**What is Java?**

Java is a popular programming language, created in 1995.

**It is used for**

Mobile applications (specially Android apps)

Desktop applications

Web applications

Web servers and application servers

Games

Database connection

And much, much more!

**Why Use Java?**

Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)

It is one of the most popular programming language in the world

It 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

**Check the version of java:** java -version

install java then set the path

"Path" variable advance system settings->set path variable

**create the file name**

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

Main.java -> javac Main.java-> compiler the java file

java Main -> Output is displayed

Helloworld

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

**The main Method:**

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

public static void main(String[] args)

System.out.println

System(Class)-> it contains useful members

Out(one of the member of system class) short for output

Println(print line) used to print value on to the screen

*Println vs print*

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

**Java Comments**

 It can also be used to prevent execution when testing alternative code.

**SingleLine (//)**

**Multi-line Comments(/\* \*/)**

**Java Variables**

Variables are containers for storing data values.

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

Syntax: *type variableName = value;*

The **equal sign** is used to assign values to the variable.

Final Variable: final int myNum = 15;

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

int myNum = 5;

float myFloatNum = 5.99f;

char myLetter = 'D';

boolean myBool = true;

String myText = "Hello";

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

System.***out***.println("Hello"+ 10+20);

System.***out***.println(10+20+"Hello");

Output:

helloworld

Hello1020

30Hello

The general rules for naming variables are:

* Names can contain letters, digits, underscores, and dollar signs
* Names must begin with a letter
* Names should start with a lowercase letter and it cannot contain whitespace
* Names can also begin with $ and \_ (but we will not use it in this tutorial)
* Names are case sensitive ("myVar" and "myvar" are different variables)
* Reserved words (like Java keywords, such as int or boolean) cannot be used as names

**int** $=10;

System.***out***.println($);

Data types are divided into two groups:

* Primitive data types - includes byte, short, int, long, float, double, boolean and char
* Non-primitive data types - such as [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp)

There are eight primitive data types in Java:

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Description** |
| byte | 1 byte | Stores whole numbers from -128 to 127 |
| short | 2 bytes | Stores whole numbers from -32,768 to 32,767 |
| int | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes | Stores fractional numbers. Sufficient for storing 15 decimal digits |
| boolean | 1 bit | Stores true or false values |
| char | 2 bytes | Stores a single character/letter or ASCII values |

**Integer types**

byte, short, int and long

**Floating point types**

float and double.

Note that you should end the value with an "f".

Note that you should end the value with a "d".

**Scientific Number**

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

**double** var = 5e6d;

System.***out***.println(var);

Output:

5000000.0

**Java Characters**

The char data type is used to store a **single** character.

The character must be surrounded by single quotes, like 'A' or 'c'.

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

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

System.out.println(myVar1);

System.out.println(myVar2);

System.out.println(myVar3);

Output:

A

B

C

String

The String data type is used to store a sequence of characters (text).

String values must be surrounded by double quotes:

**Java Type Casting:**

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

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

int myInt = 9;

double myDouble = myInt;

9.0

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

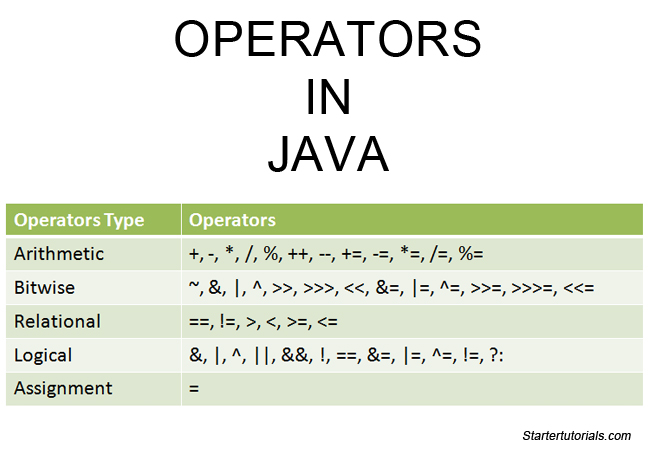
double myDouble = 9.78d;

int myInt = (int) myDouble;

9

**Java Operators**

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



**Java String**

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

String Length

txt.length(); txt.toUpperCase(); txt.toLowerCase(); txt.indexOf();

**Java Special Characters**

**backslash escape character (\) \’** \"\\

|  |  |  |
| --- | --- | --- |
| \n | New Line |  |
| \r | Carriage Return |  |
| \t | Tab |  |
| \b | Backspace |  |
| \f | Form Feed |  |

Java Math

Math.max(x,y);

Math.min(x,y);

Math.sqrt(x);

Math.abs(x);

Math.random();

returns a random number between 0.0 (inclusive), and 1.0 (exclusive)

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

Java Conditions and if statement

if.. else

if(condition){

//block of code

}

else if(condition){

//block of code

}

else{

//block of code

}

Ternary Opeartor:

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

Switch statements:

switch(*expression*) {

case x:

*// code block*

break;

case y:

*// code block*

break;

default:

*// code block*

}

**loops:**

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

**Java while loop:**while (*condition*) {

*// code block to be executed*

}

**Do while loop:**

do {

*// code block to be executed*

}

while (*condition*);

Note:

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

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:

for (Initialization; Test condition; increment/decrement) {

*// code block to be executed*

}

**Java for each loop:**

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

for (*type* *variableName* : *arrayName*) {

*// code block to be executed*

}

**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.

**Java Break:**

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

**Java Arrays**

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

String[] cars;

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

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

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

Change an array element

cars[0] = "Opel";

cars.length

Arrays-> arr.length;

String-> txt.length();

Multi Dimensional Array

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

**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.

blic class Main {

static void myMethod() {

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

}

public static void main(String[] args) {

myMethod();

}

}

// Outputs "I just got 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.

Calling of the Method:

myMethod();

**Parameters and Arguments:**

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

**Return value:**

Void no return value

Int, char, float return value

**Java method overloading**

multiple methods can have the same name with different parameters.

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:**  region where it is accessible

Method Scope

Block Scope

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

Halting Condition:

which is the condition where the function stops calling itself.

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

public class Main {

public static void main(String[] args) {

int result = sum(10);

System.out.println(result);

}

public static int sum(int k) {

if (k > 0) {

return k + sum(k - 1);

} else {

return 0;

}

}

}

**Java OOPs**

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.

*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

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

**Class**

**Fruit**

Object

Apple

Banana

Mango

**Object**

Volvo

Audi

Toyato

**Class**

Car

**Class** is a template for objects.

**Object** is an instance of a class.

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

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.

To create a class, use the keyword class.

public class Main {

int x = 5;

}

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

public class Main {

int x = 5;

}

class Second {

public static void main(String[] args) {

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

System.out.println(myObj.x);

}

}

**Java Class Attributes**(Another term for class attributes is **fields**.)

public class Main {

int x = 5;

int y = 3;

}

Attributes x and y

Accessing Attributes: Using . we can access attribues

**Modify Attributes**

public class Main {

int x;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 40;

System.out.println(myObj.x);

}

}

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

**final** int x = 10;

**Mulitple Attributes:**

You can specify as many attributes as you want.

**Java Class Methods:**

Static vs Non-static

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.

ublic 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

}

}

Accessing Methods with an Object

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

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

myCar.speed(200);

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

Using Multiple Classes

**Java Constructor:**

 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

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

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

Modifiers:

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

For **classes**, you can use either public or default:

|  |  |  |
| --- | --- | --- |
| public | The class is accessible by any other class |  |
| default | The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the [Packages chapter](https://www.w3schools.com/java/java_packages.asp) |  |

For **attributes, methods and constructors**, you can use the one of the following:

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** |  |
| public | The code is accessible for all classes |  |
| private | The code is only accessible within the declared class |  |
| default | The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the [Packages chapter](https://www.w3schools.com/java/java_packages.asp) |  |
| protected | The code is accessible in the same package and **subclasses**. You will learn more about subclasses and superclasses in the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp) |  |

**Non Access Modifers**

For **classes**, you can use either final or abstract:

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** |  |
| final | The class cannot be inherited by other classes (You will learn more about inheritance in the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp)) |  |
| abstract | The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the [Inheritance](https://www.w3schools.com/java/java_inheritance.asp) and [Abstraction](https://www.w3schools.com/java/java_abstract.asp) chapters) |  |

For **attributes and methods**, you can use the one of the following:

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| final | Attributes and methods cannot be overridden/modified |
| static | Attributes and methods belongs to the class, rather than an object |
| abstract | Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the [Inheritance](https://www.w3schools.com/java/java_inheritance.asp) and [Abstraction](https://www.w3schools.com/java/java_abstract.asp) chapters |
| transient | Attributes and methods are skipped when serializing the object containing them |
| synchronized | Methods can only be accessed by one thread at a time |
| volatile | The value of an attribute is not cached thread-locally, and is always read from the "main memory" |

**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

public class Person {

private String name; // private = restricted access

//getter

public String getName() {

return name;

}

// Setter

public void setName(String newName) {

this.name = newName;

}

}

The name variable is declared as private, we **cannot** access it from outside this class.

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:**

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.

Two types:

* Built-in Packages (packages from the Java API)

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

import java.util.Scanner;

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

* User-defined Packages (create your own 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:

To create a package, use the package keyword:

ackage mypack;

class MyPackageClass {

public static void main(String[] args) {

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

}

}

C:\Users\>javac MyPackageClass.java

C:\Users\>javac -d . MyPackageClass.java

*The -d keyword specifies the destination for where to save the class file.*

**Java Inheritance:**

In Java, it is possible to inherit attributes and methods from one class to another.

Subclass(child class)

Superclass(parent class)

We use extends keyword

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);

}

}

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

Final class Vehicle{

}

**Java Polymorphism:**

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

We can perform polymorphism in java by method overloading and method overriding.

**Runtime Polymorphism in java(Method overriding)**

1. **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time. 

**Example 1**

**class** Bike{

**void** run(){System.out.println("running");}

}

**class** Splendor **extends** Bike{

**void** run(){System.out.println("running safely with 60km");}

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

    Bike b = **new** Splendor();//upcasting

    b.run();

  }

}

Output:

running safely with 60km.

Example 2

**class** Bank{

**float** getRateOfInterest(){**return** 0;}

}

**class** SBI **extends** Bank{

**float** getRateOfInterest(){**return** 8.4f;}

}

**class** ICICI **extends** Bank{

**float** getRateOfInterest(){**return** 7.3f;}

}

**class** AXIS **extends** Bank{

**float** getRateOfInterest(){**return** 9.7f;}

}

**class** TestPolymorphism{

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

Bank b;

b=**new** SBI();

System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());

b=**new** ICICI();

System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());

b=**new** AXIS();

System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());

}

}

#### Rule: Runtime polymorphism can't be achieved by data members.

**class** Bike{

**int** speedlimit=90;

}

**class** Honda3 **extends** Bike{

**int** speedlimit=150;

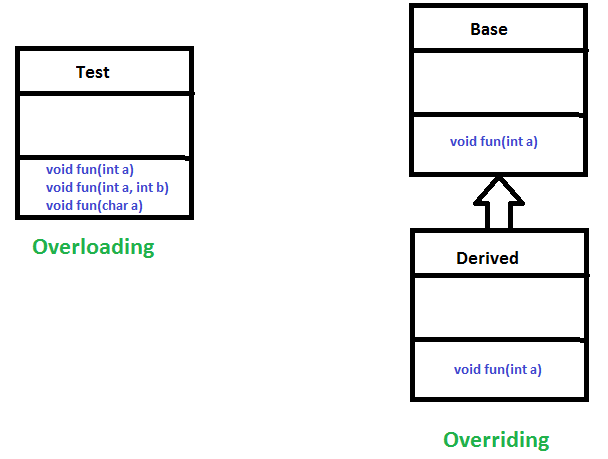
**public** **static** **void** main(String args[]){

  Bike obj=**new** Honda3();

  System.out.println(obj.speedlimit);//90

}

90



**Compile-time polymorphism** (Method overloading)

Compile-time polymorphism is a polymorphism that is resolved during the compilation process. Overloading of methods is called through the reference variable of a class. Compile-time polymorphism is achieved by **method overloading**and **operator overloading.**

We can have one or more methods with the same name that are solely distinguishable by argument numbers, type, or order.

### **different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

**Changing number of arguments**

**class** Adder{

**static** **int** add(**int** a,**int** b){**return** a+b;}

**static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}

}

**class** TestOverloading1{

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

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(11,11,11));

}}

**changing the data type**

**class** Adder{

**static** **int** add(**int** a, **int** b){**return** a+b;}

**static** **double** add(**double** a, **double** b){**return** a+b;}

}

**class** TestOverloading2{

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

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(12.3,12.6));

}}

**Java Abstraction:**

# **Java String**

In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:

1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=**new** String(ch);

is same as:

1. String s="javatpoint";

**Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* [interfaces](https://www.javatpoint.com/interface-in-java).



There are two ways to create String object:

1. By string literal
2. By new keyword

String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance



#### Note: String objects are stored in a special memory area known as the "string constant pool".

String s=**new** String("Welcome");//creates two objects and one reference variable

**public** **class** StringExample{

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

String s1="java";//creating string by Java string literal

**char** ch[]={'s','t','r','i','n','g','s'};

String s2=**new** String(ch);//converting char array to string

String s3=**new** String("example");//creating Java string by new keyword

System.out.println(s1);

System.out.println(s2);

System.out.println(s3);

}}

Output:

java

strings

example

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | It returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | It returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | It returns a formatted string. |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | It returns formatted string with given locale. |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | It returns substring for given begin index. |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | It returns substring for given begin index and end index. |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | It returns true or false after matching the sequence of char value. |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | It returns a joined string. |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | It returns a joined string. |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | It checks the equality of string with the given object. |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | It checks if string is empty. |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | It concatenates the specified string. |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | It replaces all occurrences of the specified char value. |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | It replaces all occurrences of the specified CharSequence. |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | It compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | It returns a split string matching regex. |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | It returns a split string matching regex and limit. |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | It returns an interned string. |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | It returns the specified char value index. |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | It returns the specified char value index starting with given index. |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | It returns the specified substring index. |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | It returns the specified substring index starting with given index. |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | It returns a string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | It returns a string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | It returns a string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | It returns a string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | It removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | It converts given type into string. It is an overloaded method. |

Why is string Immutable in java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

**Testimmutablestring.java**

**class** Testimmutablestring{

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

   String s="Sachin";

   s.concat(" Tendulkar");//concat() method appends the string at the end

   System.out.println(s);//will print Sachin because strings are immutable objects

 }

}

 **Testimmutablestring1.java**

**class** Testimmutablestring1{

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

   String s="Sachin";

   s=s.concat(" Tendulkar");

   System.out.println(s);

 }

}

Why string objects are immutable in java

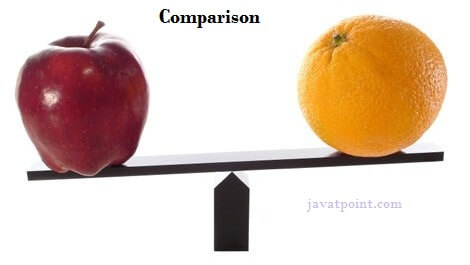
1.ClassLoader

2.Thread Safe

3.Security

4.Heap Space

**String Compare**



It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method

By Using equals() Method

* **public boolean equals(Object another)** compares this string to the specified object.
* **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

**class** Teststringcomparison1{

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

   String s1="Sachin";

   String s2="Sachin";

   String s3=**new** String("Sachin");

   String s4="Saurav";

   System.out.println(s1.equals(s2));//true

   System.out.println(s1.equals(s3));//true

   System.out.println(s1.equals(s4));//false

 }

}

Output:

true

true

false

**class** Teststringcomparison2{

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

   String s1="Sachin";

   String s2="SACHIN";

   System.out.println(s1.equals(s2));//false

   System.out.println(s1.equalsIgnoreCase(s2));//true

 }

}

**Output:**

false

true

By Using == Operator

**class** Teststringcomparison3{

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

   String s1="Sachin";

   String s2="Sachin";

   String s3=**new** String("Sachin");

   System.out.println(s1==s2);//true (because both refer to same instance)

   System.out.println(s1==s3);//false(because s3 refers to ins

**Output:**

true

false

**By compareTo() Method**

Suppose s1 and s2 are two String objects. If:

**s1 == s2** : The method returns 0.

**s1 > s2** : The method returns a positive value.

**s1 < s2** : The method returns a negative value.

**class** Teststringcomparison4{

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

   String s1="Sachin";

   String s2="Sachin";

   String s3="Ratan";

   System.out.println(s1.compareTo(s2));//0

   System.out.println(s1.compareTo(s3));//1(because s1>s3)

   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )

 }

}

**Output:**

0

1

-1

**String Concatenation in Java**

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1. By + (String concatenation) operator
2. By concat() method

**By + (String concatenation) operator**

**TestStringConcatenation1.java**

**class** TestStringConcatenation1{

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

   String s="Sachin"+" Tendulkar";

   System.out.println(s);//Sachin Tendulkar

 }

}

**Output:**

Sachin Tendulkar

The **Java compiler transforms** above code to this:

1. String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar) .toString();

**class** TestStringConcatenation2{

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

   String s=50+30+"Sachin"+40+40;

   System.out.println(s);//80Sachin4040

 }

}

**Output:**

80Sachin4040

#### Note: After a string literal, all the + will be treated as string concatenation operator.

**By concat() method**

**class** TestStringConcatenation3{

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

   String s1="Sachin ";

   String s2="Tendulkar";

   String s3=s1.concat(s2);

   System.out.println(s3);//Sachin Tendulkar

  }

}

**Output:**

Sachin Tendulkar

**Exception Handling in Java**

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

 Exception is an abnormal condition.

 an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.



**Types of Exceptions**

1. Checked Exception
2. Unchecked Exception
3. Error



### **1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### **2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### **3) Error**

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |

**JavaExceptionExample.java**

1. **public** **class** JavaExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //code that may raise exception
5. **int** data=100/0;
6. }**catch**(ArithmeticException e){System.out.println(e);}
7. //rest code of the program
8. System.out.println("rest of the code...");
9. }
10. }

**Output:**

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

**int** a=50/0;//ArithmeticException

String s=**null**;

System.out.println(s.length());//NullPointerException

String s="abc";

**int** i=Integer.parseInt(s);//NumberFormatException

**int** a[]=**new** **int**[5];

a[10]=50; //ArrayIndexOutOfBoundsException

**Java Try Catch Block**

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

1. **try**{
2. //code that may throw an exception
3. }**catch**(Exception\_class\_Name ref){}
4. **try**{
5. //code that may throw an exception
6. }**finally**{}
7. 

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.
* **public** **class** TryCatchExample3 {
* **public** **static** **void** main(String[] args) {
* **try**
* {
* **int** data=50/0; //may throw exception
* // if exception occurs, the remaining statement will not exceute
* System.out.println("rest of the code");
* }
* // handling the exception
* **catch**(ArithmeticException e)
* {
* System.out.println(e);
* }
* }

**Output:**

java.lang.ArithmeticException: / by zero

**public** **class** TryCatchExample9 {

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

**try**

        {

**int** arr[]= {1,3,5,7};

        System.out.println(arr[10]); //may throw exception

        }

            // handling the array exception

**catch**(ArrayIndexOutOfBoundsException e)

        {

            System.out.println(e);

        }

        System.out.println("rest of the code");

    }

}

**Output:**

java.lang.ArrayIndexOutOfBoundsException: 10

rest of the code

**Java Catch Multiple Exceptions**

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

