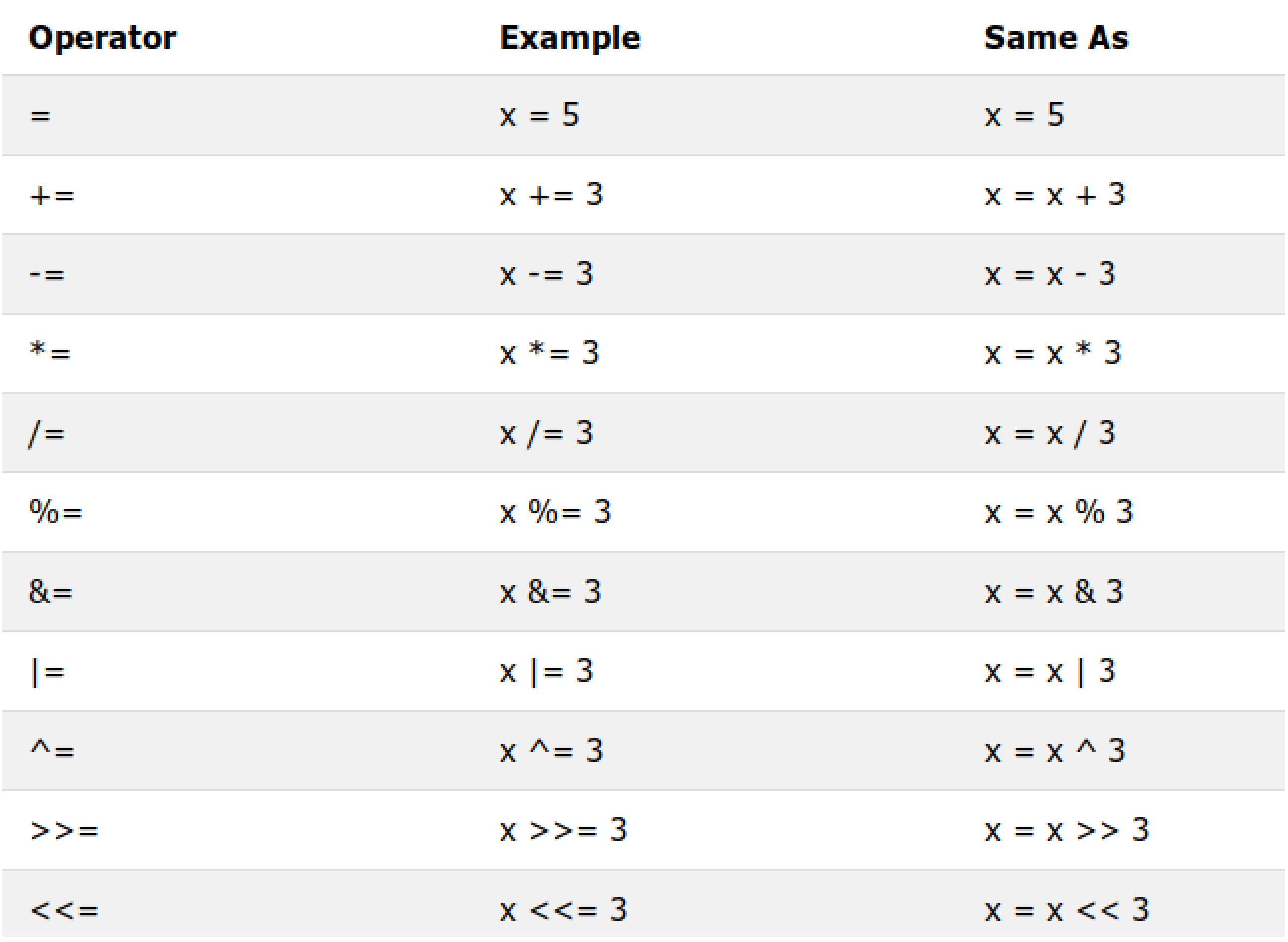
```
int x = 10;
```

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

Example

```
int x = 10;
x += 5;
```

A list of all assignment operators:



Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Java Relational (Comparison) Operators

Comparison operators are used to compare two values. It determines the relationship between the two operands. Depending on the relationship, it is evaluated to either `true` or `false`.

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

Java Logical Operators

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

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

6- Java Basic Input and Output

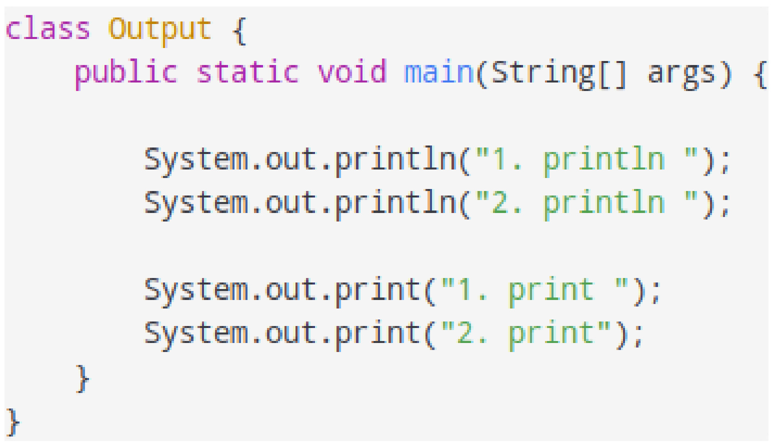
In Java, you can simply use

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

System.out.print(); or

System.out.printf();
```

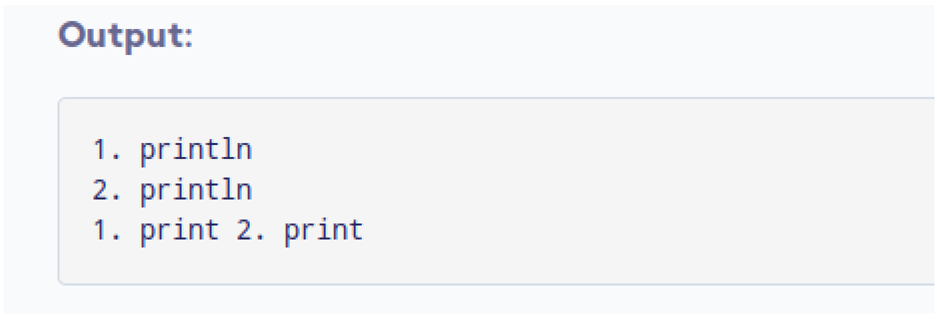
Difference between println(), print() and printf()

- `print()` - It prints string inside the quotes.
- `println()` - It prints string inside the quotes similar like `print()` method. Then the cursor moves to the beginning of the next line.
- `printf()` - Tt provides string formatting (similar to printf in C/C++ programming).

Example: print() and println()



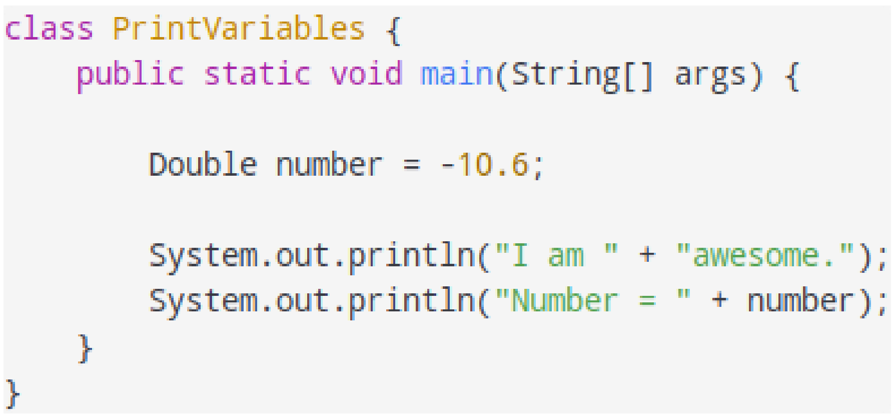
```
class Output {
    public static void main(String[] args) {

        System.out.println("1. println ");
        System.out.println("2. println ");

        System.out.print("1. print ");
        System.out.print("2. print");
    }
}
```



Output:

```
1. println
2. println
1. print 2. print
```

Example: Print Concatenated Strings



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

        Double number = -10.6;

        System.out.println("I am " + "awesome.");
        System.out.println("Number = " + number);
    }
}
```

Output:

```
I am awesome.
Number = -10.6
```

In the above example, notice the line,

```
System.out.println("I am " + "awesome.");
```

Here, we have used the `+` operator to concatenate (join) the two strings: *"I am "* and *"awesome."*.

And also, the line,

```
System.out.println("Number = " + number);
```

Here, first the value of variable *number* is evaluated. Then, the value is concatenated to the string: *"Number = "*

Java Input

in this tutorial, you will learn to get input from user using the object of `Scanner` class.

In order to use the object of `Scanner`, we need to import `java.util.Scanner` package.

```
import java.util.Scanner;
```

Then, we need to create an object of the `Scanner` class. We can use the object to take input from the user.

```java
// create an object of Scanner
Scanner input = new Scanner(System.in);

// take input from the user
int number = input.nextInt();
```

Example: Get Integer Input From the User

```java
import java.util.Scanner;

class Input {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int number = input.nextInt();
        System.out.println("You entered " + number);

        // closing the scanner object
        input.close();
    }
}
```

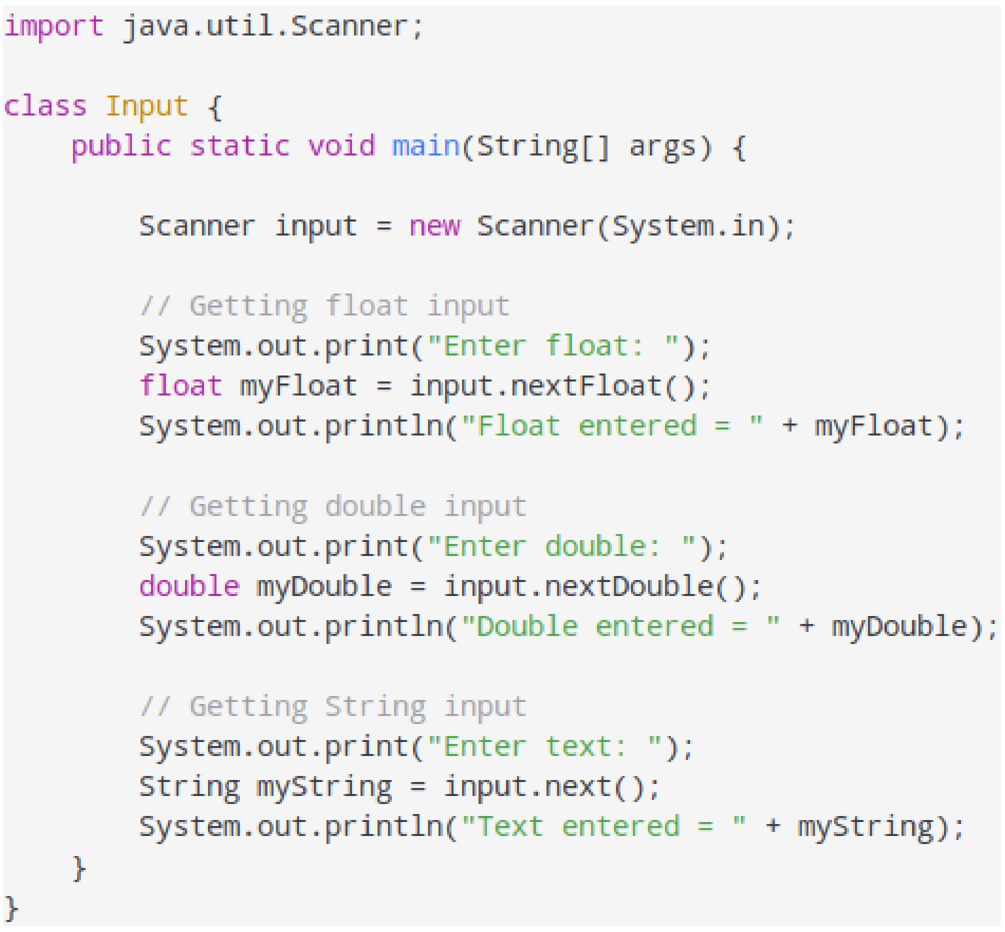
Output:

```
Enter an integer: 23
You entered 23
```

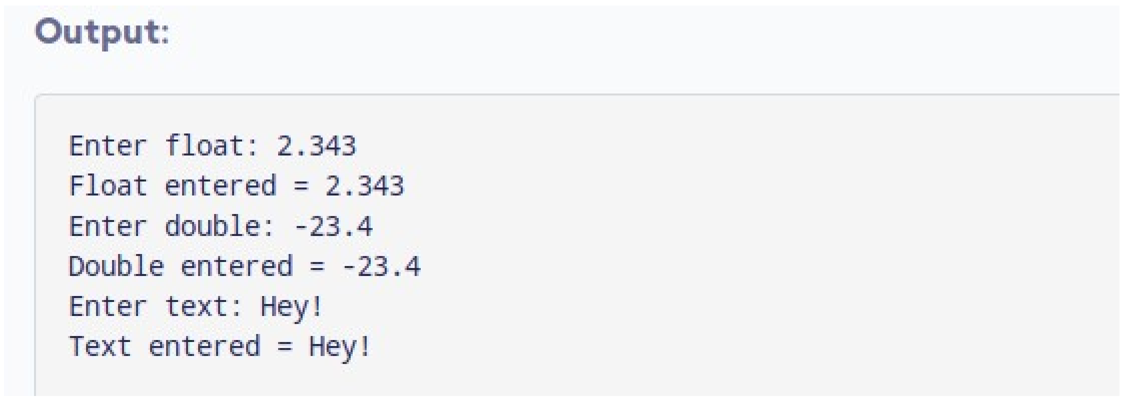
In the above example, we have created an object named *input* of the `Scanner` class. We then call the `nextInt()` method of the `Scanner` class to get an integer input from the user.

Similarly, we can use `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()` methods to get `long`, `float`, `double`, and `string` input respectively from the user.

Example: Get float, double and String Input



```java
import java.util.Scanner;

class Input {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        // Getting float input
        System.out.print("Enter float: ");
        float myFloat = input.nextFloat();
        System.out.println("Float entered = " + myFloat);

        // Getting double input
        System.out.print("Enter double: ");
        double myDouble = input.nextDouble();
        System.out.println("Double entered = " + myDouble);

        // Getting String input
        System.out.print("Enter text: ");
        String myString = input.next();
        System.out.println("Text entered = " + myString);
    }
}
```



Output:

```
Enter float: 2.343
Float entered = 2.343
Enter double: -23.4
Double entered = -23.4
Enter text: Hey!
Text entered = Hey!
```

7- Java Strings

Strings are used for storing text.

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

Example

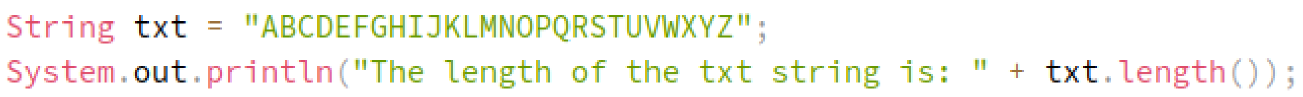
Create a variable of type `String` and assign it a value:

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

String Length

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

Example



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

More String Methods

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

Example

```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world"
```

Finding a Character in a String

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

Example

```
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

String Concatenation

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

Example

```
String firstName = "John";
String lastName = "Doe";
System.out.println(firstName + " " + lastName);
```

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

Example

```
String firstName = "John ";
String lastName = "Doe";
System.out.println(firstName.concat(lastName));
```

Special Characters

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

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

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

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

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

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

Example

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

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

Example

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

he sequence `\\` inserts a single backslash in a string:

Example

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

Adding Numbers and Strings

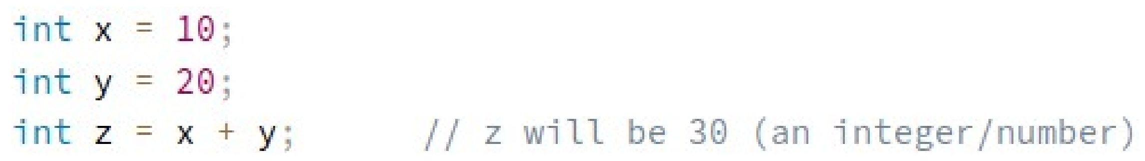
WARNING!

Java uses the + operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

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

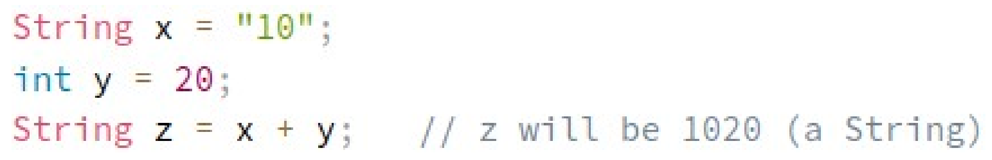
Example



```
int x = 10;
int y = 20;
int z = x + y;      // z will be 30 (an integer/number)
```

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

Example



```
String x = "10";
int y = 20;
String z = x + y;   // z will be 1020 (a String)
```

8- Java Comments

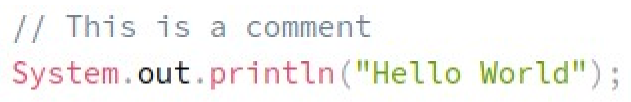
Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

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

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

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

Example



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

Java Multi-line Comments

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

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

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

Example

```
/* The code below will print the words Hello World
to the screen, and it is amazing */
System.out.println("Hello World");
```

9- Java if, if...else Statement

The if Statement

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

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

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

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

Example

```
if (20 > 18) {
  System.out.println("20 is greater than 18");
}
```

We can also test variables:

Example

```
int x = 20;
int y = 18;
if (x > y) {
  System.out.println("x is greater than y");
}
```

The else Statement

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

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Example

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

The else if Statement

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

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

Example

```
int time = 22;
if (time < 10) {
  System.out.println("Good morning.");
} else if (time < 20) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

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

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

Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

Syntax

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

Instead of writing:

Example

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

You can simply write:

Example

```
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

10- Java For Loop

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

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

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

Statement 2 defines the condition for executing the code block.

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

The example below will print the numbers 0 to 4:

Example

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

Example explained

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

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

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

Another Example

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

Example

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

For-Each Loop

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

Syntax

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

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

Example

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

Output:

```
Volvo
BMW
Ford
Mazda
```

11- Java While Loop

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

Syntax

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

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

Example

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

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

The Do/While Loop

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

Syntax

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

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

Example

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

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

JAVA LANGUAGE FUNDAMENTALS / PART 2

By

Dr. Amal S. Ajrash

1. Java Arrays

An array is a collection of similar types of data. It is a container that holds data (values) of one single type. For example, you can create an array that can hold 100 values of `int` type.
In Java, arrays are a fundamental construct that allows you to store and access a large number of values conveniently.

How to declare an array?

In Java, here is how we can declare an array.

```
dataType[] arrayName;
```

- `dataType` - it can be primitive data types like `int`, `char`, `double`, `byte`, etc. or Java objects
- `arrayName` - it is an identifier

Let's take an example,

```
double[] data;
```

Here, `data` is an array that can hold values of type `double`.

But, how many elements can array this hold?

We have to allocate memory for the array. The memory will define the number of elements that the array can hold.

```
data = new Double[10];
```

Here, the size of the array is 10. This means it can hold 10 elements (10 `double` types values). The size of an array is also known as the length of an array.

Note: Once the length of the array is defined, it cannot be changed in the program.

Let's take another example:

```
int[] age;
age = new int[5];
```

Here, `age` is an array. It can hold 5 values of `int` type.

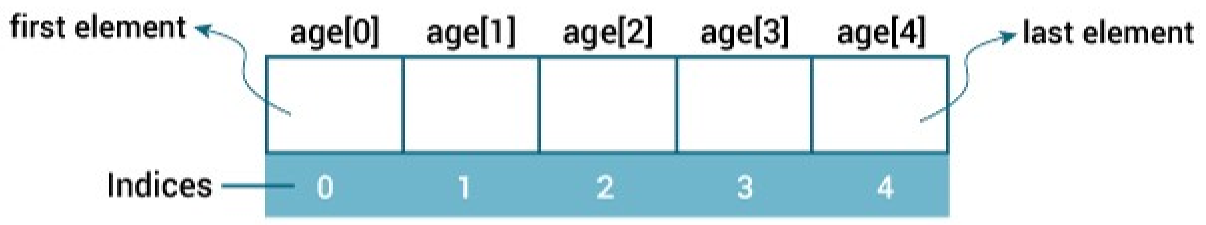
In Java, we can declare and allocate memory of an array in one single statement. For example,

```
int[] age = new int[5];
```

Java Array Index

In Java, each element in an array is associated with a number. The number is known as an array index. We can access elements of an array by using those indices. For example,

```
int[] age = new int[5];
```



Array age **of length 5**

Here, we have an array of length 5. In the image, we can see that each element consists of a number (array index). The array indices always start from 0.

Now, we can use the index number to access elements of the array. For example, to access the first element of the array is we can use `age[0]`, and the second element is accessed using `age[1]` and so on.

Note: If the length of an array is `n`, the first element of the array will be arrayName[0] and the last element will be `arrayName[n-1]`.

If we did not store any value to an array, the array will store some default value (`0` for `int` type and `false` for `boolean` type) by itself. For example,

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

        // create an array of length 5
        int[] age = new int[5];

        // access each element of the array using the index number
        System.out.println(age[0]);
        System.out.println(age[1]);
        System.out.println(age[2]);
        System.out.println(age[3]);
        System.out.println(age[4]);
    }
}
```

Output:

```
0
0
0
0
0
```

In the above example, we have created an array named age. However, we did not assign any values to the array. Hence when we access the individual elements of the array, the default values are printed to the screen.

Here, we are individually accessing the elements of the array. There is a better way to access elements of the array using a loop (generally for-loop). For example,

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

        // create an array of length 5
        int[] age = new int[5];
        // access elements using of the array
        for (int i = 0; i < 5; ++i) {
            System.out.println(age[i]);
        }
    }
}
```

Output:

```
0
0
0
0
```

```
0
```

How to initialize arrays in Java?

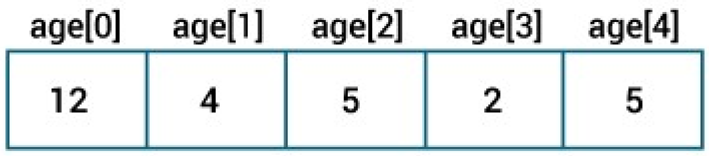
In Java, we can initialize arrays during declaration or you can initialize later in the program as per your requirement.

Initialize an Array During Declaration

Here's how you can initialize an array during declaration.

```
int[] age = {12, 4, 5, 2, 5};
```

This statement creates an array named `age` and initializes it with the value provided in the curly brackets.
The length of the array is determined by the number of values provided inside the curly braces separated by commas. In our example, the length of `age` is 5.



Let's write a simple program to print elements of an array.

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

    // create an array
    int[] age = {12, 4, 5, 2, 5};

    // access elements of tha arau
    for (int i = 0; i < 5; ++i) {
      System.out.println("Element at index " + i +": " + age[i]);
    }
  }
}
```

Output:

```
Element at index 0: 12
Element at index 1: 4
Element at index 2: 5
Element at index 3: 2
Element at index 4: 5
```

How to access array elements?

We can easily access and alter elements of an array by using its numeric index. For example,

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

        int[] age = new int[5];

        // insert 14 to third element
        age[2] = 14;

        // insert 34 to first element
        age[0] = 34;

        for (int i = 0; i < 5; ++i) {
            System.out.println("Element at index " + i +": " + age[i]);

        }
    }
}
```

Output:

```
Element at index 0: 34
Element at index 1: 0
Element at index 2: 14
Element at index 3: 0
Element at index 4: 0
```

Example: Java arrays

The program below computes sum and average of values stored in an array of type `int`.

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

        int[] numbers = {2, -9, 0, 5, 12, -25, 22, 9, 8, 12};
        int sum = 0;
        Double average;

        // for each loop is used to access elements
        for (int number: numbers) {
            sum += number;
        }

        int arrayLength = numbers.length;

        // Change sum and arrayLength to double as average is in double
        average =  ((double)sum / (double)arrayLength);

        System.out.println("Sum = " + sum);
        System.out.println("Average = " + average);
    }
}
```

Output:

```
Sum = 36
Average = 3.6
```

In the above example, we have created an array of named `numbers`. We have used the `for...each` loop to access each element of the array. To learn more about `for...each` loop. Inside the loop, we are calculating the sum of each element. Notice the line,

```
int arrayLength = number.length;
```

Here, we are using the length attribute of the array to calculate the size of the array. We then calculate the average using:

```
average = ((double)sum / (double)arrayLength);
```

As you can see, we are converting the `int` value into `double`. This is called type casting in Java.

Multidimensional Arrays

Arrays we have mentioned till now are called one-dimensional arrays. However, we can declare multidimensional arrays in Java.

A multidimensional array is an array of arrays. That is, each element of a multidimensional array is an array itself. For example,

```
double[][] matrix = {{1.2, 4.3, 4.0},{4.1, -1.1}};
```

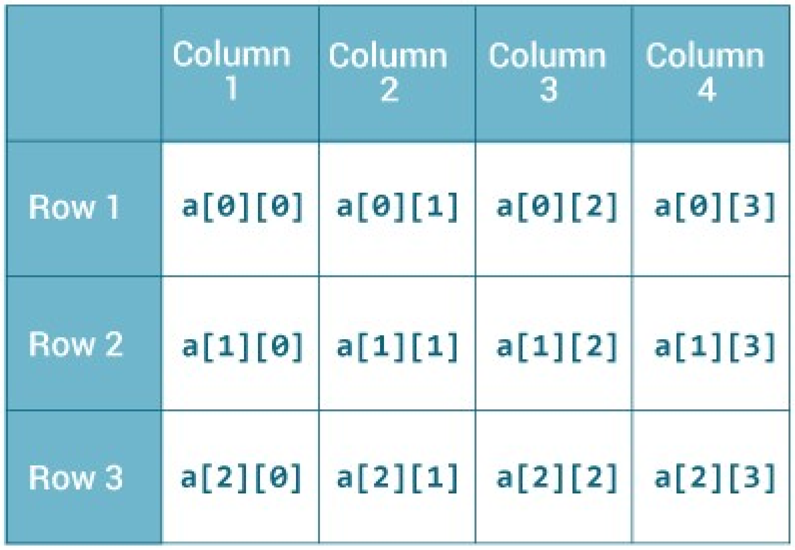
Here, we have created a multidimensional array named matrix. It is a 2-dimensional array.

2.Java Multidimensional Arrays

The Java multidimensional array using 2-dimensional arrays and 3-dimensional arrays. A multidimensional array is an array of arrays. Each element of a multidimensional array is an array itself. For example,

```
int[][] a = new int[3][4];
```

Here, we have created a multidimensional array named a. It is a 2-dimensional array, that can hold a maximum of 12 elements,



	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Remember, Java uses zero-based indexing, that is, indexing of arrays in Java starts with 0 and not 1. Let's take another example of the multidimensional array. This time we will be creating a 3-dimensional array. For example,

```
String[][][] data = new String[3][4][2];
```

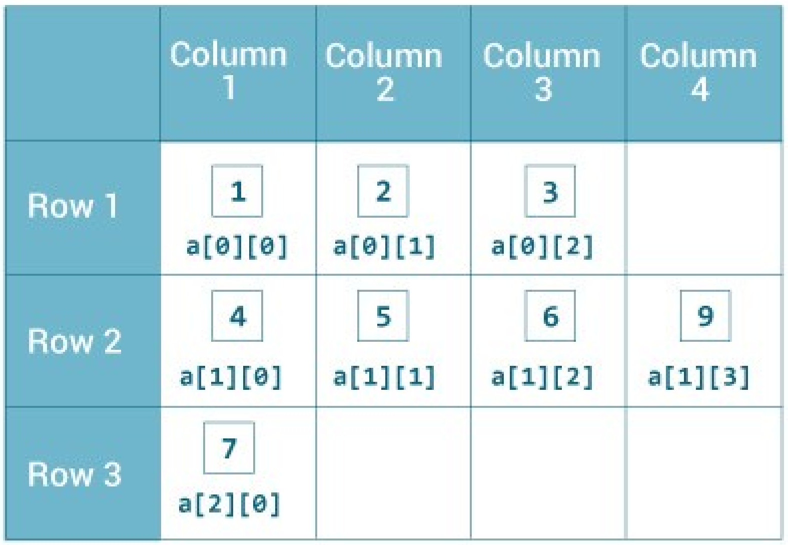
Here, `data` is a 3d array that can hold a maximum of 24 (3*4*2) elements of type `String`.

How to initialize a 2d array in Java?

Here is how we can initialize a 2-dimensional array in Java.

```
int[][] a = {
      {1, 2, 3},
      {4, 5, 6, 9},
      {7},
};
```

As we can see, each element of the multidimensional array is an array itself. And also, unlike C/C++, each row of the multidimensional array in Java can be of different lengths.



	Column 1	Column 2	Column 3	Column 4
Row 1	1 a[0][0]	2 a[0][1]	3 a[0][2]	
Row 2	4 a[1][0]	5 a[1][1]	6 a[1][2]	9 a[1][3]
Row 3	7 a[2][0]			

Example: 2-dimensional Array

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

        // create a 2d array
        int[][] a = {
            {1, 2, 3},
            {4, 5, 6, 9},
            {7},
        };

        // calculate the length of each row
        System.out.println("Length of row 1: " + a[0].length);
        System.out.println("Length of row 2: " + a[1].length);
        System.out.println("Length of row 3: " + a[2].length);
    }
}
```

Output:

```
Length of row 1: 3
Length of row 2: 4
Length of row 3: 1
```

In the above example, we are creating a multidimensional array named `a`. Since each component of a multidimensional array is also an array (`a[0]`, `a[1]` and `a[2]` are also arrays). Here, we are using the `length` attribute to calculate the length of each row.

Example: Print all elements of 2d array Using Loop

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

    int[][] a = {
        {1, -2, 3},
        {-4, -5, 6, 9},
        {7},
    };

    for (int i = 0; i < a.length; ++i) {
        for(int j = 0; j < a[i].length; ++j) {
            System.out.println(a[i][j]);
        }
    }
  }
}
```

Output:

```
1
-2
```

```
3
-4
-5
6
9
7
```

We can also use the for...each loop to access elements of the multidimensional array. For example,

```
class MultidimensionalArray {
    public static void main(String[] args) {
        // create a 2d array
        int[][] a = {
            {1, -2, 3},
            {-4, -5, 6, 9},
            {7},
        };

        // first for...each loop access the individual array
        // inside the 2d array
        for (int[] innerArray: a) {
            // second for...each loop access each element inside the row
            for(int data: innerArray) {
                System.out.println(data);
            }
        }
    }
}
```

Output:

```
1

-2

3

-4

-5

6
```

```
9

7
```

In the above example, we are have created a 2d array named `a`. We then used `for` loop and `for...each` loop to access each element of the array.

3.Java Class and Objects

Java is an object-oriented programming language. It is based on the concept of objects. These objects share two characteristics:

- state (fields)
- behavior (methods)

For example,

1. `Lamp` is an object
   State: `on` or `off`
   Behavior: `turn on` or `turn off`
2. `Bicycle` is an object
   States: `current gear`, `two wheels`, `number of gear`, etc
   Behavior: `braking`, `accelerating`, `changing gears`, etc

Principles of Object-oriented Programming:

- Encapsulation
- Inheritance
- Polymorphism

The focus of object-oriented programming is to break a complex programming task into objects that contain fields (to store data) and methods (to perform operations on fields).

Java Class

A class is a blueprint for the object. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.Since many houses can be made from the same description, we can create many objects from a class.

How to define a class in Java?

Here's how we can define a class in Java:

```
class ClassName {
   // variables
   // methods
}
```

For example,

```
class Lamp {

  // instance variable
  private boolean isOn;

  // method
  public void turnOn() {
    isOn = true;
  }

  // method
  public void turnOff() {
                isOn = false;
  }
}
```

Here, we have created a class named Lamp.

The class has one variable named `isOn` and two methods `turnOn()` and `turnOff()`. These variables and methods defined within a class are called **members** of the class.

In the above example, we have used keywords `private` and `public`. These are known as access modifiers.

Java Objects

An object is called an instance of a class. For example, suppose `Animal` is a class then `Cat`, `Dog`, `Horse`, and so on can be considered as objects of `Animal` class.
Here is how we can create objects in Java:

```
className object = new className();
```

Here, we are using the constructor `className()` to create the object. Constructors have the same name as the class and are similar to methods. Let's see how we can create objects of the `Lamp` class.

```
// l1 object
Lamp l1 = new Lamp();
// l2 object
Lamp l2 = new Lamp();
```

Here, we have created objects named `l1` and `l2` using the constructor of `Lamp` class (`Lamp()`).
Objects are used to access members of a class. Let's create objects of the Lamp class

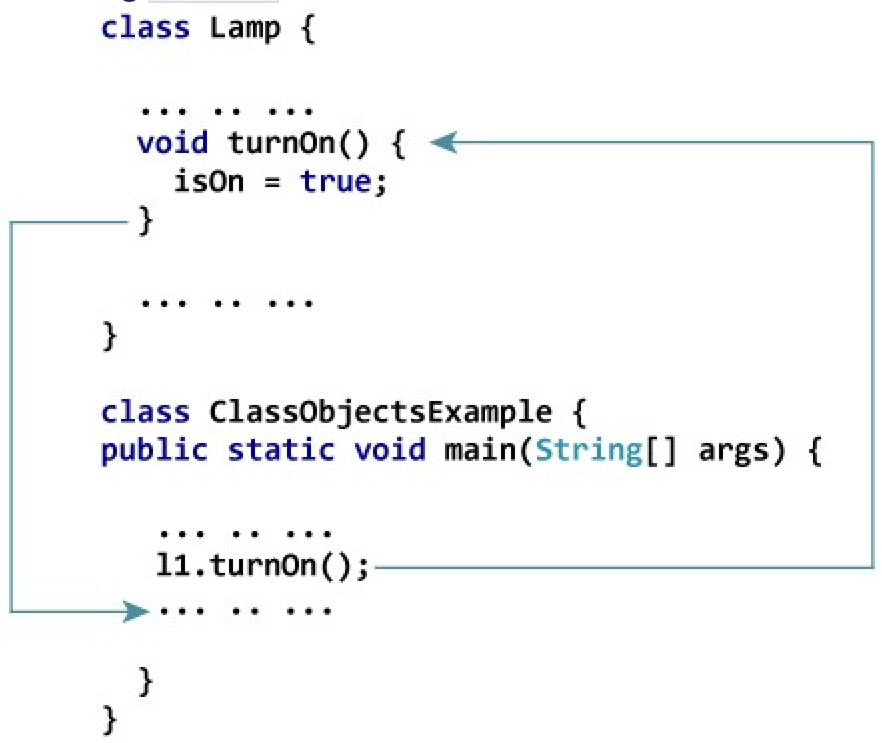
How to access members?

Objects are used to access members of the class. We can access members (call methods and access instance variables) by using the `.` operator. For example,

```
class Lamp {
   turnOn() {...}
}

// create object
Lamp l1 = new Lamp();

// access method turnOn()
l1.turnOn();
```

This statement calls the `turnOn()` method inside the Lamp class for the `l1` object.

When you call the method using the above statement, all statements within the body of the `turnOn()` method is executed. Then, the control of the program jumps back to the statement following `l1.turnOn()`;



```
class Lamp {

  ... .. ...
  void turnOn() {
    isOn = true;
  }

  ... .. ...
}

class ClassObjectsExample {
public static void main(String[] args) {

   ... .. ...
   l1.turnOn();
   ... .. ...

  }
}
```

Similarly, the instance variable can be accessed as:

```
class Lamp {
    boolean isOn;
}

// create object
Lamp l1 = new Lamp();

// access method turnOn()
l1.isOn = true;
```

Example: Java Class and Objects

```
class Lamp {
    boolean isOn;
```

```
  void turnOn() {
    // initialize variable with value true
    isOn = true;
    System.out.println("Light on? " + isOn);

  }

  void turnOff() {
    // initialize variable with value false
    isOn = false;
    System.out.println("Light on? " + isOn);
  }
}

class Main {
  public static void main(String[] args) {

    // create objects l1 and l2
    Lamp l1 = new Lamp();
    Lamp l2 = new Lamp();
    // call methods turnOn() and turnOff()
    l1.turnOn();
    l2.turnOff();
  }
}
```

Output:

```
Light on? true
Light on? false
```

In the above program,

1. We have created a class named `Lamp`.
2. The class has an instance variable `isOn` and two methods `turnOn()` and `turnOff()`.
3. Inside the `Main` class, we have created two objects `l1` and `l2` of the `Lamp` class.
4. We then use the `l1` object to call `turnOn()` and the `l2` object to call `turnOff()`:

```
l1.turnOn();
```

```
l2.turnOff();
```

5. The `turnOn()` method sets the `isOn` variable of `l1` object to `true`. and prints the output. Similarly, the `turnOff()` method sets the `isOn` variable of the `l2` object to `false` and prints the output.
   Note: The variables defined inside a class are known as instance variables for a reason. When an object is created, it is called an instance of the class.
   Each instance contains its own copy of the variables defined inside the class. Hence, known as instance variables. For example, the isOn variable is different for objects l1 and l2.

4.Java Methods

What is a method?

In mathematics, we might have studied about functions. For example, $f(x) = x^2$ is a function that returns a squared value of `x`.

```
If x = 2, then f(2) = 4
If x = 3, f(3) = 9
and so on.
```

Similarly, in computer programming, a function is a block of code that performs a specific task.

In object-oriented programming, the method is a jargon used for function. Methods are bound to a class and they define the behavior of a class.

Types of Java methods

Depending on whether a method is defined by the user, or available in the standard library, there are two types of methods in Java:

- Standard Library Methods
- User-defined Methods

Standard Library Methods

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.

For example,

- `print()` is a method of `java.io.PrintSteam`. The `print("...")` method prints the string inside quotation marks.
- `sqrt()` is a method of `Math` class. It returns the square root of a number.

Here's a working example:

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

    // using the sqrt() method
    System.out.print("Square root of 4 is: " + Math.sqrt(4));
  }
}
```

Output:

```
Square root of 4 is: 2.0
```

User-defined Method

We can also create methods of our own choice to perform some task. Such methods are called user-defined methods.

How to create a user-defined method?

Here is how we can create a method in Java:

```
public static void myMethod() {
  System.out.println("My Function called");
}
```

Here, we have created a method named `myMethod()`. We can see that we have used the `public`, `static` and `void` before the method name.

- `public` - access modifier. It means the method can be accessed from anywhere. `static` - It means that the method can be accessed without any objects.
- `void` - It means that the method does not return any value. This is a simple example of how we can create a method. However, the complete syntax of a method definition in Java is:

```
modifier static returnType nameOfMethod (parameters) {
    // method body
}
```

Here,

- **modifier** - It defines access types whether the method is public, private and so on.
- **static** - If we use the `static` keyword, it can be accessed without creating objects.

  For example, the `sqrt()` method of standard Math class is static. Hence, we can directly call `Math.sqrt()` without creating an instance of `Math` class.
- **returnType** - It specifies what type of value a method returns For example if a method has `int` return type then it returns an integer value.

  A method can return native data types (`int`, `float`, `double`, etc), native objects (`String`, `Map`, `List`, etc), or any other built-in and user-defined objects.

  If the method does not return a value, its return type is `void`.
- **nameOfMethod** - It is an identifier that is used to refer to the particular method in a program.

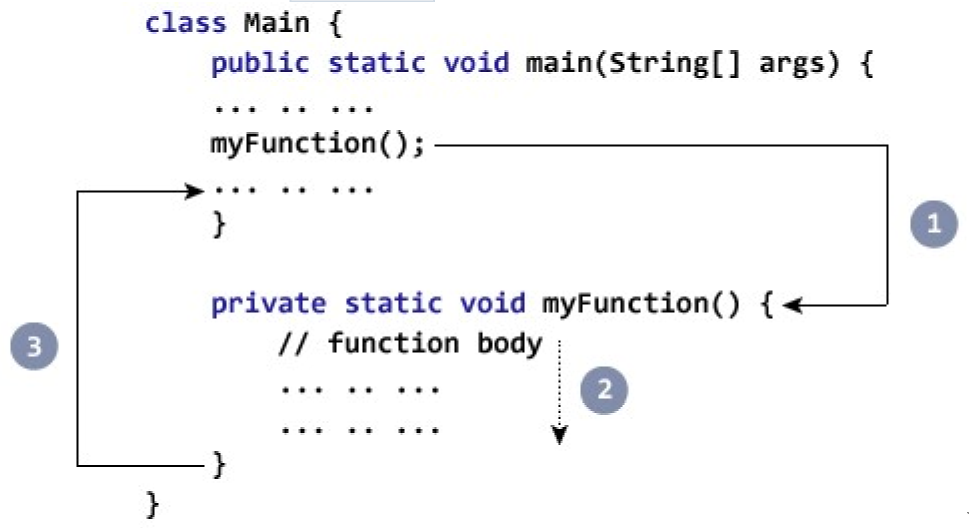
  We can give any name to a method. However, it is more conventional to name it after the tasks it performs. For example, `calculateArea()`, `display()`, and so on.
- **parameters (arguments)** - These are values passed to a method. We can pass any number of arguments to a method.
- **method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces `{ }`.

How to call a Java Method?

Now that we know how to define methods, we need to learn to use them. For that, we have to call the method. Here's how

```
myMethod();
```

This statement calls `myMethod()` method that was declared earlier.



```
class Main {
    public static void main(String[] args) {
    ... .. ...
    myFunction();
    ... .. ...
    }

    private static void myFunction() {
        // function body
        ... .. ...
        ... .. ...
    }
}
```

Working of the method call in Java

1. While executing the program code, it encounters `myFunction();` in the code.
2. The execution then branches to the `myFunction()` method and executes code inside the body of the method.
3. After the execution of the method body, the program returns to the original state and executes the next statement after the method call.

Example: Java Method

Let's see how we can use methods in a Java program.

```
class Main {

  public static void main(String[] args) {
      System.out.println("About to encounter a method.");

      // method call
      myMethod();

      System.out.println("Method was executed successfully!");
  }

  // method definition
  private static void myMethod(){
      System.out.println("Printing from inside myMethod()!");
  }
}
```

Output:

```
About to encounter a method.
Printing from inside myMethod().
Method was executed successfully!
```

In the above program, we have a method named `myMethod()`. The method doesn't accept any arguments. Also, the return type of the method is `void` (means doesn't return any value). Here, the method is `static`. Hence we have called the method without creating an object of the class. Let's see another example,

```
class Main {

    public static void main(String[] args) {

        // create object of the Output class
        Output obj = new Output();
        System.out.println("About to encounter a method.");

        // calling myMethod() of Output class
        obj.myMethod();

        System.out.println("Method was executed successfully!");
    }
}

class Output {

    // public: this method can be called from outside the class
    public void myMethod() {
        System.out.println("Printing from inside myMethod().");
    }
}
```

Output:

```
About to encounter a method.
Printing from inside myMethod().
Method was executed successfully!
```

In the above example, we have created a method named `myMethod()`. The method is inside a class named `Output`.

Since the method is not `static`, it is called using the object `obj` of the class.

```
obj.myMethod();
```

Method Arguments and Return Value

Java method can have zero or more parameters. And, it may also return some value.

Example: Return Value from Method

Let's take an example of a method returning a value.

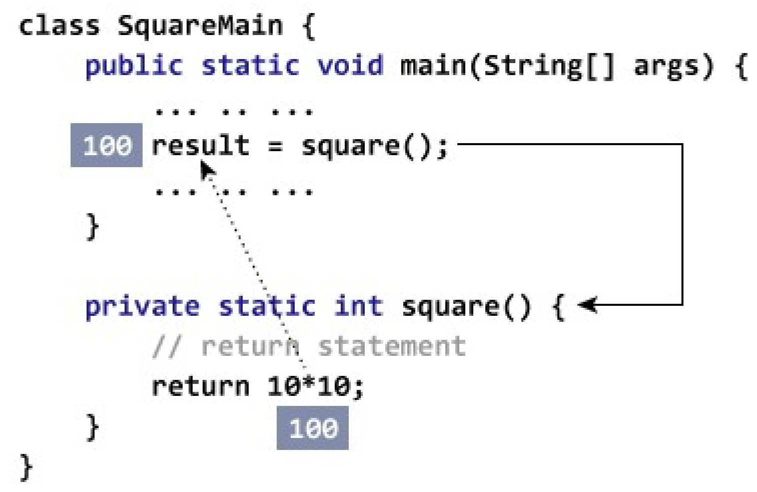
```
class SquareMain {
    public static void main(String[] args) {
        int result;
        // call th method and store returned value
        result = square();
        System.out.println("Squared value of 10 is: " + result);
    }

    public static int square() {
        // return statement
        return 10 * 10;
    }
}
```

Output:

```
Squared value of 10 is: 100
```

In the above program, we have created a method named `square()`. This method does not accept any arguments and returns value `10 *10`.

Here, we have mentioned the return type of the method as `int`. Hence, the method should always return an integer value.



```
class SquareMain {
    public static void main(String[] args) {
        ... .. ...
  100   result = square();
        ... .. ...
    }

    private static int square() {
        // return statement
        return 10*10;
    }          100
}
```

Representation of a method returning a value

As we can see, the scope of this method is limited as it always returns the same value. Now, let's modify the above code snippet so that instead of always returning the squared value of 10, it returns the squared value of any integer passed to the method.

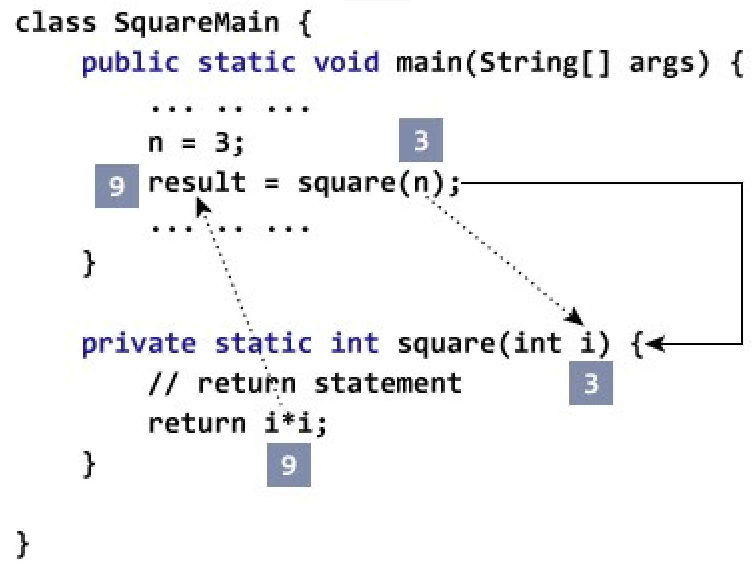
Example: Method Accepting Arguments and Returning Value

```
public class Main {

  public static void main(String[] args) {
    int result, n;

    n = 3;
    result = square(n);
    System.out.println("Square of 3 is: " + result);
    n = 4
    result = square(n);
    System.out.println("Square of 4 is: " + result);
  }

  // method
  static int square(int i) {
        return i * i;
    }
}
```

Output:

```
Squared value of 3 is: 9
Squared value of 4 is: 16
```

Here, the `square()` method accepts an argument `i` and returns the square of `i`. The returned value is stored in the variable `result`.



```
class SquareMain {
    public static void main(String[] args) {
        ... .. ...
        n = 3;
        result = square(n);
        ... .. ...
    }

    private static int square(int i) {
        // return statement
        return i*i;
    }

}
```

Passing arguments and returning a value from a method in Java

If we pass any other data type instead of int, the compiler will throw an error. It is because Java is a strongly typed language.

The argument `n` passed to the `getSquare()` method during the method call is called an actual argument.

```
result = getSquare(n);
```

The argument `i` accepted by the method definition is known as a formal argument. The type of formal argument must be explicitly typed.

```
public static int square (int i) {...}
```

We can also pass more than one argument to the Java method by using commas. For example,

```
public class Main {

    // method definition
    public static int getIntegerSum (int i, int j) {
        return i + j;
    }

    // method definition
    public static int multiplyInteger (int x, int y) {
        return x * y;
    }

    public static void main(String[] args) {

        // calling methods
        System.out.println("10 + 20 = " + getIntegerSum(10, 20));
        System.out.println("20 x 40 = " + multiplyInteger(20, 40));
    }
}
```

Output:

```
10 + 20 = 30
20 x 40 = 800
```

Note: The data type of actual and formal arguments should match, i.e., the data type of first actual argument should match the type of first formal argument. Similarly, the type of second actual argument must match the type of second formal argument and so on.

What are the advantages of using methods?

1. The main advantage is **code reusability**. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times". For example,

```
public class Main {

    // method defined
    private static int getSquare(int x){
        return x * x;
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {

            // method call
            int result = getSquare(i);
            System.out.println("Square of " + i + " is: " + result);
        }
    }
}
```

Output:

```
Square of 1 is: 1
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
Square of 5 is: 25
```

In the above program, we have created the method named getSquare() to calculate the square of a number. Here, the same method is used to calculate the square of numbers less than 6.

Hence, we use the same method again and again.

2. Methods make code more **readable and easier** to debug. For example, `getSquare()` method is so readable, that we can know what this method will be calculating the square of a number.

5.Java String

In Java, a string is a sequence of characters. For example, `"hello"` is a string containing a sequence of characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`.

Unlike other programming languages, strings in Java are not primitive types (like `int`, `char`, etc). Instead, all strings are objects of a predefined class named `String`. For example,

```
// create a string
String type = "java programming";
```

Here, we have created a string named `type`. Here, we have initialized the string with `"java programming"`. In Java, we use **double quotes** to represent a string.

The string is an instance of the `String` class.

Note: All string variables are instances of the `String` class.

Java String Methods

Java `String` provides various methods that allow us to perform different string operations. Here are some of the commonly used string methods.

Methods	Description
`concat()`	joins the two strings together
`equals()`	compares the value of two strings
`charAt()`	returns the character present in the specified location
`getBytes()`	converts the string to an array of bytes
`indexOf()`	returns the position of the specified character in the string
`length()`	returns the size of the specified string
`replace()`	replaces the specified old character with the specified new character
`substring()`	returns the substring of the string
`split()`	breaks the string into an array of strings
`toLowerCase()`	converts the string to lowercase
`toUpperCase()`	converts the string to uppercase
`valueOf()`	returns the string representation of the specified data

Let's take a few examples.

Example 1: Java find string's length

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

    // create a string
    String greet = "Hello! World";
    System.out.println("The string is: " + greet);

    //checks the string length
    System.out.println("The length of the string: " + greet.length());
  }
}
```

Output

```
The string is: Hello! World
The length of the string: 12
```

In the above example, we have created a string named `greet`. Here we have used the `length()` method to get the size of the string.

Example 2: Java join two strings using concat()

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

    // create string
    String greet = "Hello! ";
    System.out.println("First String: " + greet);

    String name = "World";
    System.out.println("Second String: " + name);

    // join two strings
    String joinedString = greet.concat(name);
    System.out.println("Joined String: " + joinedString);
  }
}
```

Output

```
First String: Hello!
Second String: World
Joined String: Hello! World
```

In the above example, we have created 2 strings named `greet` and `name`.

Here, we have used the `concat()` method to join the strings. Hence, we get a new string named `joinedString`.

In Java, we can also join two strings using the `+` operator.

Example 3: Java join strings using + operator

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

    // create string
    String greet = "Hello! ";
    System.out.println("First String: " + greet);

    String name = "World";
    System.out.println("Second String: " + name);

    // join two strings
    String joinedString = greet + name;
    System.out.println("Joined String: " + joinedString);
  }
}
```

Output

```
First String: Hello!
Second String: World
Joined String: Hello! World
```

Here, we have used the `+` operator to join the two strings.

Example 4: Java compare two strings

```
class Main {
  public static void main(String[] args) {
    // create strings
    String first = "java programming";
    String second = "java programming";
    String third = "python programming";

    // compare first and second strings
    boolean result1 = first.equals(second);
    System.out.println("Strings first and second are equal: " + result1);

    //compare first and third strings
    boolean result2 = first.equals(third);
    System.out.println("Strings first and third are equal: " + result2);
  }
}
```

Output

```
Strings first and second are equal: true
Strings first and third are equal: false
```

In the above example, we have used the `equals()` method to compare the value of two strings. The method returns `true` if both strings are the same otherwise it returns `false`.

Note: We can also use the `==` operator and `compareTo()` method to make a comparison between 2 strings.

Example 5: Java get characters from a string

```
class Main {
  public static void main(String[] args) {
    // create string using the string literal
    String greet = "Hello! World";
    System.out.println("The string is: " + greet);
    // returns the character at 3
    System.out.println("The character at : " + greet.charAt(3));
    // returns the character at 7
    System.out.println("The character at 7: " + greet.charAt(7));
  }
}
```

Output

```
The string is: Hello! World
The character at 3: l
The character at 7: W
```

In the above example, we have used the `charAt()` method to access the character from the specified position.

Example 6: Java Strings other methods

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

    // create string using the new keyword
    String example = new String("Hello! World");

    // returns the substring World
    System.out.println("Using the subString(): " + example.substring(7));

    // converts the string to lowercase
    System.out.println("Using the toLowerCase(): " + example.toLowerCase());

    // converts the string to uppercase
    System.out.println("Using the toUpperCase(): " + example.toUpperCase());

    // replaces the character '!' with 'o'
    System.out.println("Using the replace(): " + example.replace('!', 'o'));
  }
}
```

Output

```
Using the subString(): World
Using the toLowerCase(): hello! world
Using the toUpperCase(): HELLO! WORLD
Using the replace(): Helloo World
```

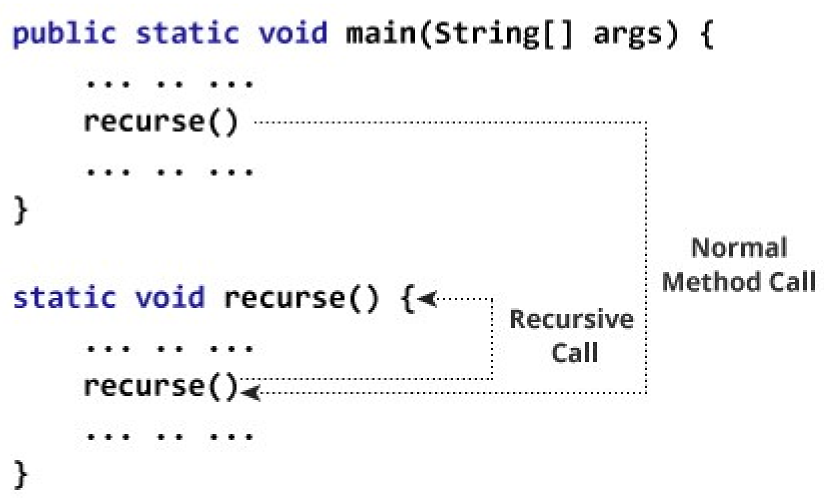
In the above example, we have created a string named `example` using the `new` keyword. Here,

- the `substring()` method returns the string `World`
- the `toLowerCase()` method converts the string to the lower case
- the `toUpperCase()` method converts the string to the upper case
- the `replace()` method replaces the character `'!'` with `'o'`.

6.Java Recursion

In Java, a method that calls itself is known as a recursive method. And, this process is known as recursion. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

How Recursion works?



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

static void recurse() {
    ... .. ...
    recurse()
    ... .. ...
}
```

Working of Java Recursion

In the above example, we have called the `recurse()` method from inside the `main` method. (normal method call). And, inside the recurse() method, we are again calling the same recurse method. This is a recursive call. In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely. Hence, we use the if...else statement (or similar approach) to terminate the recursive call inside the method.

Example: Factorial of a Number Using Recursion

```
class Factorial {

    static int factorial( int n ) {
        if (n != 0)  // termination condition
            return n * factorial(n-1); // recursive call
        else
            return 1;
    }

    public static void main(String[] args) {
        int number = 4, result;
        result = factorial(number);
        System.out.println(number + " factorial = " + result);
    }
}
```

Output:

```
4 factorial = 24
```

In the above example, we have a method named `factorial()`. The `factorial()` is called from the `main()` method. with the `number` variable passed as an argument.

Here, notice the statement,

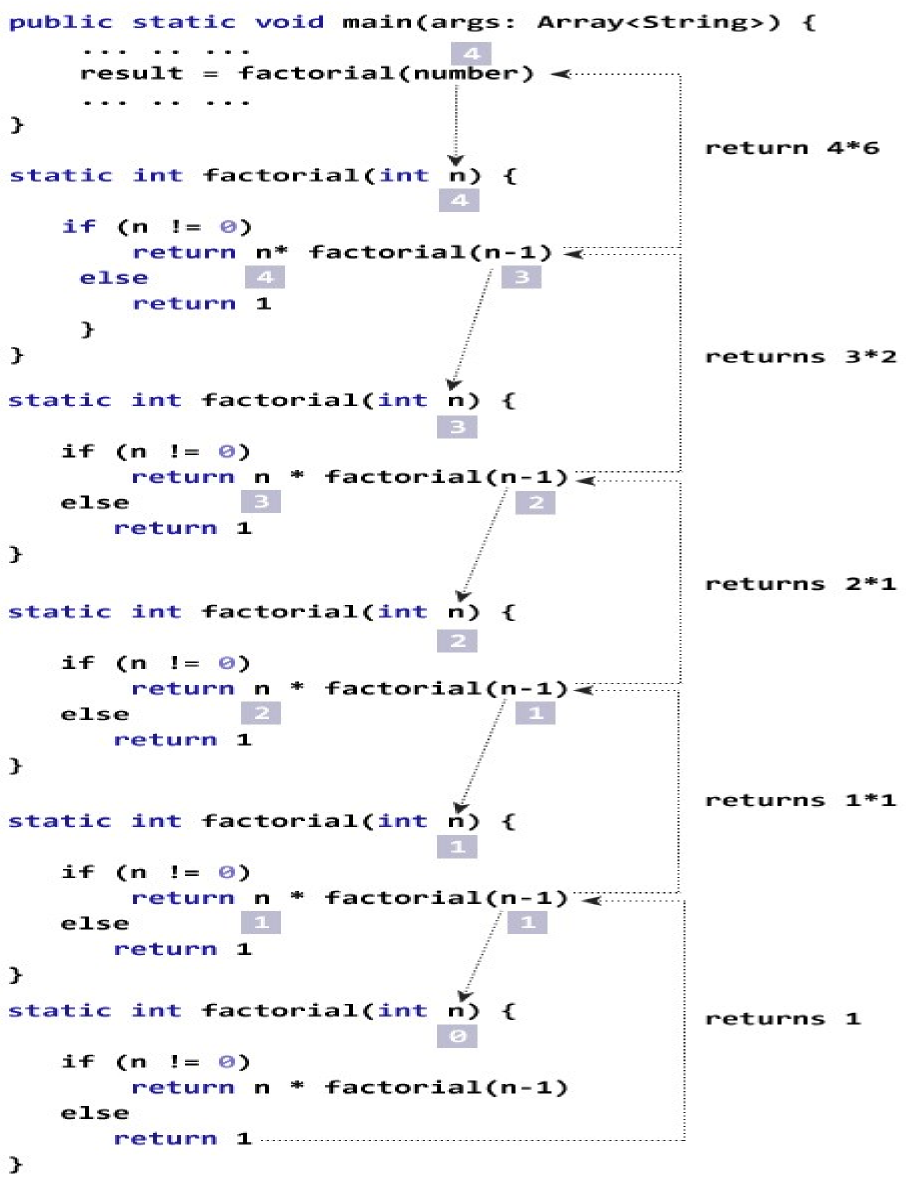
```
return n * factorial(n-1);
```

The `factorial()` method is calling itself. Initially, the value of n is 4 inside `factorial()`. During the next recursive call, 3 is passed to the `factorial()` method. This process continues until `n` is equal to 0.

When `n` is equal to 0, the `if` statement returns false hence 1 is returned. Finally, the accumulated result is passed to the `main()` method.

Working of Factorial Program

The image below will give you a better idea of how the factorial program is executed using recursion.



```
public static void main(args: Array<String>) {
     ... .. ...
     result = factorial(number)
     ... .. ...
}

static int factorial(int n) {

   if (n != 0)
       return n* factorial(n-1)
    else
       return 1
    }
}

static int factorial(int n) {

   if (n != 0)
       return n * factorial(n-1)
   else
      return 1
}

static int factorial(int n) {

   if (n != 0)
       return n * factorial(n-1)
   else
      return 1
}

static int factorial(int n) {

   if (n != 0)
       return n * factorial(n-1)
   else
      return 1
}
static int factorial(int n) {

   if (n != 0)
       return n * factorial(n-1)
   else
      return 1
}
```

Recursion

Advantages and Disadvantages of Recursion

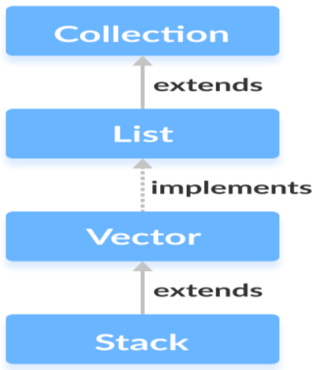
When a recursive call is made, new storage locations for variables are allocated on the stack. As, each recursive call returns, the old variables and parameters are removed from the stack. Hence, recursion generally uses more memory and is generally slow.

On the other hand, a recursive solution is much simpler and takes less time to write, debug and maintain.

7.Java Stack Class

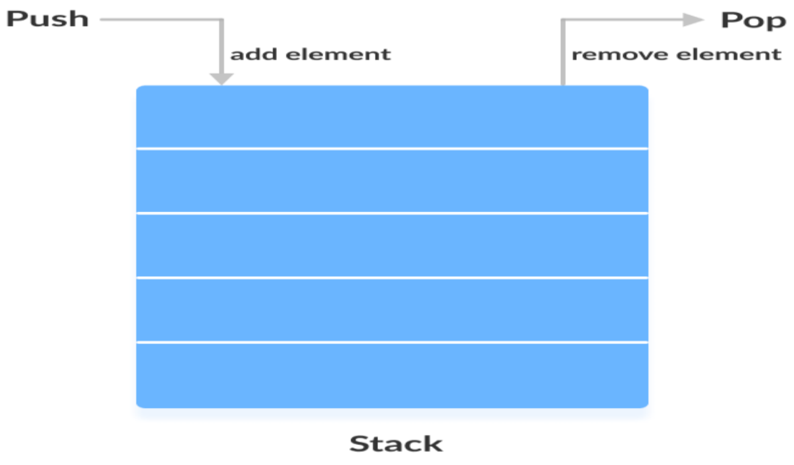
In this tutorial, we will learn about the Java Stack class and its methods with the help of examples.

The Java collections framework has a class named `Stack` that provides the functionality of the stack data structure.
The `Stack` class extends the `Vector` class.



Stack Implementation

In stack, elements are stored and accessed in **Last In First Out** manner. That is, elements are added to the top of the stack and removed from the top of the stack.



Creating a Stack

In order to create a stack, we must import the `java.util.Stack` package first. Once we import the package, here is how we can create a stack in Java.

```
Stack<Type> stacks = new Stack<>();
```

Here, `Type` indicates the stack's type. For example,

```
// Create Integer type stack
Stack<Integer> stacks = new Stack<>();

// Create String type stack
Stack<String> stacks = new Stack<>();
```

Stack Methods

Since `Stack` extends the `Vector` class, it inherits all the methods `Vector`. To learn about different `Vector` method.

Besides these methods, the `Stack` class includes 5 more methods that distinguish it from `Vector`.

push() Method

To add an element to the top of the stack, we use the `push()` method. For example,

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        StackString> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);
    }
}
```

Output

```
Stack: [Dog, Horse, Cat]
```

pop() Method

To remove an element from the top of the stack, we use the `pop()` method. For example,

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Initial Stack: " + animals);
        // Remove element stacks

        String element = animals.pop();

        System.out.println("Removed Element: " + element);
    }
}
```

Output

```
Initial Stack: [Dog, Horse, Cat]
Removed Element: Cat
```

peek() Method

The `peek()` method returns an object from the top of the stack. For example,

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Access element from the top
        String element = animals.peek();
        System.out.println("Element at top: " + element);

    }
}
```

Output

```
Stack: [Dog, Horse, Cat]
Element at top: Cat
```

search() Method

To search an element in the stack, we use the `search()` method. It returns the position of the element from the top of the stack. For example,

```
import java.util.Stack;
class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();
        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Search an element
        int position = animals.search("Horse");
        System.out.println("Position of Horse: " + position);
    }
}
```

Output

```
Stack: [Dog, Horse, Cat]
Position of Horse: 2
```

empty() Method To check whether a stack is empty or not, we use the `empty()` method. For example,

```
import java.util.Stack;
class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();
        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);
        // Check if stack s empty
        boolean result = animals.empty();
        System.out.println("Is the stack empty? " + result);
    }
}
```

Output

```
Stack: [Dog, Horse, Cat]
Is the stack empty? false
```