# Java Introduction

**JAVA** was developed by James Gosling at **Sun Microsystems** Inc in 1991, later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs.

[Java](https://www.geeksforgeeks.org/java/) is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to *write once run anywhere* that is compiled Java code can run on all platforms that support Java. Java applications are compiled to bytecode that can run on any Java Virtual Machine. The syntax of Java is similar to c/c++.

The principles for creating java were simple, robust, secured, high performance, portable, multi-threaded, interpreted, dynamic, etc. **James** Gosling in 1995 developed Java, who is known as the Father of Java. Currently, Java is used in mobile devices, internet programming, games, e-business, etc.

# Features of Java

### 1) Simple

Java is easy to learn and its syntax is quite simple, clean and easy to understand.The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.

*Eg :* Pointers and Operator Overloading are not there in java but were an important part of C++.

2)Object-oriented

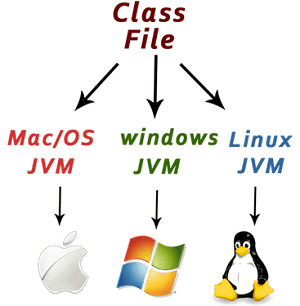
Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

3)Platform Independent

Java is platform independent because it is different from other languages like [C](https://www.javatpoint.com/c-programming-language-tutorial), [C++](https://www.javatpoint.com/cpp-tutorial), etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms: software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

4)Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

* **No explicit pointer**
* **Java Programs run inside a virtual machine sandbox**
* **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
* **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access rights to objects.
* **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

5)Robust

The English meaning of Robust is strong. Java is robust because:

* It uses strong memory management.
* There is a lack of pointers that avoids security problems.
* Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* There are exception handling and the type checking mechanism in Java. All these points make Java robust.

6)Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

7)Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

8)**High Performance**

With the use of Just-In-Time compilers, Java enables high performance.

### 9)Distributed

Java is designed for the distributed environment of the internet. Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

10)Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multimedia, Web applications, etc.

11)Dynamic

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

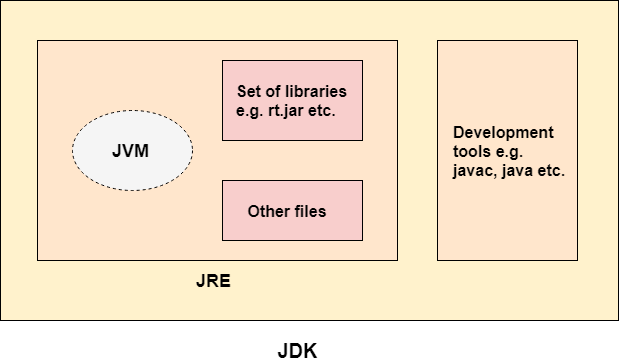
# Difference between JDK, JRE, and JVM

# JDK: Java Development Kit

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation:

* Standard Edition Java Platform
* Enterprise Edition Java Platform
* Micro Edition Java Platform

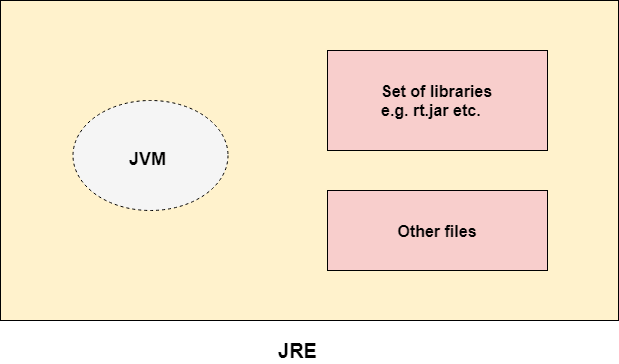
The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.

Here are the important reasons of using JDK:

* JDK contains tools required to write Java programs, and JRE to execute them.
* It includes a compiler, Java application launcher, Appletviewer, etc.
* Compiler converts code written in Java into byte code.
* Java application launcher opens a JRE, loads the necessary class, and executes its main method.

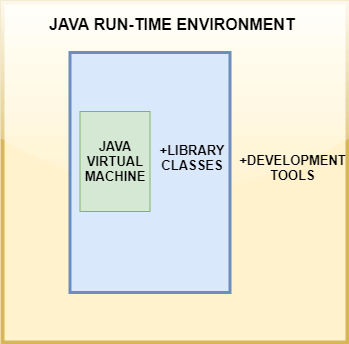
### JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

Java Run-time Environment (JRE) is part of the Java Development Kit (JDK). It has Java Class Library, specific tools, and a stand-alone JVM. It is the most common environment available on devices to run java programs. The source Java code gets compiled and converted to Java bytecode. If you wish to run this bytecode on any platform, you require JRE. The JRE loads classes, verify access to memory, and retrieves the system resources. JRE acts as a layer on the top of the operating system

JRE consists of the following components:

* **Deployment technologies** such as deployment, Java plug-in, and Java Web Start.
* **User interface toolkits**, including Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag, and drop (DnD) and input methods.
* **Integration libraries** including Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) and scripting.
* **Other base libraries**, including international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization and Java for XML Processing (XML JAXP).
* **Lang and util base libraries**, including lang and util, zip, Java Archive (JAR), instrument, reflection, Collections, Concurrency Utilities, management, versioning, Logging, Preferences API, Ref Objects and Regular Expressions.
* **Java Virtual Machine (JVM)**, which comprises Server Virtual Machine and Java HotSpot Client.

Here are theimportant features of JRE:

* Java Runtime Environment is a set of tools using which the JVM actually runs.
* JRE contains deployment technology, including Java Web Start and Java Plug-in.
* Developers can easily run the source code in JRE, but he/she cannot write and compile the Java program.
* It includes integration libraries like Java Database Connectivity (JDBC), Remote Method Invocation (RMI), Java Naming and Directory Interface (JNDI), and more.
* JRE has a JVM and Java HotSpot virtual machine client.

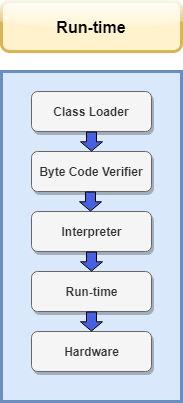
.JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

Once you write this program, you have to save it with .java extension. Compile your program. The output of the Java compiler is a byte-code which is platform independent. After compiling, the compiler generates a .class file which has the bytecode. The bytecode is platform independent and runs on any device having the JRE. From here, the work of JRE begins. To run any Java program, you need JRE. The flow of the bytecode to run is as follows:

### What it does

The JVM performs following operation:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

JVM provides definitions for the:

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

Here are the important features of JVM:

* It enables you to run applications in a cloud environment or on your device.
* Java Virtual Machine converts bytecode to the machine-specific code.
* It provides basic java functions like memory management, security, garbage collection, and more.
* JVM runs the program by using libraries and files given by Java Runtime Environment.
* JDK and JRE both contain Java Virtual Machine.
* It can execute the java program line by line hence it is also called an interpreter.
* JVM is easily customizable for example, you can allocate minimum and maximum memory to it.
* It is independent from hardware and the operating system. So, you can write a java program once and run anywhere.

# Print in Java

In Java, we usually use the **println()** method to print the statement. It belongs to the **PrintStream** class. The class also provides the other methods for the same purpose. In this section, we will learn **how to print in** [**Java**](https://www.javatpoint.com/java-tutorial). Along with this, we will also explain the statement **System.out.println()**.

The method we should use depends on what we want to print and what type of output we want. There are following three methods to print the statements:

* print() Method
* println() Method
* printf() Method

### print() Method

The print() method is used to print text on the console. It is an overloaded method of the **PrintStream** class. It accepts a string as a parameter. After printing the statement, the cursor remains on the same line. It also works if we do not parse any parameter.

### println() Method

It is an upgraded version of the print() method. It also used to display text on the console. It is an overloaded method of the **PrintStream** class. It accepts string as a parameter. After printing the statement, it throws the cursor at the starting of the next line. It is the main difference between the println() and the print() method.

### printf() Method

The printf() method is used if we want to print the formatted string to the console using the specified format string and arguments. It is also an overloaded method of the PrintStream class. The method behaves the same as the invocation of the format() method. want to know more about formatted String, go through the link <https://bit.ly/2EaKzmq>

1. **import** java.io.\*;
2. **public** **class** PrintDemo
3. {
4. **public** **static** **void** main(String[] args)
5. {
6. **int** num = 122;
7. **char** ch = 'A';
8. String str = "Oracle";
9. **double** d = 190.98;
10. **float** f = 3.14f;
11. //prints the values on the console
12. System.out.println(); //prints nothing but throws the cursor to the next line
13. System.out.println(num); //prints integer
14. System.out.println(ch); //prints character
15. System.out.print(str+"\n");
16. System.out.print(d +"\n");
17. System.out.print(f+"\n");
18. System.out.printf("'%s' %n", "javatpoint");
19. System.out.printf("'%S' %n", "Jack");
20. }
21. }

# Java User Input (Scanner)

## Java Scanner Class

Java **Scanner class** allows the user to take input from the console. It belongs to **java.util** package. It is used to read the input of primitive types like int, double, long, short, float, and byte. It is the easiest way to read input in a Java program.

### Syntax

1. Scanner sc=**new** Scanner(System.in);

The above statement creates a constructor of the Scanner class having **System.in** as an argument. It means it is going to read from the standard input stream of the program. The **java.util** package should be imported while using the Scanner class.

## Methods of Java Scanner Class

Java Scanner class provides the following methods to read different primitives types:

| **Method** | **Description** |
| --- | --- |
| **int nextInt()** | It is used to scan the next token of the input as an integer. |
| **float nextFloat()** | It is used to scan the next token of the input as a float. |
| **double nextDouble()** | It is used to scan the next token of the input as a double. |
| **byte nextByte()** | It is used to scan the next token of the input as a byte. |
| **String nextLine()** | Advances this scanner past the current line. |
| **boolean nextBoolean()** | It is used to scan the next token of the input into a boolean value. |
| **long nextLong()** | It is used to scan the next token of the input as a long. |
| **short nextShort()** | It is used to scan the next token of the input as a Short. |
| **BigInteger nextBigInteger()** | It is used to scan the next token of the input as a BigInteger. |
| **BigDecimal nextBigDecimal()** | It is used to scan the next token of the input as a BigDecimal. |



# Java Comments

## Java Comments

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

## Single-line Comments

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

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

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

### Example

// This is a comment

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

## 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");

# Java Variables

# Java Variables

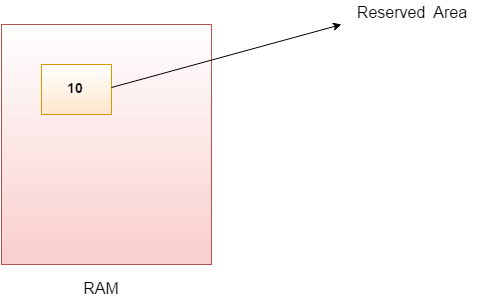
A variable is a container which holds the value while the [Java program](https://www.javatpoint.com/simple-program-of-java) is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of [data types in Java](https://www.javatpoint.com/java-data-types): primitive and non-primitive.

## Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

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

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

## Declaring (Creating) Variables

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

### Syntax

*type variable = value;*

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

### Example

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

String name = "John";

System.out.println(name);

## Final Variables

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

### Example

final int myNum = 15;

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

## Other Types

A demonstration of how to declare variables of other types:

### Example

int myNum = 5;

float myFloatNum = 5.99f;

char myLetter = 'D';

boolean myBool = true;

String myText = "Hello";

## Display Variables

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

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

### Example

String name = "John";

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

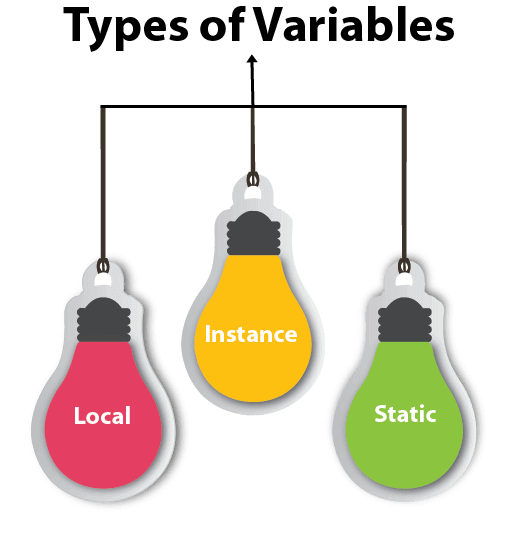
## Java Identifiers

All Java variables must be identified with unique names.

These unique names are called identifiers.

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

The general rules for constructing names for variables (unique identifiers) 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

### Types of Variables

There are three types of variables in [Java](https://www.javatpoint.com/java-tutorial):

* local variable
* instance variable
* static variable

#### 1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

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

#### 2) Instance Variable

* A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as [static](https://www.javatpoint.com/static-keyword-in-java).
* It is called an instance variable because its value is instance-specific and is not shared among instances.

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

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

#### 3) Static variable

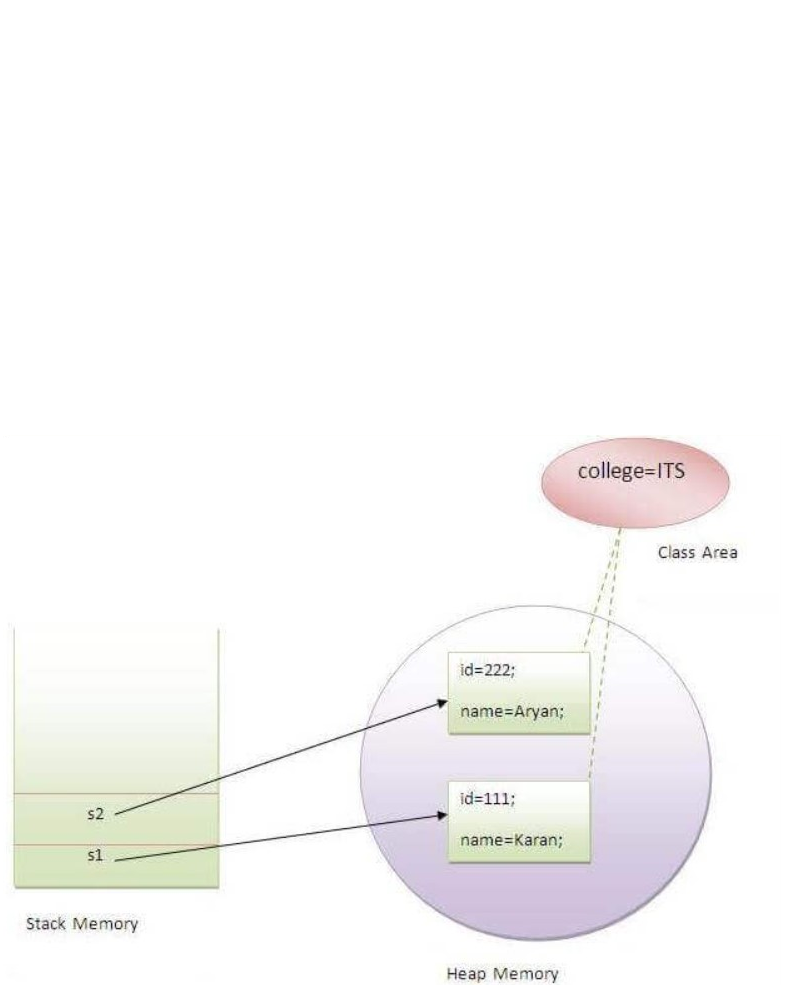
A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

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

Now let us do discuss the differences between the Instance variable Vs the Static variables

* Each object will have its own copy of the instance variable whereas We can only have one copy of a static variable per class irrespective of how many objects we create.
* Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of static, changes will be reflected in other objects as static variables are common to all objects of a class.
* We can access instance variables through object references and Static Variables can be accessed directly using class name**.**





# Java Files

In Java, a **File** is an abstract data type. A named location used to store related information is known as a **File**. There are several **File Operations** like **creating a new File, getting information about File, writing into a File, reading from a File** and **deleting a File**.

## Stream

A series of data is referred to as **a stream**. In [Java](https://www.javatpoint.com/java-tutorial), **Stream** is classified into two types, i.e., **Byte Stream** and **Character Stream**.

**Java I/O** (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

**1) System.out:** standard output stream

**2) System.in:** standard input stream

**3) System.err:** standard error stream

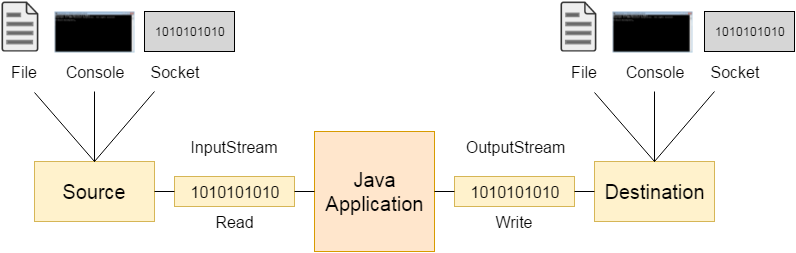
### OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

### InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



## OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

## OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

### OutputStream Hierarchy

Java output stream hierarchy

## InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

### Useful methods of InputStream

| **Method** | **Description** |
| --- | --- |
| 1) public abstract int read()throws IOException | reads the next byte of data from the input stream. It returns -1 at the end of the file. |
| 2) public int available()throws IOException | returns an estimate of the number of bytes that can be read from the current input stream. |
| 3) public void close()throws IOException | is used to close the current input stream. |

### InputStream Hierarchy

Java input stream hierarchy

### Byte Stream

**Byte Stream** is mainly involved with byte data. A file handling process with a byte stream is a process in which an input is provided and executed with the byte data.

### Character Stream

**Character Stream** is mainly involved with character data. A file handling process with a character stream is a process in which an input is provided and executed with the character data.

## Java File Class Methods

| S.No. | Method | Return Type | Description |
| --- | --- | --- | --- |
| 1. | canRead() | Boolean | The canRead() method is used to check whether we can read the data of the file or not. |
| 2. | createNewFile() | Boolean | The createNewFile() method is used to create a new empty file. |
| 3. | canWrite() | Boolean | The canWrite() method is used to check whether we can write the data into the file or not. |
| 4. | exists() | Boolean | The exists() method is used to check whether the specified file is present or not. |
| 5. | delete() | Boolean | The delete() method is used to delete a file. |
| 6. | getName() | String | The getName() method is used to find the file name. |
| 7. | getAbsolutePath() | String | The getAbsolutePath() method is used to get the absolute pathname of the file. |
| 8. | length() | Long | The length() method is used to get the size of the file in bytes. |
| 9. | list() | String[] | The list() method is used to get an array of the files available in the directory. |
| 10. | mkdir() | Boolean | The mkdir() method is used for creating a new directory. |

## File Operations

We can perform the following operation on a file:

* Create a File
* Get File Information
* Write to a File
* Read from a File
* Delete a File

### Create a File

**Create a File** operation is performed to create a new file. We use the **createNewFile()** method of file. The **createNewFile()** method returns true when it successfully creates a new file and returns a message when the file already exists.



**Explanation:**

In the above code, we import the File and IOException class for performing file operation and handling errors, respectively. We create the **fileobj** object of the File class and specify the location of the directory where we want to create a file. In the try block, we call the **createNewFile()** method through the **fileobj** object to create a new file in the specified location. If the method returns false, it will jump to the else section. If there is any error, it gets handled in the catch block.

### Get File Information

The operation is performed to get the file information. We use several methods to get the information about the file like name, absolute path, is readable, is writable and length.



In the above code, we import the **java.io.File** package and create a class **FileInfo**. In the main method, we create an object of the text file which we have created in our previous example. We check the existence of the file using a conditional statement, and if it is present, we get the following information about that file:

1. We get the name of the file using the **getName()**
2. We get the absolute path of the file using the **getAbsolutePath()** method of the file.
3. We check whether we can write data into a file or not using the **canWrite()**
4. We check whether we can read the data of the file or not using the **canRead()**
5. We get the length of the file by using the **length()**

If the file doesn't exist, we show a custom message.

### Write to a File

The next operation which we can perform on a file is **"writing into a file"**. In order to write data into a file, we will use the **FileWriter** class and its **write()** method together. We need to close the stream using the **close()** method to retrieve the allocated resources.

Let's take an example to understand how we can write data into a file.



**Explanation:**

In the above code, we import the **java.io.FileWriter** and **java.io.IOException** classes. We create a class WriteToFile, and in its main method, we use the **try-catch** block. In the try section, we create an instance of the FileWriter class, i.e., **filewriteobj**. We call the **write** method of the FileWriter class and pass the content to that function which we want to write. After that, we call the **close()** method of the FileWriter class to close the file stream. After writing the content and closing the stream, we print a custom message.

If we get any error in the try section, it jumps to the catch block. In the catch block, we handle the **IOException** and print a custom message.

### Read from a File

The next operation which we can perform on a file is **"read from a file"**. In order to read data from a file, we will use the **Scanner** class. Here, we need to close the stream using the **close()** method. We will create an instance of the [Scanner class](https://www.javatpoint.com/Scanner-class) and use the [**hasNextLine()** method](https://www.javatpoint.com/post/java-scanner-hasnextline-method) [**nextLine()** method](https://www.javatpoint.com/post/java-scanner-nextline-method) to get data from the file.

Let's take an example to understand how we can read data from a file.



**Explanation:**

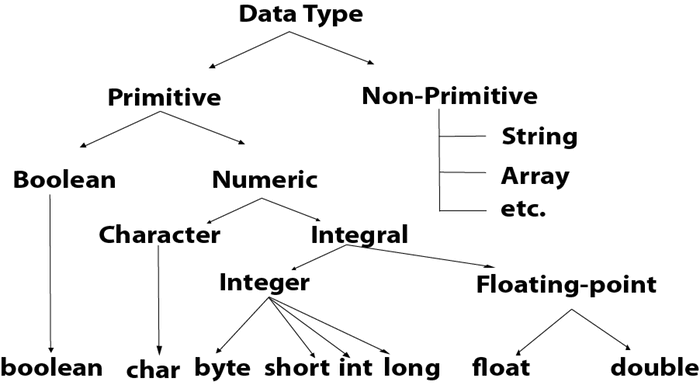
In the above code, we import the **"java.util.Scanner", "java.io.File"** and **"java.io.IOException"** classes. We create a class **ReadFromFile**, and in its main method, we use the **try-catch block**. In the try section, we create an instance of both the **Scanner** and the **File** classes. We pass the **File** class object to the **Scanner** class object and then iterate the Scanner class object using the **"While"** loop and print each line of the file. We also need to close the scanner class object, so we use the close() function. If we get any error in the try section, it jumps to the catch block. In the catch block, we handle the IOException and print a custom message.

# Java Data Types

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).

## Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in [Java language](https://www.javatpoint.com/java-tutorial).

Java is a statically-typed programming language. It means, all [variables](https://www.javatpoint.com/java-variables) must be declared before its use. That is why we need to declare a variable's type and name

| 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 |

## Numbers

Primitive number types are divided into two groups:

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

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

## Integer Types

### Byte

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

### Example

byte myNum = 100;

System.out.println(myNum);

### Short

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

### Example

short myNum = 5000;

System.out.println(myNum);

### Int

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

### Example

int myNum = 100000;

System.out.println(myNum);

### Long

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

### Example

long myNum = 15000000000L;

System.out.println(myNum);

## Floating Point Types

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

### Float

The float data type can store fractional numbers from 3.4e−038 to 3.4e+038. Note that you should end the value with an "f":

### Example

float myNum = 5.75f;

System.out.println(myNum);

### Double

The double data type can store fractional numbers from 1.7e−308 to 1.7e+308. Note that you should end the value with a "d":

### Example

double myNum = 19.99d;

System.out.println(myNum);

Use float or double?

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

### Scientific Numbers

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

### Example

float f1 = 35e3f;

double d1 = 12E4d;

System.out.println(f1);

System.out.println(d1);

## Booleans

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

### Example

boolean isJavaFun = true;

boolean isFishTasty = false;

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

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

## Characters

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

### Example

char myGrade = 'B';

System.out.println(myGrade);

Alternatively, you can use ASCII values to display certain characters:

char a = 65, b = 66, c = 67;

System.out.println(a);

System.out.println(b);

System.out.println(c);

## Strings

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

Example

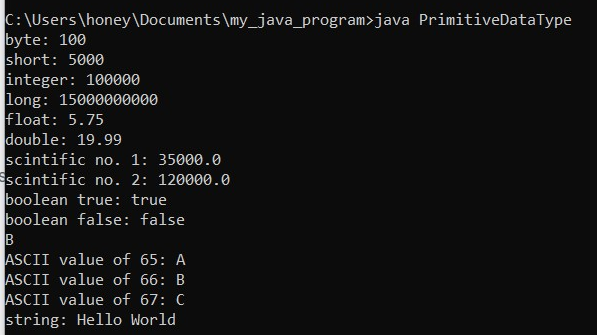
String greeting = "Hello World";

System.out.println(greeting);

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

A String in Java is actually a non-primitive data type, because it refers to an object. The String object has methods that are used to perform certain operations on strings.

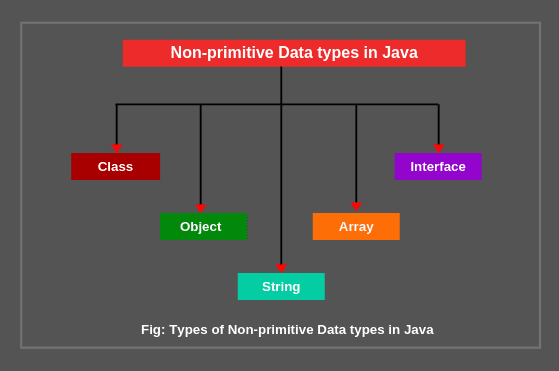




## Non-Primitive Data Types

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

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

* Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and are not defined by Java (except for String).
* Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type has always a value, while non-primitive types can be null.
* A primitive type starts with a lowercase letter, while non-primitive types start with an uppercase letter.
* The size of a primitive type depends on the data type, while non-primitive types all have the same size.

Examples of non-primitive types are [Strings](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp), [Classes,](https://www.w3schools.com/java/java_classes.asp) [Interface](https://www.w3schools.com/java/java_interface.asp), etc.

There are five types of non-primitive data types in Java. They are as follows:

1. Class
2. Object
3. String
4. Array
5. Interface

**1. Class and objects:** Every class is data type and it is also considered as user-defined data types. This is because a user creates a class.

**2. String:** A string represents a sequence of characters like India, ABC123, etc. The simplest way to create a string object is by storing sequence of characters into string type variable like this:

String str = “Universe”;

Here, string type variable str contains “Universe”. A string is also a class.

**3. Array:** An array in java is an object which is used to store multiple variables of the same type. These variables can be primitive or non-primitive data types.

The example of declaring an array variable of primitive data type int is as follows:

int [ ] scores;

The example of declaring an array variable of non-primitive data type is

Student [ ] students; // Student is the name of class.

**4. Interface:** An interface is declared like a class but the only difference is that it contains only final variables and method declarations. It is a fully abstract class.

Here, we have given just basic knowledge of non-primitive data types in java.

# Java Operators

ava provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups −

* Arithmetic Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Assignment Operators
* Conditional Operators

## The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators −

Assume integer variable A holds 10 and variable B holds 20, then

| Operator | Description | Example |
| --- | --- | --- |
| + (Addition) | Adds values on either side of the operator. | A + B will give 30 |
| - (Subtraction) | Subtracts right-hand operand from left-hand operand. | A - B will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator. | A \* B will give 200 |
| / (Division) | Divides left-hand operand by right-hand operand. | B / A will give 2 |
| % (Modulus) | Divides left-hand operand by right-hand operand and returns remainder. | B % A will give 0 |
| ++ (Increment) | Increases the value of operand by 1. | B++ gives 21 |
| -- (Decrement) | Decreases the value of operand by 1. | B-- gives 19 |

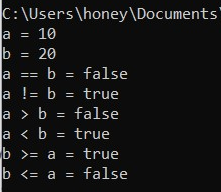
## 

## The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20 then

| Operator | Description | Example |
| --- | --- | --- |
| == (equal to) | Checks if the values of two operands are equal or not, if yes then the condition becomes true. | (A == B) is not true. |
| != (not equal to) | Checks if the values of two operands are equal or not, if values are not equal then the condition becomes true. | (A != B) is true. |
| > (greater than) | Checks if the value of the left operand is greater than the value of right operand, if yes then the condition becomes true. | (A > B) is not true. |
| < (less than) | Checks if the value of the left operand is less than the value of the right operand, if yes then the condition becomes true. | (A < B) is true. |
| >= (greater than or equal to) | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= (less than or equal to) | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |



## 

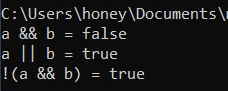
## The Logical Operators

The following table lists the logical operators −

Assume Boolean variables A holds true and variable B holds false, then −

[Show Examples](https://www.tutorialspoint.com/java/java_logical_operators_examples.htm)

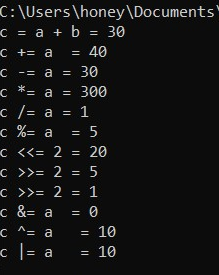
| Operator | Description | Example |
| --- | --- | --- |
| && (logical and) | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false |
| || (logical or) | Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. | (A || B) is true |
| ! (logical not) | Called Logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then the Logical NOT operator will make it false. | !(A && B) is true |



## The Assignment Operators

Following are the assignment operators supported by Java language −

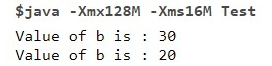
| Operator | Description | Example |
| --- | --- | --- |
| = | Simple assignment operator. Assigns values from right side operands to left side operand. | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand. | C -= A is equivalent to C = C – A |
| \*= | Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |



### Conditional Operator ( ? : )

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as −

variable x = (expression) ? value if true : value if false



# Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, [Java](https://www.javatpoint.com/java-tutorial) provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements/Java Conditions
   * if statements
   * switch statement
2. Loop statements
   * do while loop
   * while loop
   * for loop
   * for-each loop
3. Jump statements
   * break statement
   * continue statement

### Decision-Making statements/Java Conditions:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

Java supports the usual logical conditions from mathematics:

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

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

Java has the following conditional statements:

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

### 1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

### 1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

1. **if**(condition) {
2. statement 1; //executes when condition is true
3. }

### 2) if-else statement

The [if-else statement](https://www.javatpoint.com/java-if-else) is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

1. **if**(condition) {
2. statement 1; //executes when condition is true
3. }
4. **else**{
5. statement 2; //executes when condition is false
6. }

### 3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

1. **if**(condition 1) {
2. statement 1; //executes when condition 1 is true
3. }
4. **else** **if**(condition 2) {
5. statement 2; //executes when condition 2 is true
6. }
7. **else** {
8. statement 2; //executes when all the conditions are false
9. }

### 4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

1. **if**(condition 1) {
2. statement 1; //executes when condition 1 is true
3. **if**(condition 2) {
4. statement 2; //executes when condition 2 is true
5. }
6. **else**{
7. statement 2; //executes when condition 2 is false
8. }
9. }

Consider the following example.

**Student.java**



# Java Switch

The Java *switch statement* executes one statement from multiple conditions.Use the switch statement to select one of many code blocks to be executed.The switch statement allows us to execute a block of code among many alternatives.

**Syntax:**

**switch**(expression){

**case** value1:

//code to be executed;

**break**; //optional

**case** value2:

//code to be executed;

**break**; //optional

......

**default**:

code to be executed **if** all cases are not matched;

}

This is how it works:

* The switch expression is evaluated once.
* The value of the expression is compared with the values of each case.
* If there is a match, the associated block of code is executed.
* The break and default keywords are optional

## The break Keyword

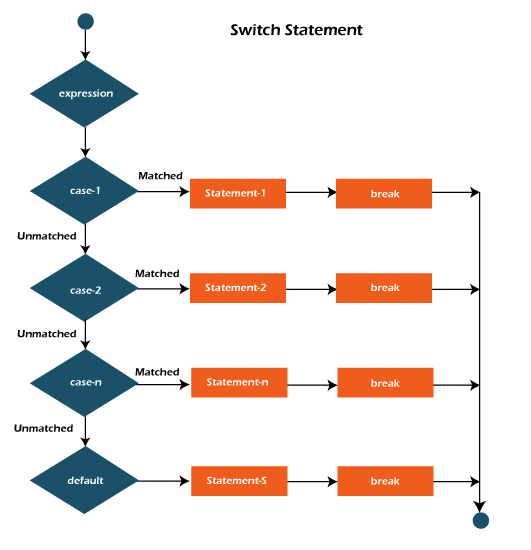
When Java reaches a break keyword, it breaks out of the switch block.

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

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

## The default Keyword

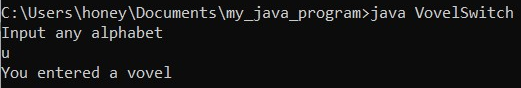
The default keyword specifies some code to run if there is no case match











# Java For Loop

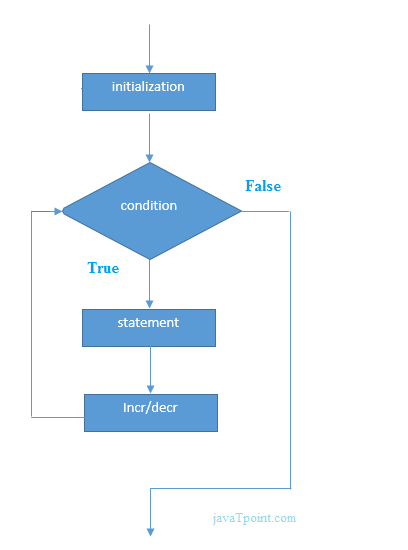
The Java *for loop* is used to iterate a part of the program several times. If the number of iterations is **fixed**, it is recommended to use a for loop.

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

## Java Simple for Loop

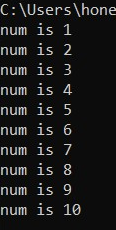
A simple for loop is the same as [C](https://www.javatpoint.com/c-programming-language-tutorial)/[C++](https://www.javatpoint.com/cpp-tutorial). We can initialize the [variable](https://www.javatpoint.com/java-variables), check condition and increment/decrement value. It consists of four parts:

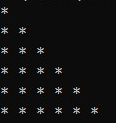
1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return a boolean value either true or false. It is an optional condition.
3. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.
4. **Statement**: The statement of the loop is executed each time until the second condition is false.

**Syntax:**

**for**(initialization; condition; increment/decrement){

//statement or code to be executed

}



## For-Each Loop/ Enhanced for loop

There is also a "for-each" loop, which is used exclusively to loop through elements in an array. In Java, the for-each loop is used to iterate through elements of [arrays](https://www.programiz.com/java-programming/arrays) and collections (like [ArrayList](https://www.programiz.com/java-programming/arraylist)). It is also known as the enhanced for loop. It is known as the for-each loop because it traverses each element one by one.

The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order. Here, you do not have the option to skip any element because it does not work on an index basis.

### Syntax

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

*// code block to be executed*

}

### Example







# Java While Loop

The [Java](https://www.javatpoint.com/java-tutorial) *while loop* is used to iterate a part of the [program](https://www.javatpoint.com/programs-list) repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while [loop](https://www.javatpoint.com/java-for-loop).

### Syntax

while (*condition*) {

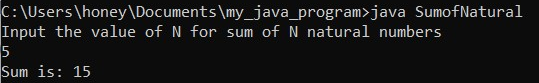
*// code block to be executed*

}









## 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.

The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iterations is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while checks the condition at the end of loop body. The Java *do-while loop* is executed at least once because the condition is checked after the loop body.

### Syntax

do {

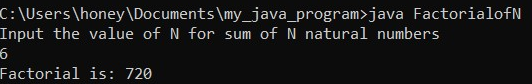
*// code block to be executed*

}

while (*condition*);

### Example





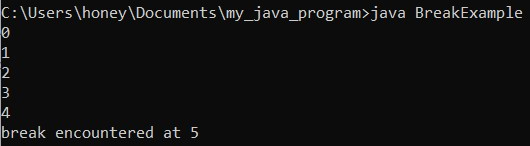
# Java Break and Continue

## Java Break

The Java *break* statement is used to break a loop or [switch](https://www.javatpoint.com/java-switch) statement. It breaks the current flow of the program at specified conditions. In the case of an inner loop, it breaks only the inner loop.

We can use Java break statements in all types of loops such as [for loop](https://www.javatpoint.com/java-for-loop), [while loop](https://www.javatpoint.com/java-while-loop) and [do-while loop](https://www.javatpoint.com/java-do-while-loop).





# Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

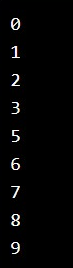
The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In the case of an inner loop, it continues the inner loop only.

We can use Java continue statements in all types of loops such as for loop, while loop and do-while loop.

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

This example skips the value of 4:

### Example



# Java OOPs Concepts

Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

## OOPs (Object-Oriented Programming System)

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

* [Object](https://www.javatpoint.com/object-and-class-in-java)
* Class
* [Inheritance](https://www.javatpoint.com/inheritance-in-java)
* [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
* [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
* [Encapsulation](https://www.javatpoint.com/encapsulation)

## Object



It is a basic unit of Object-Oriented Programming and represents real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

## Class

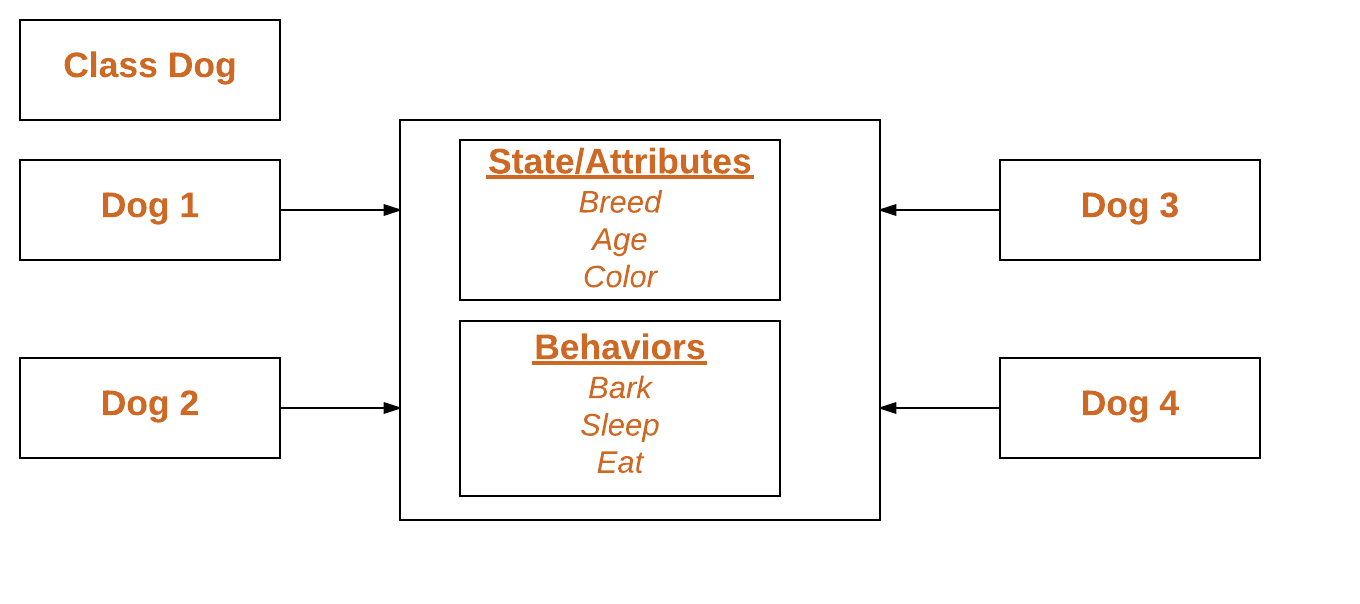
A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access **class keyword:** class keyword is used to create a class.
2. **Class name:** The name should begin with an initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }

### **Declaring Objects (Also called instantiating a class)**

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example:



### Inheritance

*When one object acquires all the properties and behaviors of a parent object*, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

Let us discuss some of frequent used important terminologies:

* **Super Class:** The class whose features are inherited is known as superclass(or a base class or a parent class).
* **Sub Class:** The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
* **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

### Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc. In Java, we use method overloading and method overriding to achieve polymorphism.

### Abstraction

*Hiding internal details and showing functionality* is known as abstraction. For example phone calls, we don't know the internal processing.

Data Abstraction is the property by virtue of which only the essential details are displayed to the user.The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

In Java, we use abstract classes and interface to achieve abstraction.

### Encapsulation

*Binding (or wrapping) code and data together into a single unit are known as encapsulation*. For example, a capsule, it is wrapped with different medicines.

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

A java class is an example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

# Java Classes and Objects

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

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

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

**Object Definitions:**

* An object is *a real-world entity*.
* An object is *a runtime entity*.
* The object is *an entity which has state and behavior*.
* The object is *an instance of a class*.

## Create a Class

To create a class, use the keyword class:

### Syntax to declare a class:

**class** <class\_name>{

field;

method;

}

### Main.java

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

public class Main {

int x = 5;

}

### Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variables don't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

### Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

#### Advantage of Method

* Code Reusability
* Code Optimization

### new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in the Heap memory area.

## Create an Object

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

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

### Example

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

public class Main {

int x = 5;

public static void main(String[] args) {

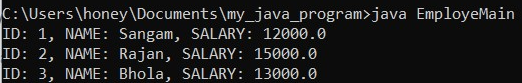
Main myObj = new Main();

System.out.println(myObj.x);

}

}





# Java Methods

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

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

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

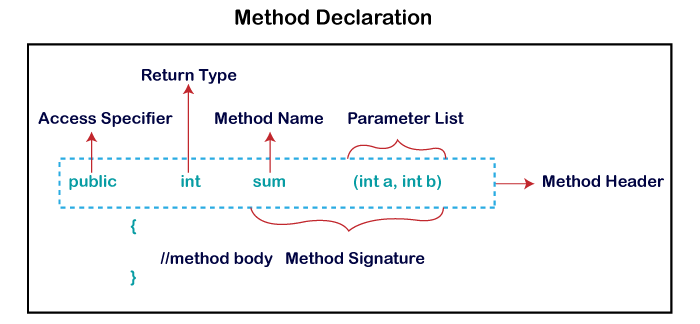
Why use methods? To reuse code: define the code once, and use it many times.

## Create a Method

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

### Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method headers**, as we have shown in the following figure.



**Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

**Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

* **Public:** The method is accessible by all classes when we use a public specifier in our application.
* **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
* **Protected:** When we use protected access specifiers, the method is accessible within the same package or subclasses in a different package.
* **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

**Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use the void keyword.

**Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

**Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

**Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

### Static Method

A method that has a static keyword is known as a static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

### Example of static method

**Display.java**



### Instance Method

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

**InstanceMethodExample.java**

****

## Return Values

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

### Example



You can also store the result in a variable (recommended, as it is easier to read and maintain):



# Java Modifiers

Modifiers are keywords that you add to those definitions to change their meanings. Java language has a wide variety of modifiers, including the following −

We divide modifiers into two groups:

Access Modifiers - controls the access level

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

## Access Modifiers

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

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

| **Modifier** | **Description** |  |
| --- | --- | --- |
| 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. |  |

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. |  |
| protected | The code is accessible in the same package and subclasses. |  |

## Non-Access Modifiers

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" |



# Constructors in Java

In [Java](https://www.javatpoint.com/java-tutorial), a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling the constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such cases, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

**Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because the Java compiler creates a default constructor if your class doesn't have any.

### Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

## Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

## Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameters.

| In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation. |
| --- |

//Java Program to create and call a default constructor

**class** Bike1{

//creating a default constructor

Bike1(){System.out.println("Bike is created");}

//main method

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

//calling a default constructor

Bike1 b=**new** Bike1();

}

}

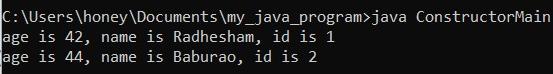
### Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

### Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.





# Inheritance in Java

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

### Why use inheritance in java

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

### Terms used in Inheritance

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

### The syntax of Java Inheritance

**class** Subclass-name **extends** Superclass-name

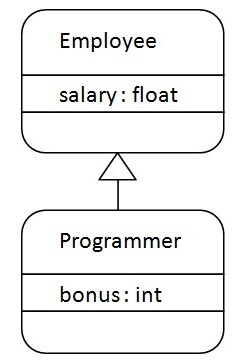
{

//methods and fields

}

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

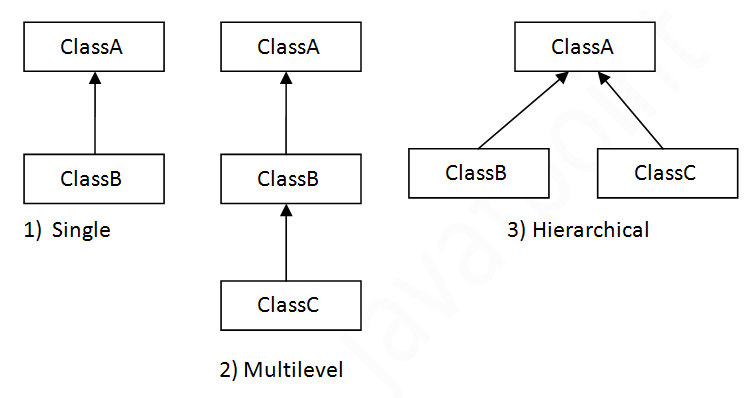
In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

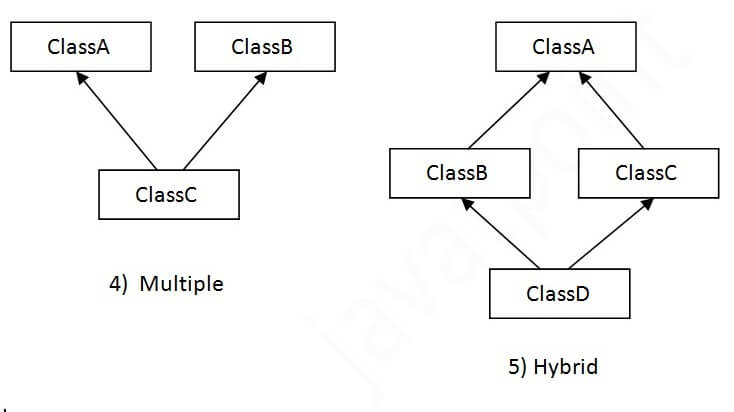
As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.



## Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.





## Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.



## Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

## Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, the BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.



## Hierarchical Inheritance Example

When two or more classes inherit a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherit the Animal class, so there is hierarchical inheritance.



# Java Polymorphism

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

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

# Method Overloading in Java

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having the same name but different in parameters, it is known as **Method Overloading**. If we have to perform only one operation, having the same name of the methods increases the readability of the program.

## Advantage of method overloading

Method overloading *increases the readability of the program*.

### 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

### 1) Method Overloading: changing no. of arguments

In this example, we have created two methods, the first add() method performs addition of two numbers and the second add method performs addition of three numbers.

In this example, we are creating [static methods](https://www.javatpoint.com/static-keyword-in-java) so that we don't need to create an instance for calling methods.

**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));

}}

### 2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differ in [data type](https://www.javatpoint.com/java-data-types). The first add method receives two integer arguments and the second add method receives two double arguments.

**class** Adder{

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

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

}

**class** TestOverloading2{

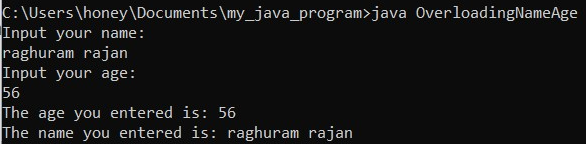
**public** **static** **void** main(String[] args){

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

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

}}





# Method Overriding in Java

If a subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent classes, it is known as method overriding.

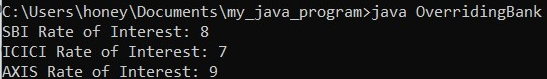
### Usage of Java Method Overriding

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

#### Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Method overriding is one of the ways by which java achieves [Run Time Polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/).The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.



# **Abstraction in Java**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

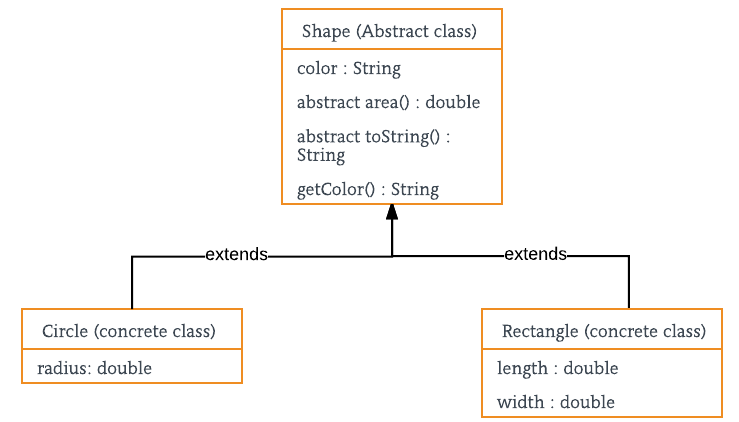
In java, abstraction is achieved by [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) and [abstract classes](https://www.geeksforgeeks.org/abstract-classes-in-java/). We can achieve 100% abstraction using interfaces.

**Abstract classes and Abstract methods :**

1. An abstract class is a class that is declared with an [abstract keyword.](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method defined abstract must always be redefined in the subclass, thus making [overriding](https://www.geeksforgeeks.org/overriding-in-java/) compulsory OR either make the subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the [*new operator*](https://www.geeksforgeeks.org/new-operator-java/).
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

**When to use abstract classes and abstract methods with an example**

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.



# Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

### Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### Points to Remember

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

**Example of abstract class**

1. **abstract** **class** A{}

### Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

1. **abstract** **void** printStatus();//no method body and abstract

## **Rules of Abstract Method**

1. Abstract methods don’t have a body, they just have method signatures as shown above.

2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn’t need to have an abstract method compulsory.

3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of the abstract parent class or it has to be declared abstract as well.

# Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve* [*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also represents the IS-A relationship.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have default and static methods in an interface.

Since Java 9, we can have private methods in an interface.

## Why use the Java interface?

There are mainly three reasons to use interfaces. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

## How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

### Syntax:

**interface** <interface\_name>{

// declare constant fields

// declare methods that abstract

// by default.

}

#### The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



#### Notes on Interfaces:

* Like abstract classes, interfaces cannot be used to create objects
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public
* Interface attributes are by default public, static and final
* An interface cannot contain a constructor (as it cannot be used to create objects)

#### Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma (see example below).

# Difference between abstract class and interface

| Abstract class | Interface |
| --- | --- |
| 1) Abstract class can have abstract and non-abstract methods. | Interface can have only abstract methods. Since Java 8, it can have default and static methods also. |
| 2) Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| 3) Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| 4) Abstract class can provide the implementation of interface. | Interfaces can't provide the implementation of abstract classes. |
| 5) The abstract keyword is used to declare abstract class. | The interface keyword is used to declare the interface. |
| 6) An abstract class can extend another Java class and implement multiple Java interfaces. | An interface can extend another Java interface only. |
| 7) An abstract class can be extended using the keyword "extends". | An interface can be implemented using the keyword "implements". |
| 8) A Java abstract class can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)Example:  public abstract class Shape{  public abstract void draw();  } | Example:  public interface Drawable{  void draw();  } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

//Creating interface that has 4 methods

**interface** A{

**void** a();//bydefault, public and abstract

**void** b();

**void** c();

**void** d();

}

//Creating abstract class that provides the implementation of one method of A interface

**abstract** **class** B **implements** A{

**public** **void** c(){System.out.println("I am C");}

}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods

**class** M **extends** B{

**public** **void** a(){System.out.println("I am a");}

**public** **void** b(){System.out.println("I am b");}

**public** **void** d(){System.out.println("I am d");}

}

//Creating a test class that calls the methods of A interface

**class** Test5{

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

A a=**new** M();

a.a();

a.b();

a.c();

a.d();

}}

# Encapsulation in Java

**Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed with several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

It is a way to achieve **data hiding** in Java because other classes will not be able to access the data through the private data members.

### Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

1. //A Java class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. **package** com.javatpoint;
4. **public** **class** Student{
5. //private data member
6. **private** String name;
7. //getter method for name
8. **public** String getName(){
9. **return** name;
10. }
11. //setter method for name
12. **public** **void** setName(String name){
13. **this**.name=name
14. }
15. }
16. //A Java class to test the encapsulated class.
17. **package** com.javatpoint;
18. **class** Test{
19. **public** **static** **void** main(String[] args){
20. //creating instance of the encapsulated class
21. Student s=**new** Student();
22. //setting value in the name member
23. s.setName("vijay");
24. //getting value of the name member
25. System.out.println(s.getName());
26. }
27. }

#### Example 2:

class Democ

{

private int acc\_no;

private String name;

private float amount;

public void setaccount(int acc\_no)

{this.acc\_no =acc\_no;}

public int getaccount()

{return acc\_no;}

public void setname(String name)

{this.name =name;}

public String getname()

{return name;}

public void setamount(float amount)

{this.amount =amount;}

public float getamount()

{return amount;}

}

class EncapsulationProg

{

public static void main(String args[])

{

Democ d = new Democ();

d.setaccount(2001);

d.setname("rafael dos anjos");

d.setamount(560000);

System.out.println("Account number is "+d.getaccount()+" name is "+d.getname()+" amount is "+d.getamount());

}

}

# Java Arrays

An array is a collection of similar types of elements which have contiguous memory locations. Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Arrays in Java are index-based, the first element of the array is stored at the 0th index, the 2nd element is stored on the 1st index and so on. Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, an array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



### Advantages

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an index position.

### Disadvantages

* **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow in size at runtime. To solve this problem, a collection framework is used in Java which grows automatically.

### Types of Array in java

There are two types of arrays.

* Single Dimensional Array
* Multidimensional Array

**Syntax to Declare an Array in Java**

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

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

String[] cars;

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

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

To create an array of integers, you could write:

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

**Instantiation of an Array in Java**

1. arrayRefVar=**new** datatype[size];



# Java ArrayList

Java **ArrayList** class uses a *dynamic* [*array*](https://www.javatpoint.com/array-in-java) for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array.

ArrayList is a part of [collection framework](https://www.geeksforgeeks.org/collections-in-java-2/) and is present in java.util package. It provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This class is found in [java.util](https://www.geeksforgeeks.org/java-util-package-java/) package.

It inherits the AbstractList class and implements [List interface](https://www.javatpoint.com/java-list).

Syntax:

ArrayList<String> list=**new** ArrayList<String>();//creating new generic arraylist 

**Output:**

[1, 2, 3, 4, 5]

[1, 2, 3, 5]

1 2 3 5

**1. Adding Elements:** In order to add an element to an ArrayList, we can use the [add() method](https://www.geeksforgeeks.org/java-util-arraylist-add-method-java/?ref=rp). This method is overloaded to perform multiple operations based on different parameters. They are:

* **add(Object):** This method is used to add an element at the end of the ArrayList.
* **add(int index, Object):** This method is used to add an element at a specific index in the ArrayList.



**Output:**[Geeks, For, Geeks]

**2. Changing Elements:** After adding the elements, if we wish to change the element, it can be done using the [set()](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples/) method. Since an ArrayList is indexed, the element which we wish to change is referenced by the index of the element. Therefore, this method takes an index and the updated element which needs to be inserted at that index.



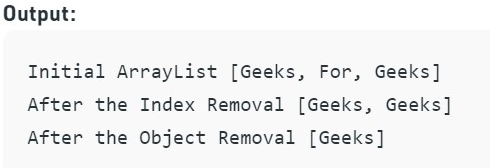
**Output:**

Initial ArrayList [Geeks, Geeks, Geeks]

Updated ArrayList [Geeks, For, Geeks]

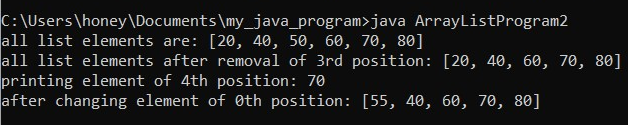
**3. Removing Elements:** In order to remove an element from an ArrayList, we can use the [remove() method](https://www.geeksforgeeks.org/arraylist-linkedlist-remove-methods-java-examples/). This method is overloaded to perform multiple operations based on different parameters. They are:

* **remove(Object):** This method is used to simply remove an object from the ArrayList. If there are multiple such objects, then the first occurrence of the object is removed.
* **remove(int index):** Since an ArrayList is indexed, this method takes an integer value which simply removes the element present at that specific index in the ArrayList. After removing the element, all the elements are moved to the left to fill the space and the indices of the objects are updated.



**4. Iterating the ArrayList:** There are multiple ways to iterate through the ArrayList. The most famous ways are by using the basic [for loop](https://www.geeksforgeeks.org/loop-java-important-points/) in combination with a [get() method](https://www.geeksforgeeks.org/arraylist-get-method-java-examples/) to get the element at a specific index and the [advanced for loop](https://www.geeksforgeeks.org/for-each-loop-in-java/).





# **Conversion of Array To ArrayList**

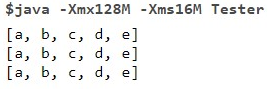
# We can convert an array to arraylist using following ways.

# **Using Arrays.asList() method** - Pass the required array to this method and get a **List** object and pass it as a parameter to the constructor of the **ArrayList** class.

# **Collections.addAll() method** - Create a new list before using this method and then add array elements using this method to existing list.

# **Iteration method** - Create a new list. Iterate the array and add each element to the list.

# 



# 

# Collections in Java

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).

#### What is a framework in Java

* It provides readymade architecture.
* It represents a set of classes and interfaces.
* It is optional.

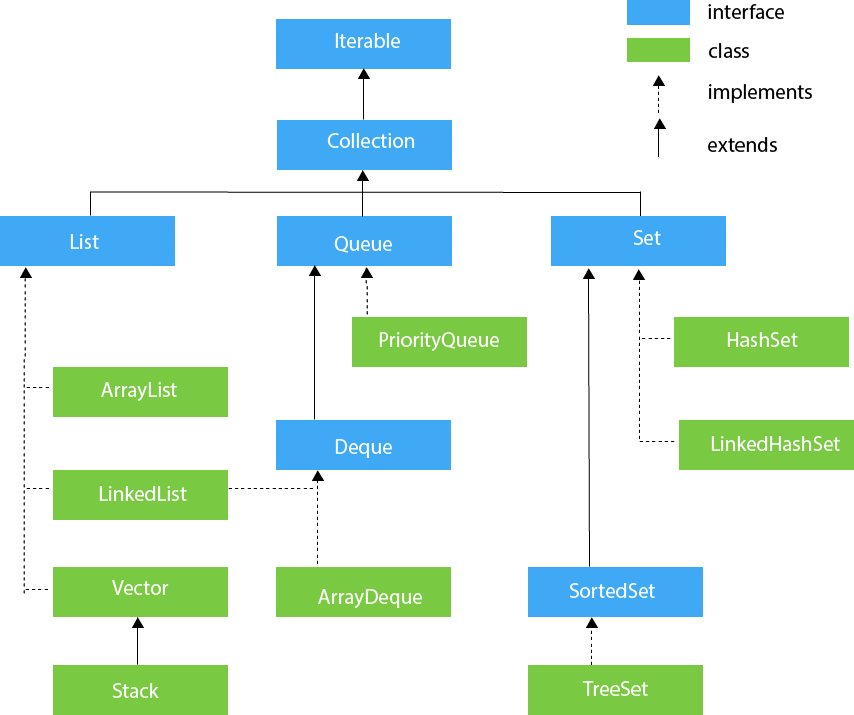
#### What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

### Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the [classes](https://www.javatpoint.com/object-and-class-in-java) and [interfaces](https://www.javatpoint.com/interface-in-java) for the Collection framework.



### Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

| No. | Method | Description |
| --- | --- | --- |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

### Iterator interface

| Iterator interface provides the facility of iterating the elements in a forward direction only. |
| --- |

#### Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

| No. | Method | Description |
| --- | --- | --- |
| 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |

## Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

## Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

## ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

1. **import** java.util.\*;
2. **class** TestJavaCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

Output:

Ravi

Vijay

Ravi

Ajay

## LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection2{
3. **public** **static** **void** main(String args[]){
4. LinkedList<String> al=**new** LinkedList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }



## Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection3{
3. **public** **static** **void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }



## Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection4{
3. **public** **static** **void** main(String args[]){
4. Stack<String> stack = **new** Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }



## Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

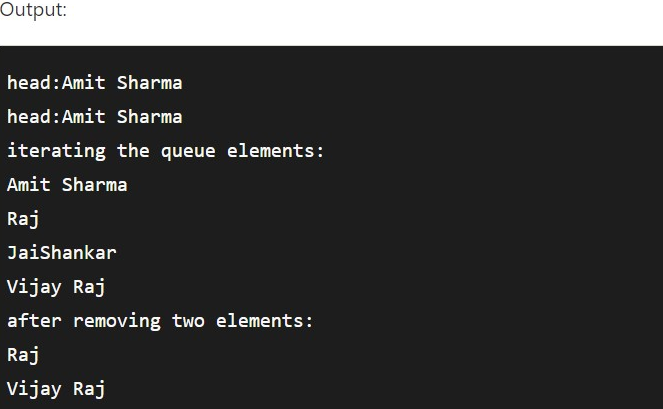
1. Queue<String> q1 = **new** PriorityQueue();
2. Queue<String> q2 = **new** ArrayDeque();

## PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection5{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. **while**(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }



## Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = **new** ArrayDeque();

## ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection6{
3. **public** **static** **void** main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }



## Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

## HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection7{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }



## LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection8{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> set=**new** LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }



## SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

1. SortedSet<data-type> set = **new** TreeSet();

## TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=**new** TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }



# Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

### HashMap class declaration

Let's see the declaration for java.util.HashMap class.

1. **public** **class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializable

### HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

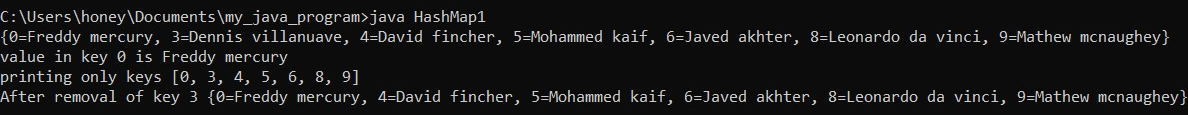
1. **import** java.util.\*;
2. **public** **class** HashMapExample1{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap
5. map.put(1,"Mango"); //Put elements in Map
6. map.put(2,"Apple");
7. map.put(3,"Banana");
8. map.put(4,"Grapes");
10. System.out.println("Iterating Hashmap...");
11. **for**(Map.Entry m : map.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }



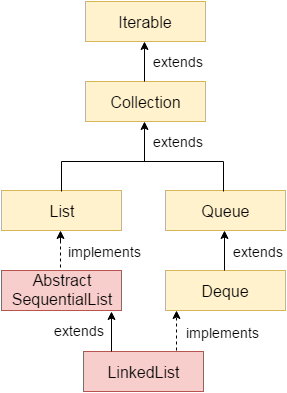
In this example, we are storing Integer as the key and String as the value, so we are using HashMap<Integer,String> as the type. The put() method inserts the elements in the map.

To get the key and value elements, we should call the getKey() and getValue() methods. The Map.Entry interface contains the *getKey()* and *getValue()* methods. But, we should call the entrySet() method of the Map interface to get the instance of Map.Entry.





# Java LinkedList class



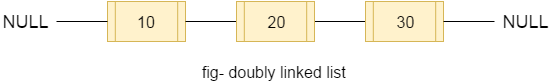
Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.

### Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.



### LinkedList class declaration

Let's see the declaration for java.util.LinkedList class.

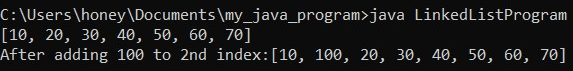
1. **public** **class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

### Java LinkedList Example

1. **import** java.util.\*;
2. **public** **class** LinkedList1{
3. **public** **static** **void** main(String args[]){
5. LinkedList<String> al=**new** LinkedList<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }







# Exception Handling in Java

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

## What is Exception in Java?

**Dictionary Meaning:** Exception is an abnormal condition.

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

## What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

### Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

### 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.

## Java Exception Keywords

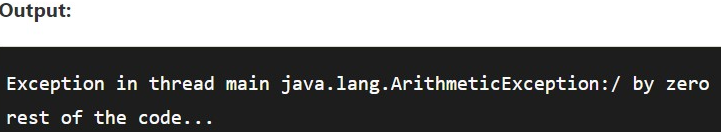
Java provides five keywords that are used to handle the exception. The following table describes each.

| 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. |

## Java Exception Handling Example

**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. }



## Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

### 1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

1. **int** a=50/0;//ArithmeticException

### 2) A scenario where NullPointerException occurs

If we have a null value in any [variable](https://www.javatpoint.com/java-variables), performing any operation on the variable throws a NullPointerException.

1. String s=**null**;
2. System.out.println(s.length());//NullPointerException

### 3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a [string](https://www.javatpoint.com/java-string) variable that has characters; converting this variable into digit will cause NumberFormatException.

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

### 4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

1. **int** a[]=**new** **int**[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException