| **UNIT-I** | **INTRODUCTION TO OOPS** | | **9** |
| --- | --- | --- | --- |
| Introduction - need of object oriented programming - principles of object oriented languages - procedural languages Vs. OOPs - applications of OOPs - history of JAVA - java virtual machine  java features - program structures - installation of JDK1.6 | | | |
| **UNIT-II** | **PROGRAMMING CONSTRUCTS** | | **9** |
| Variables - primitive data types – identifiers - naming conventions – keywords – literals – operators – binary - unary and ternary – expression - precedence rules and associativity - primitive type conversion and casting - flow of control – arrays- command line arguments. | | | |
| **UNIT-III** | **INTERFACE AND EXCEPTIONS** | | **9** |
| Types of inheritance – interface - interface vs abstract classes - packages-creating packages - access protection -java.lang package - exception handling techniques - user defined exception - exception encapsulation – enrichment – assertions | | | |
| **UNIT-IV** | **MULTITHREADING** | | **9** |
| The main thread - creation of new threads - thread priority – multithreading - using is Alive () and join () – Synchronization - suspending and resuming threads - communication between threads - reading and writing data. | | | |
| **UNIT-V** | **CONCURRENT AND NETWORK PROGRAMMING** | | **9** |
| Threads – Thread states – Interrupting threads – Thread communication - Networking basics – Java and the Net - InetAddress –- TCP/IP Server Sockets – Remote Method Invocation – A simple client/server application using RMI. | | | |
| Total hours to be taught | | **L:45 HOURS** | |

| **TEXT BOOK :** | |
| --- | --- |
| 1 | Herbert Schildt, “Java The Complete Reference”, TMH, 2014. |
| **REFERENCES:** | |
| 1 | Elliotte Rustry Harold, “Java Network Programming”, O’Reilly, 2014. |
| 2 | Peter Haggar, “Practical Java Programming Language Guide”, Addison Wesley, 2000. |
| 3 | Daniel Liang Y, “An Introduction to Java Programming”, PHI pvt ltd, 2003. |
| 4 | Coursera, Java Programming: Solving Problems with Software, https://www.coursera.org/learn/java-programming |

**UNIT - 1**

**Introduction to OOPS :**

OOP stands for Object-Oriented Programming. In Java, everything is based on the object. Java has a root class called Object from which the entire functionality of Java is derived.

OOP language supports the following features:

• Classes

• Encapsulation

• Abstraction

• Inheritance

• Polymorphism

**Need of object oriented programming:**

Object-oriented programming (OOP) is a programming paradigm that allows you to package together data states and functionality to modify those data states, while keeping the details hidden away. As a result, code with OOP design is flexible, modular, and abstract. This makes it particularly useful when you create larger programs.

**Principles of Object Oriented Languages:**

Object-Oriented Principles mainly include the 4 pillars that together make the OOP a very powerful concept. That is –

• Abstraction

• Encapsulation

• Inheritance

• Polymorphism

**OOP vs Procedural Programming :**

Procedural programming is based on a sequential execution of instructions. The algorithm is based on data and functions, and the programmer has access to both of these entities and the independence to modify either of them. Since the programming is step-by-step, in a really long program it becomes tough to back and follow up on the developments. Some of the popular OOP languages are: JAVA, C#. NET and VB.NET.

Object-Oriented Programming, or OOP, is made of a number of entities referred to as objects. An object has a behaviour and a purpose associated with it. An object cannot modify the data of another object directly. To get information about an object, the other object sends messages and requests for the data. Some of the popular procedural languages are: Perl, C, VB, FORTRAN, and Basic

**OOP Applications in Java :**

Object-Oriented Programming (OOP) is a programming paradigm that focuses on the use of objects to design and develop software. Java is a popular programming language that supports OOP concepts, and here are some of the applications of OOPs in Java:

**Abstraction:** OOP allows developers to abstract complex real-world concepts into simpler, more manageable objects. This allows for a more modular and flexible software design, where individual objects can be modified and updated independently of the rest of the system.

**Encapsulation:** OOP allows developers to encapsulate data and behaviour within an object. This protects the internal state of an object from outside interference, ensuring that the object operates correctly and consistently.

**Inheritance:** OOP supports inheritance, which allows developers to create new classes that inherit properties and behaviour from existing classes. This saves time and effort when creating new classes, as developers can reuse code that has already been written.

**Polymorphism:** OOP supports polymorphism, which allows developers to create objects that can take on multiple forms. This means that a single method can be used to operate on different objects, simplifying code and making it more reusable.

## What is Java?

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

**Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

## Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, java.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

## Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

#### 1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

#### 2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, [Servlet](https://www.javatpoint.com/servlet-tutorial), [JSP](https://www.javatpoint.com/jsp-tutorial), [Struts](https://www.javatpoint.com/struts-2-tutorial), [Spring](https://www.javatpoint.com/spring-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. technologies are used for creating web applications in Java.

#### 3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, [EJB](https://www.javatpoint.com/ejb-tutorial) is used for creating enterprise applications.

#### 4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

**History of Java :**

Java is a general-purpose, high-level programming language that was created by James Gosling at Sun Microsystems (later acquired by Oracle) in the mid-1990s. Here is a brief history of Java:

• Creation: In 1991, James Gosling, Mike Sheridan, and Patrick Naughton created a new programming language called "Oak" for use in small consumer electronics devices. Oak was later renamed "Java" and was released publicly in 1995.

• Early years: In the mid-1990s, Java gained popularity as a programming language for building interactive web applications. The introduction of the Java applet allowed developers to create rich, interactive content for the web.

• Standardization: In 1997, Sun Microsystems released the first version of the Java Development Kit (JDK), which included the Java Virtual Machine (JVM) and a set of core libraries. This enabled developers to write programs in Java that could run on any platform with a JVM.

• Today, Java remains one of the most popular programming languages in the world, with a large and active developer community. It is widely used for building enterprise applications, mobile apps, games, and web applications, among other things.

**JVM :**

Java Virtual Machine (JVM) is a software that provides a runtime environment for Java programs. It is a virtual machine that runs on top of the physical machine (computer hardware) and is responsible for executing Java bytecode, which is compiled from Java source code.

**Java Features :**

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java buzzwords.

A list of the most important features of the Java language is given below.

• Simple

• Object-Oriented

• Portable

• Platform independent

• Secured

• Robust

• Architecture neutral

• Interpreted

• High Performance

• Multithreaded

• Distributed

• Dynamic

**Structured programming:**

It is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines.

# First Java Program | Hello World Example

In this section, we will learn how to write the simple program of Java. We can write a simple hello Java program easily after installing the JDK.

To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.

### The requirement for Java Hello World Example

For executing any Java program, the following software or application must be properly installed.

* Install the JDK if you don't have installed it, download the JDK and install it.
* Set path of the jdk/bin directory.  Create the Java program
* Compile and run the Java program

### Creating Hello World Example

Let's create the hello java program

**class** Simple{

1. **public** **static** **void** main(String args[]){
2. System.out.println("Hello Java");
3. }
4. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Simple)

Save the above file as Simple.java.

|  |  |
| --- | --- |
| **To compile:** | javac Simple.java |
| **To execute:** | java Simple |

**Output:**

Hello Java

**Compilation Flow:**

When we compile Java program using javac tool, the Java compiler converts the source code into byte code.

## Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

* **class** keyword is used to declare a class in Java.
* **public** keyword is an access modifier that represents visibility. It means it is visible to all.
* **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
* **void** is the return type of the method. It means it doesn't return any value.
* **main** represents the starting point of the program.
* **String[] args** or **String args[]** is used for command line argument. We will discuss it in coming section.
* **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. We will discuss the internal working of System.out.println() statement in the coming section.

**Unit - 2 :**

**Variables and data types:**

Variables are containers for storing data values.

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

• 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

• String - stores text, such as "Hello". String values are surrounded by double quotes

• boolean - stores values with two states: true or false

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** |
| boolean | false | 1 bit |
| char | '\u0000' | 2 byte |
| byte | 0 | 1 byte |
| short | 0 | 2 byte |
| int | 0 | 4 byte |
| long | 0L | 8 byte |
| float | 0.0f | 4 byte |
| double | 0.0d | 8 byte |

**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, total Volume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code

The general rules for naming variables are:

• Names can contain letters, digits, underscores, and dollar signs

• Names must begin with a letter

• Names should start with a lowercase letter and it cannot contain whitespace

• Names can also begin with $ and \_ (but we will not use it in this tutorial)

• Names are case sensitive ("myVar" and "myvar" are different variables)

• Reserved words (like Java keywords, such as int or boolean) cannot be used as names

### 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.

#### 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.

#### 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.

### Example to understand the types of variables in java

1. **public** **class** A
2. {
3. **static** **int** m=100;//static variable
4. **void** method()
5. {
6. **int** n=90;//local variable
7. }
8. **public** **static** **void** main(String args[])
9. {
10. **int** data=50;//instance variable
11. }
12. }//end of class

### Java Variable Example: Add Two Numbers

1. **public** **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. **int** b=10;
5. **int** c=a+b;
6. System.out.println(c);
7. }
8. }

**Java Naming Convention:**

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

**Java Reserved Keywords:**

Java has a set of keywords that are reserved words that cannot be used as variables, methods, classes, or any other identifiers. For example: break, char, double, final, etc,..

**Literal:**

Any constant value which can be assigned to the variable is called literal/constant.

In simple words, Literals in Java is a synthetic representation of boolean, numeric, character, or string data.

**Java Operators:**

Operators are used to perform operations on variables and values.

Java divides the operators into the following groups:

Arithmetic operators

Assignment operators

Comparison operators

Logical operators

Bitwise operators

**Unary Operators:**

A unary operator is an operator that operates on a single operand. An operand can be a value or an expression.

For example: a++

**Binary Operators**

Binary operator operates on two operands. Arithmetic operators are examples of binary operators.

For example: 4+5, 7+1

**Ternary conditional operator:**

the ternary conditional operator. A question mark and a colon separate the three targets of the operation. If the condition preceding the question mark evaluates to true, the expression before the colon is executed and returned. If the condition preceding the question mark evaluates to false, the expression after the colon is executed and returned.

**Java Expressions:**

An expression is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value.

**Java Operators Precedence:**

Operator precedence defines the order in which a given mathematical expression is evaluated. When an expression includes multiple operators then every single part of the given expression is evaluated in a certain order following some rules defined as per operator precedence. The higher precedence is evaluated first and the lowest precedence is evaluated last.

**Java Operator Associativity:**

With the same precedence follow operator associativity defined for their operator group. In Java, operators can either follow left-associative, right-associative, or have no associativity. Operators with left-associative are evaluated from the left to right, operators with right-associative are evaluated from right to the left, and with no associativity, do not follow any predefined order.

**Casting between primitive Java types:**

Changing a value from one data type to a variable of another type is known as data type conversion.

There are two types of casting,

• Primitive Type Casting

• Reference Type Casting

**Primitive Type Casting:**

Casting between primitive types enables you to convert the value of one type to another primitive type is called Primitive Type Casting. This is most commonly occurs with the numeric data types . But boolean primitive type can never be used in a cast. Its values must be either true or false and cannot be used in a casting operation.

**Reference Type Casting:**

Objects of classes also can be cast into objects of other classes when the source and destination classes are related by inheritance and one class is a subclass of the other. The cast can be to its own class type or to one of its subclass or superclass types or interfaces. There are compile-time rules and runtime rules for casting in java. There are two types of Reference Type Casting in Java, they are

• Upcasting

• Downcasting

Up-casting is casting to a super type, while down casting is casting to a subtype. Super casting is always allowed, but sub casting involves a type check and can throw a ClassCastException.

**Control Flow in Java:**

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

### 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. }

Consider the following example in which we have used the **if** statement in the java code.

Student.java

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. **int** x = 10;
4. **int** y = 12;
5. **if**(x+y > 20) {
6. System.out.println("x + y is greater than 20");
7. }
8. }
9. }

**Output:**

x + y is greater than 20

### 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.

**Syntax:**

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

Consider the following example.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. **int** x = 10;
4. **int** y = 12;
5. **if**(x+y < 10) {
6. System.out.println("x + y is less than      10");
7. }   **else** {
8. System.out.println("x + y is greater than 20");
9. }
10. }
11. }

**Output:**

x + y is greater than 20

### 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. }

Consider the following example.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. String city = "Delhi";
4. **if**(city == "Meerut") {
5. System.out.println("city is meerut");
6. }**else** **if** (city == "Noida") {
7. System.out.println("city is noida");
8. }**else** **if**(city == "Agra") {
9. System.out.println("city is agra");
10. }**else** {
11. System.out.println(city);
12. }
13. }
14. }

**Output:**

Delhi

### 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**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. String address = "Delhi, India";
5. **if**(address.endsWith("India")) {
6. **if**(address.contains("Meerut")) {
7. System.out.println("Your city is Meerut");
8. }**else** **if**(address.contains("Noida")) {
9. System.out.println("Your city is Noida");
10. }**else** {
11. System.out.println(address.split(",")[0]);
12. }
13. }**else** {
14. System.out.println("You are not living in India");
15. }
16. }
17. }

**Output:**

Delhi

### Switch Statement:

In Java, [Switch statements](https://www.javatpoint.com/java-switch) are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

* The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
* Cases cannot be duplicate
* Default statement is executed when any of the case doesn't match the value of expression. It is optional.
* Break statement terminates the switch block when the condition is satisfied.  
  It is optional, if not used, next case is executed.
* While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

1. **switch** (expression){
2. **case** value1:
3. statement1;
4. **break**;
5. .
6. .
7. .
8. **case** valueN:
9. statementN;
10. **break**;
11. **default**:
12. **default** statement;
13. }

Consider the following example to understand the flow of the switch statement.

**Student.java**

1. **public** **class** Student **implements** Cloneable {
2. **public** **static** **void** main(String[] args) {
3. **int** num = 2;
4. **switch** (num){
5. **case** 0:
6. System.out.println("number is 0");
7. **break**;
8. **case** 1:
9. System.out.println("number is 1");
10. **break**;
11. **default**:
12. System.out.println(num);
13. }
14. }
15. }

**Output:**

2

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

### Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

Let's understand the loop statements one by one.

### Java for loop

In Java, [for loop](https://www.javatpoint.com/java-for-loop) is similar to [C](https://www.javatpoint.com/c-programming-language-tutorial) and [C++](https://www.javatpoint.com/cpp-tutorial). It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

1. **for**(initialization, condition, increment/decrement) {
2. //block of statements
3. }

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

**Calculation.java**

1. **public** **class** Calculattion {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** sum = 0;
5. **for**(**int** j = 1; j<=10; j++) {
6. sum = sum + j;
7. }
8. System.out.println("The sum of first 10 natural numbers is " + sum);
9. }
10. }

**Output:**

The sum of first 10 natural numbers is 55

### Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

1. **for**(data\_type var : array\_name/collection\_name){
2. //statements
3. }

Consider the following example to understand the functioning of the for-each loop in Java.

**Calculation.java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. String[] names = {"Java","C","C++","Python","JavaScript"};
5. System.out.println("Printing the content of the array names:\n");
6. **for**(String name:names) {
7. System.out.println(name);
8. }
9. }
10. }

**Output:**

Printing the content of the array names:

Java

C

C++

Python

JavaScript

### Java while loop

The [while loop](https://www.javatpoint.com/java-while-loop) is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

1. **while**(condition){
2. //looping statements
3. }

The flow chart for the while loop is given in the following image.



Consider the following example.

**Calculation .java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. **while**(i<=10) {
7. System.out.println(i);
8. i = i + 2;
9. }
10. }
11. }

**Output:**

Printing the list of first 10 even numbers

0

2

4

6

8

10

### Java do-while loop

The [do-while loop](https://www.javatpoint.com/java-do-while-loop) checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

1. **do**
2. {
3. //statements
4. } **while** (condition);

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

**Calculation.java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. **do** {
7. System.out.println(i);
8. i = i + 2;
9. }**while**(i<=10);
10. }
11. }

**Output:**

Printing the list of first 10 even numbers

0

2

4

6

8

10

### Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

### Java break statement

As the name suggests, the [break statement](https://www.javatpoint.com/java-break) is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

**The break statement example with for loop**

Consider the following example in which we have used the break statement with the for loop.

**BreakExample.java**

1. **public** **class** BreakExample {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
5. **for**(**int** i = 0; i<= 10; i++) {
6. System.out.println(i);
7. **if**(i==6) {
8. **break**;
9. }
10. }
11. }
12. }

**Output:**

0

1

2

3

4

5

6

**break statement example with labeled for loop**

**Calculation.java**

1. **public** **class** Calculation {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
5. a:
6. **for**(**int** i = 0; i<= 10; i++) {
7. b:
8. **for**(**int** j = 0; j<=15;j++) {
9. c:
10. **for** (**int** k = 0; k<=20; k++) {
11. System.out.println(k);
12. **if**(k==5) {
13. **break** a;
14. }
15. }
16. }
18. }
19. }

22. }

**Output:**

0

1

2

3

4

5

### Java continue statement

Unlike break statement, the [continue statement](https://www.javatpoint.com/java-continue) doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

1. **public** **class** ContinueExample {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
6. **for**(**int** i = 0; i<= 2; i++) {
8. **for** (**int** j = i; j<=5; j++) {
10. **if**(j == 4) {
11. **continue**;
12. }
13. System.out.println(j);
14. }
15. }
16. }
18. }

**Output:**

0

1

2

3

5

1

2

3

5

2

3

5

**Arrays:**

Normally, an array is a collection of similar type of elements which has contiguous memory location.

**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 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.

### 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 its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

### Types of Array in java

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

## Single Dimensional Array in Java

**Syntax to Declare an Array in Java**

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

**Instantiation of an Array in Java**

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

### Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. **class** Testarray{
4. **public** **static** **void** main(String args[]){
5. **int** a[]=**new** **int**[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. **for**(**int** i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray)

Output:

10

20

70

40

50

## Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

1. **int** a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. **class** Testarray1{
4. **public** **static** **void** main(String args[]){
5. **int** a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
7. **for**(**int** i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray1)

Output:

33

3

4

5

## For-each Loop for Java Array

We can also print the Java array using [**for-each loop**](https://www.javatpoint.com/for-each-loop). The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

1. **for**(data\_type variable:array){
2. //body of the loop
3. }

Let us see the example of print the elements of Java array using the for-each loop.

1. //Java Program to print the array elements using for-each loop
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. //printing array using for-each loop
6. **for**(**int** i:arr)
7. System.out.println(i);
8. }}

Output:

33

3

4

5

## Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. **class** Testarray2{
4. //creating a method which receives an array as a parameter
5. **static** **void** min(**int** arr[]){
6. **int** min=arr[0];
7. **for**(**int** i=1;i<arr.length;i++)
8. **if**(min>arr[i])
9. min=arr[i];
11. System.out.println(min);
12. }
14. **public** **static** **void** main(String args[]){
15. **int** a[]={33,3,4,5};//declaring and initializing an array
16. min(a);//passing array to method
17. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray2)

Output:

3

## Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

1. //Java Program to demonstrate the way of passing an anonymous array
2. //to method.
3. **public** **class** TestAnonymousArray{
4. //creating a method which receives an array as a parameter
5. **static** **void** printArray(**int** arr[]){
6. **for**(**int** i=0;i<arr.length;i++)
7. System.out.println(arr[i]);
8. }
10. **public** **static** **void** main(String args[]){
11. printArray(**new** **int**[]{10,22,44,66});//passing anonymous array to method
12. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAnonymousArray)

Output:

10

22

44

66

## Returning Array from the Method

We can also return an array from the method in Java.

1. //Java Program to return an array from the method
2. **class** TestReturnArray{
3. //creating method which returns an array
4. **static** **int**[] get(){
5. **return** **new** **int**[]{10,30,50,90,60};
6. }
8. **public** **static** **void** main(String args[]){
9. //calling method which returns an array
10. **int** arr[]=get();
11. //printing the values of an array
12. **for**(**int** i=0;i<arr.length;i++)
13. System.out.println(arr[i]);
14. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestReturnArray)

Output:

10

30

50

90

60

## ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

1. //Java Program to demonstrate the case of
2. //ArrayIndexOutOfBoundsException in a Java Array.
3. **public** **class** TestArrayException{
4. **public** **static** **void** main(String args[]){
5. **int** arr[]={50,60,70,80};
6. **for**(**int** i=0;i<=arr.length;i++){
7. System.out.println(arr[i]);
8. }
9. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestArrayException)

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at TestArrayException.main(TestArrayException.java:5)

50

60

70

80

## Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

**Syntax to Declare Multidimensional Array in Java**

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

**Example to instantiate Multidimensional Array in Java**

1. **int**[][] arr=**new** **int**[3][3];//3 row and 3 column

**Example to initialize Multidimensional Array in Java**

1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;

### Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

1. //Java Program to illustrate the use of multidimensional array
2. **class** Testarray3{
3. **public** **static** **void** main(String args[]){
4. //declaring and initializing 2D array
5. **int** arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6. //printing 2D array
7. **for**(**int** i=0;i<3;i++){
8. **for**(**int** j=0;j<3;j++){
9. System.out.print(arr[i][j]+" ");
10. }
11. System.out.println();
12. }
13. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray3)

Output:

1 2 3

2 4 5

4 4 5

## Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

1. //Java Program to illustrate the jagged array
2. **class** TestJaggedArray{
3. **public** **static** **void** main(String[] args){
4. //declaring a 2D array with odd columns
5. **int** arr[][] = **new** **int**[3][];
6. arr[0] = **new** **int**[3];
7. arr[1] = **new** **int**[4];
8. arr[2] = **new** **int**[2];
9. //initializing a jagged array
10. **int** count = 0;
11. **for** (**int** i=0; i<arr.length; i++)
12. **for**(**int** j=0; j<arr[i].length; j++)
13. arr[i][j] = count++;
15. //printing the data of a jagged array
16. **for** (**int** i=0; i<arr.length; i++){
17. **for** (**int** j=0; j<arr[i].length; j++){
18. System.out.print(arr[i][j]+" ");
19. }
20. System.out.println();//new line
21. }
22. }
23. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestJaggedArray)

Output:

0 1 2

3 4 5 6

7 8

**Java command-line argument:**

Java command-line argument is an argument i.e. passed at the time of running the Java program. In the command line, the arguments passed from the console can be received in the java program and they can be used as input. The users can pass the arguments during the execution bypassing the command-line arguments inside the main() method.

We need to pass the arguments as space-separated values. We can pass both strings and primitive data types(int, double, float, char, etc) as command-line arguments. These arguments convert into a string array and are provided to the main() function as a string array argument.

When command-line arguments are supplied to JVM, JVM wraps these and supplies them to args[]. It can be confirmed that they are wrapped up in an args array by checking the length of args using args.length.

Internally, JVM wraps up these command-line arguments into the args[ ] array that we pass into the main() function. We can check these arguments using args.length method. JVM stores the first command-line argument at args[0], the second at args[1], the third at args[2], and so on.

### Simple example of command-line argument in java

|  |
| --- |
| In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt. |

1. **class** CommandLineExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println("Your first argument is: "+args[0]);
4. }
5. }
6. compile by > javac CommandLineExample.java
7. run by > java CommandLineExample sonoo

### Example of command-line argument that prints all the values

|  |
| --- |
| In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop. |

1. **class** A{
2. **public** **static** **void** main(String args[]){
4. **for**(**int** i=0;i<args.length;i++)
5. System.out.println(args[i]);
7. }
8. }
9. compile by > javac A.java
10. run by > java A sonoo jaiswal 1 3 abc

Output: sonoo

jaiswal

1

3

abc

**UNIT-III**

**INTERFACE AND EXCEPTIONS**

# Inheritance in Java

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPs (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 (so runtime polymorphism 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

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

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.

### Java Inheritance Example



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.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Programmer)

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

## Types of inheritance in java

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

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



#### Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



## 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.

*File: TestInheritance.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** TestInheritance{
8. **public** **static** **void** main(String args[]){
9. Dog d=**new** Dog();
10. d.bark();
11. d.eat();
12. }}

Output:

barking...

eating...

## 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.

*File: TestInheritance2.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

Output:

weeping...

barking...

eating...

## Hierarchical Inheritance Example

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

*File: TestInheritance3.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** meow(){System.out.println("meowing...");}
9. }
10. **class** TestInheritance3{
11. **public** **static** **void** main(String args[]){
12. Cat c=**new** Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}

Output:

meowing...

eating...

## Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

1. **class** A{
2. **void** msg(){System.out.println("Hello");}
3. }
4. **class** B{
5. **void** msg(){System.out.println("Welcome");}
6. }
7. **class** C **extends** A,B{//suppose if it were
9. **public** **static** **void** main(String args[]){
10. C obj=**new** C();
11. obj.msg();//Now which msg() method would be invoked?
12. }
13. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=C)

Compile Time Error

# 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. 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 Java interface?

There are mainly three reasons to use interface. 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:

1. **interface** <interface\_name>{
3. // declare constant fields
4. // declare methods that abstract
5. // by default.
6. }

## Java 8 Interface Improvement

Since Java 8, interface can have default and static methods which is discussed later.

## Internal addition by the compiler

#### The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



#### 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**.



## Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

1. **interface** printable{
2. **void** print();
3. }
4. **class** A6 **implements** printable{
5. **public** **void** print(){System.out.println("Hello");}
7. **public** **static** **void** main(String args[]){
8. A6 obj = **new** A6();
9. obj.print();
10. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=A6)

Output:

Hello

## Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

*File: TestInterface1.java*

1. //Interface declaration: by first user
2. **interface** Drawable{
3. **void** draw();
4. }
5. //Implementation: by second user
6. **class** Rectangle **implements** Drawable{
7. **public** **void** draw(){System.out.println("drawing rectangle");}
8. }
9. **class** Circle **implements** Drawable{
10. **public** **void** draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. **class** TestInterface1{
14. **public** **static** **void** main(String args[]){
15. Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()
16. d.draw();
17. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface1)

Output:

drawing circle

## Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

*File: TestInterface2.java*

1. **interface** Bank{
2. **float** rateOfInterest();
3. }
4. **class** SBI **implements** Bank{
5. **public** **float** rateOfInterest(){**return** 9.15f;}
6. }
7. **class** PNB **implements** Bank{
8. **public** **float** rateOfInterest(){**return** 9.7f;}
9. }
10. **class** TestInterface2{
11. **public** **static** **void** main(String[] args){
12. Bank b=**new** SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface2)

Output:

ROI: 9.15

## Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** show();
6. }
7. **class** A7 **implements** Printable,Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **void** show(){System.out.println("Welcome");}
11. **public** **static** **void** main(String args[]){
12. A7 obj = **new** A7();
13. obj.print();
14. obj.show();
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=A7)

Output:Hello

Welcome

## Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of [class](https://www.javatpoint.com/object-and-class-in-java) because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** print();
6. }
8. **class** TestInterface3 **implements** Printable, Showable{
9. **public** **void** print(){System.out.println("Hello");}
10. **public** **static** **void** main(String args[]){
11. TestInterface3 obj = **new** TestInterface3();
12. obj.print();
13. }
14. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface3)

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

## Interface inheritance

A class implements an interface, but one interface extends another interface.

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable **extends** Printable{
5. **void** show();
6. }
7. **class** TestInterface4 **implements** Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **void** show(){System.out.println("Welcome");}
11. **public** **static** **void** main(String args[]){
12. TestInterface4 obj = **new** TestInterface4();
13. obj.print();
14. obj.show();
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface4)

Output:

Hello

Welcome

## Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

*File: TestInterfaceDefault.java*

1. **interface** Drawable{
2. **void** draw();
3. **default** **void** msg(){System.out.println("default method");}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** TestInterfaceDefault{
9. **public** **static** **void** main(String args[]){
10. Drawable d=**new** Rectangle();
11. d.draw();
12. d.msg();
13. }}

# 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.

### Abstraction in Java

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery. instead of how it does it.

### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

### 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

### Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

1. **abstract** **class** Bike{
2. **abstract** **void** run();
3. }
4. **class** Honda4 **extends** Bike{
5. **void** run(){System.out.println("running safely");}
6. **public** **static** **void** main(String args[]){
7. Bike obj = **new** Honda4();
8. obj.run();
9. }
10. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Honda4)

running safely

### Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

*File: TestAbstraction1.java*

1. **abstract** **class** Shape{
2. **abstract** **void** draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. **class** Rectangle **extends** Shape{
6. **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle1 **extends** Shape{
9. **void** draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. **class** TestAbstraction1{
13. **public** **static** **void** main(String args[]){
14. Shape s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
15. s.draw();
16. }
17. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction1)

drawing circle

### Another example of Abstract class in java

*File: TestBank.java*

1. **abstract** **class** Bank{
2. **abstract** **int** getRateOfInterest();
3. }
4. **class** SBI **extends** Bank{
5. **int** getRateOfInterest(){**return** 7;}
6. }
7. **class** PNB **extends** Bank{
8. **int** getRateOfInterest(){**return** 8;}
9. }
11. **class** TestBank{
12. **public** **static** **void** main(String args[]){
13. Bank b;
14. b=**new** SBI();
15. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16. b=**new** PNB();
17. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestBank)

Rate of Interest is: 7 %

Rate of Interest is: 8 %

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **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**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare 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 keyword "extends". | An **interface** can be implemented using 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(); } |

# Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

## Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



## Simple example of java package

The **package keyword** is used to create a package in java.

1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

## How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

## How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |
| Output:Welcome to packageThe -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder. |

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

## Example of package that import the packagename.\*

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B{
11. **public** **static** **void** main(String args[]){
12. A obj = **new** A();
13. obj.msg();
14. }
15. }

Output:Hello

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

## Example of package by import package.classname

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
8. **package** mypack;
9. **import** pack.A;
11. **class** B{
12. **public** **static** **void** main(String args[]){
13. A obj = **new** A();
14. obj.msg();
15. }

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

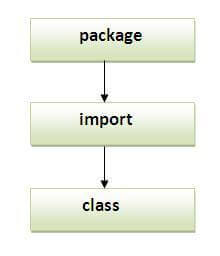
It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

## Example of package by import fully qualified name

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **class** B{
9. **public** **static** **void** main(String args[]){
10. pack.A obj = **new** pack.A();//using fully qualified name
11. obj.msg();
12. }
13. }  **Note: If you import a package, subpackages will not be imported.**

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

#### Note: Sequence of the program must be package then import then class.



## Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

#### The standard of defining package is domain.company.package e.g. com.java.bean or org.sssit.dao.

### Example of Subpackage

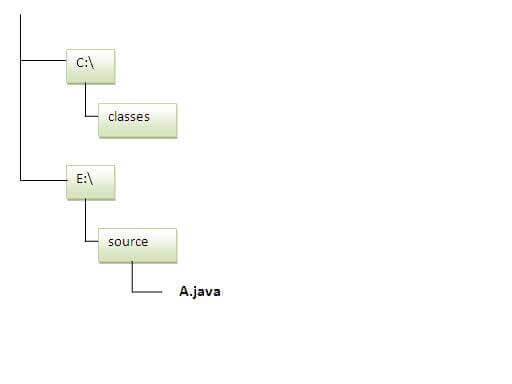
1. **package** com.java.core;
2. **class** Simple{
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java com.java.core.Simple |

Output:Hello subpackage

## How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:



1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

### To Compile:

**e:\sources> javac -d c:\classes Simple.java**

### To Run:

|  |
| --- |
| To run this program from e:\source directory, you need to set classpath of the directory where the class file resides. |
| **e:\sources> set classpath=c:\classes;.;** |
| **e:\sources> java mypack.Simple** |

### Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

**e:\sources> java -classpath c:\classes mypack.Simple**

Output:Welcome to package

### Ways to load the class files or jar files

|  |
| --- |
| There are two ways to load the class files temporary and permanent. |

* Temporary
  + By setting the classpath in the command prompt
  + By -classpath switch
* Permanent
  + By setting the classpath in the environment variables
  + By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

#### Rule: There can be only one public class in a java source file and it must be saved by the public class name.

1. //save as C.java otherwise Compilte Time Error
3. **class** A{}
4. **class** B{}
5. **public** **class** C{}

### How to put two public classes in a package?

|  |
| --- |
| If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example: |

1. //save as A.java
3. **package** javat;
4. **public** **class** A{}
5. //save as B.java
7. **package** javat;
8. **public** **class** B{}

# Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

### Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

### 1) Private

The private access modifier is accessible only within the class.

**Simple example of private access modifier**

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

1. **class** A{
2. **private** **int** data=40;
3. **private** **void** msg(){System.out.println("Hello java");}
4. }
6. **public** **class** Simple{
7. **public** **static** **void** main(String args[]){
8. A obj=**new** A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }

### Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

1. **class** A{
2. **private** A(){}//private constructor
3. **void** msg(){System.out.println("Hello java");}
4. }
5. **public** **class** Simple{
6. **public** **static** **void** main(String args[]){
7. A obj=**new** A();//Compile Time Error
8. }
9. }

#### Note: A class cannot be private or protected except nested class.

### 2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

1. //save by A.java
2. **package** pack;
3. **class** A{
4. **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
9. **class** B{
10. **public** **static** **void** main(String args[]){
11. A obj = **new** A();//Compile Time Error
12. obj.msg();//Compile Time Error
13. }
14. }

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### 3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **protected** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B **extends** A{
11. **public** **static** **void** main(String args[]){
12. B obj = **new** B();
13. obj.msg();
14. }
15. }

Output:Hello

### 4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

**Example of public access modifier**

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

## Package java.lang Description

The most important classes are [**Object**](https://www.javaguides.net/2018/09/object-class-methods-in-java-with-examples.html), which is the root of the class hierarchy, and Class, instances of which represent classes at runtime.  
Frequently it is necessary to represent a value of primitive type as if it were an object. The wrapper classes Boolean, Character, Integer, Short, Byte, Long, Float, and Double serve this purpose.  
  
The classes String, StringBuffer, and StringBuilder similarly provide commonly used operations on character strings.  
  
Class Throwable encompasses objects that may be thrown by the throw statement. Subclasses of Throwable represent errors and exceptions.  
  
Following are a list of classes under *java.lang* package. I explained all the methods with lots of examples from each class. Our suggestion is to do lots of hands experience using this tutorial.  
  
Click on each class will navigate to their respective page.

### >> [java.lang Object Class](https://www.javaguides.net/2018/09/object-class-methods-in-java-with-examples.html)

The Object class, in the [**java.lang**](https://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html) package sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. In this article, you will learn all the Object class methods with examples.

### >> [java.lang Boolean Class](https://www.javaguides.net/2018/08/boolean-wrapper-class-in-java.html)

The Boolean class wraps a value of the primitive type boolean in an object. An object of type Boolean contains a single field whose type is boolean. In this article, you will learn all the Boolean class methods with examples.

### >> [java.lang Byte Class](https://www.javaguides.net/2018/08/byte-wrapper-class-in-java.html)

In this article, you will learn all the Byte class methods with examples. The Byte class wraps a value of primitive type byte in an object. An object of type Byte contains a single field whose type is a byte.

### >> [java.lang Character Class](https://www.javaguides.net/2018/08/character-wrapper-class-in-java.html)

In this article, you will learn all the Character wrapper class methods with examples. The Character class wraps a value of the primitive type char in an object. An object of type Character contains a single field whose type is char.

### >> [java.lang Double Class](https://www.javaguides.net/2018/08/double-wrapper-class-in-java.html)

In this article, you will learn all the Double wrapper class methods with examples. The Double class wraps a value of the primitive type double in an object. An object of type Double contains a single field whose type is double.

### >> [java.lang Float Class](https://www.javaguides.net/2018/08/float-wrapper-class-in-java.html)

In this article, you will learn all the Float wrapper class methods with examples. The Float class wraps a value of primitive type float in an object. An object of type Float contains a single field whose type is a float.

### >> [java.lang Long Class](https://www.javaguides.net/2018/08/long-wrapper-class-in-java.html)

In this article, you will learn all the Long wrapper class methods with examples. The Long class wraps a value of the primitive type long in an object. An object of type Long contains a single field whose type is long.

### >> [java.lang Integer Class](https://www.javaguides.net/2018/08/integer-wrapper-class-in-java.html)

In this article, you will learn all the Integer wrapper class methods with examples. The Integer class wraps a value of the primitive type int in an object. An object of type Integer contains a single field whose type is int.

### >> [java.lang.Number Class](https://www.javaguides.net/2018/12/java-number-class-methods-with-examples.html)

In this tutorial, we will discuss the important and commonly used Number class methods with examples.

### >> [java.lang.Package Class](https://www.javaguides.net/2018/12/java-package-class-methods-with-examples.html)

In this tutorial, we will discuss the important and commonly used Package class methods with examples.

### >> [java.lang Short Class](https://www.javaguides.net/2018/08/short-wrapper-class-in-java.html)

In this article, you will learn all the Short wrapper class methods with examples. The Short class wraps a value of primitive type short in an object. An object of type Short contains a single field whose type is short.

### >> [java.lang Enum Class](https://www.javaguides.net/2018/06/enums-in-java.html)

In this article, we will learn basics and create an Enum datatype in real time projects with lots of examples.

### >> [java.lang String Class](https://www.javaguides.net/2018/08/java-string-class-api-guide.html)

In this article, we will learn all the String methods with examples. As we know Strings are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

### >> [java.lang StringBuilder Class](https://www.javaguides.net/2018/08/java-stringbuilder-class-api-guide.html)

In this article, we will learn all the StringBuilder methods with examples.Java StringBuilder class is used to create a mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

### >> [java.lang StringBuffer Class](https://www.javaguides.net/2018/08/java-stringbuffer-class-api-guide.html)

In this article, we will learn all the StringBuilder methods with examples. Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

### >> [java.lang Thread Class](https://www.javaguides.net/2018/09/thread-class-in-java.html)

In this article, we will learn about Thread Class and it's methods with examples. Thread creates a new thread of execution. It implements the Runnable interface. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

### >> [java.lang ThreadLocal Class](https://www.javaguides.net/2018/09/threadlocal-class-in-java.html)

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share its variables, so the variable is not a thread safe. We can use synchronization for thread safety but if we want to avoid synchronization, we can use ThreadLocal variables.

### >> [java.lang ThreadGroup Class](https://www.javaguides.net/2018/09/threadgroup-class-in-java.html)

In this article, we will learn how to group threads in Java. Java provides a convenient way to group multiple threads in a single object. Java thread group is implemented by java.lang.ThreadGroup class.

### >> [java.lang Throwable Class](https://www.javaguides.net/2018/08/javalangthrowable-class-in-java.html)

The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause.

### >> [java.lang Comparable Interface](https://www.javaguides.net/2018/06/guide-to-comparable-interface.html)

The Comparable interface has a single method called *compareTo()* that you need to implement in order to define how an object compares with the supplied object.

### >> [java.lang Runnable Interface](https://www.javaguides.net/2018/09/runnable-interface-in-java.html)

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The Runnable interface defines a single run() method, which contains a code that executed in the thread. The Runnable object is passed to the Thread constructor.

# Exception Handling in Java

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

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

## 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.

### Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

Do You Know?

|  |
| --- |
| * What is the difference between checked and unchecked exceptions? * What happens behind the code int data=50/0;? * Why use multiple catch block? * Is there any possibility when the finally block is not executed? * What is exception propagation? * What is the difference between the throw and throws keyword? * What are the 4 rules for using exception handling with method overriding? |

## Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



### 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

## Difference between Checked and Unchecked Exceptions

### 1) Checked Exception

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

### 2) Unchecked Exception

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

### 3) Error

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

## 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

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

**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. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=JavaExceptionExample)

**Output:**

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

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

## 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

# Java try-catch block

## Java try block

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

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

### Syntax of Java try-catch

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

### Syntax of try-finally block

1. **try**{
2. //code that may throw an exception
3. }**finally**{}

## Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

## Internal Working of Java try-catch block



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

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

## Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

### Example 1

**TryCatchExample1.java**

1. **public** **class** TryCatchExample1 {
3. **public** **static** **void** main(String[] args) {
5. **int** data=50/0; //may throw exception
7. System.out.println("rest of the code");
9. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample1)

**Output:**

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

# Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

### Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

### Syntax:

1. ....
2. //main try block
3. **try**
4. {
5. statement 1;
6. statement 2;
7. //try catch block within another try block
8. **try**
9. {
10. statement 3;
11. statement 4;
12. //try catch block within nested try block
13. **try**
14. {
15. statement 5;
16. statement 6;
17. }
18. **catch**(Exception e2)
19. {
20. //exception message
21. }
23. }
24. **catch**(Exception e1)
25. {
26. //exception message
27. }
28. }
29. //catch block of parent (outer) try block
30. **catch**(Exception e3)
31. {
32. //exception message
33. }
34. ....

## Java Nested try Example

### Example 1

Let's see an example where we place a try block within another try block for two different exceptions.

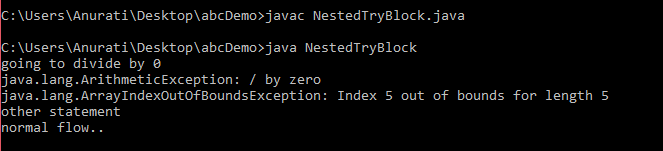
**NestedTryBlock.java**

1. **public** **class** NestedTryBlock{
2. **public** **static** **void** main(String args[]){
3. //outer try block
4. **try**{
5. //inner try block 1
6. **try**{
7. System.out.println("going to divide by 0");
8. **int** b =39/0;
9. }
10. //catch block of inner try block 1
11. **catch**(ArithmeticException e)
12. {
13. System.out.println(e);
14. }

17. //inner try block 2
18. **try**{
19. **int** a[]=**new** **int**[5];
21. //assigning the value out of array bounds
22. a[5]=4;
23. }
25. //catch block of inner try block 2
26. **catch**(ArrayIndexOutOfBoundsException e)
27. {
28. System.out.println(e);
29. }

32. System.out.println("other statement");
33. }
34. //catch block of outer try block
35. **catch**(Exception e)
36. {
37. System.out.println("handled the exception (outer catch)");
38. }
40. System.out.println("normal flow..");
41. }
42. }

**Output:**



When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

### Example 2

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

**NestedTryBlock.java**

1. **public** **class** NestedTryBlock2 {
3. **public** **static** **void** main(String args[])
4. {
5. // outer (main) try block
6. **try** {
8. //inner try block 1
9. **try** {
11. // inner try block 2
12. **try** {
13. **int** arr[] = { 1, 2, 3, 4 };
15. //printing the array element out of its bounds
16. System.out.println(arr[10]);
17. }
19. // to handles ArithmeticException
20. **catch** (ArithmeticException e) {
21. System.out.println("Arithmetic exception");
22. System.out.println(" inner try block 2");
23. }
24. }
26. // to handle ArithmeticException
27. **catch** (ArithmeticException e) {
28. System.out.println("Arithmetic exception");
29. System.out.println("inner try block 1");
30. }
31. }
33. // to handle ArrayIndexOutOfBoundsException
34. **catch** (ArrayIndexOutOfBoundsException e4) {
35. System.out.print(e4);
36. System.out.println(" outer (main) try block");
37. }
38. **catch** (Exception e5) {
39. System.out.print("Exception");
40. System.out.println(" handled in main try-block");
41. }
42. }
43. }

# Java finally block

**Java finally block** is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

### Flowchart of finally block



#### Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

## Why use Java finally block?

* finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

## Usage of Java finally

Let's see the different cases where Java finally block can be used.

### Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

**TestFinallyBlock.java**

1. **class** TestFinallyBlock {
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //below code do not throw any exception
5. **int** data=25/5;
6. System.out.println(data);
7. }
8. //catch won't be executed
9. **catch**(NullPointerException e){
10. System.out.println(e);
11. }
12. //executed regardless of exception occurred or not
13. **finally** {
14. System.out.println("finally block is always executed");
15. }
17. System.out.println("rest of phe code...");
18. }
19. }

**Output:**



### Case 2: When an exception occurr but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

**TestFinallyBlock1.java**

1. **public** **class** TestFinallyBlock1{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside the try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
12. //cannot handle Arithmetic type exception
13. //can only accept Null Pointer type exception
14. **catch**(NullPointerException e){
15. System.out.println(e);
16. }
18. //executes regardless of exception occured or not
19. **finally** {
20. System.out.println("finally block is always executed");
21. }
23. System.out.println("rest of the code...");
24. }
25. }

# Java Custom Exception

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

In this section, we will learn how custom exceptions are implemented and used in Java programs.

## Why use custom exceptions?

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

* To catch and provide specific treatment to a subset of existing Java exceptions.
* Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Consider the following example, where we create a custom exception named WrongFileNameException:

1. **public** **class** WrongFileNameException **extends** Exception {
2. **public** WrongFileNameException(String errorMessage) {
3. **super**(errorMessage);
4. }
5. }

#### Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

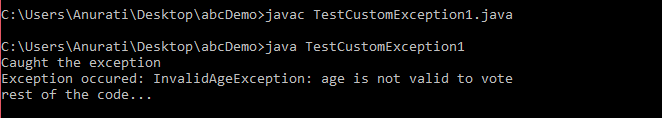
### Example 1:

Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

**TestCustomException1.java**

1. // class representing custom exception
2. **class** InvalidAgeException  **extends** Exception
3. {
4. **public** InvalidAgeException (String str)
5. {
6. // calling the constructor of parent Exception
7. **super**(str);
8. }
9. }
11. // class that uses custom exception InvalidAgeException
12. **public** **class** TestCustomException1
13. {
15. // method to check the age
16. **static** **void** validate (**int** age) **throws** InvalidAgeException{
17. **if**(age < 18){
19. // throw an object of user defined exception
20. **throw** **new** InvalidAgeException("age is not valid to vote");
21. }
22. **else** {
23. System.out.println("welcome to vote");
24. }
25. }
27. // main method
28. **public** **static** **void** main(String args[])
29. {
30. **try**
31. {
32. // calling the method
33. validate(13);
34. }
35. **catch** (InvalidAgeException ex)
36. {
37. System.out.println("Caught the exception");
39. // printing the message from InvalidAgeException object
40. System.out.println("Exception occured: " + ex);
41. }
43. System.out.println("rest of the code...");
44. }
45. }

**Output:**



### Example 2:

**TestCustomException2.java**

1. // class representing custom exception
2. **class** MyCustomException **extends** Exception
3. {
5. }
7. // class that uses custom exception MyCustomException
8. **public** **class** TestCustomException2
9. {
10. // main method
11. **public** **static** **void** main(String args[])
12. {
13. **try**
14. {
15. // throw an object of user defined exception
16. **throw** **new** MyCustomException();
17. }
18. **catch** (MyCustomException ex)
19. {
20. System.out.println("Caught the exception");
21. System.out.println(ex.getMessage());
22. }
24. System.out.println("rest of the code...");
25. }
26. }

# 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 of 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.

### Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

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

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

### 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.

*File: Student.java*

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.javat;
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. }

*File: Test.java*

1. //A Java class to test the encapsulated class.
2. **package** com.javat;
3. **class** Test{
4. **public** **static** **void** main(String[] args){
5. //creating instance of the encapsulated class
6. Student s=**new** Student();
7. //setting value in the name member
8. s.setName("vijay");
9. //getting value of the name member
10. System.out.println(s.getName());
11. }
12. }

Compile By: javac -d . Test.java

Run By: java com.javatTest

Output:

vijay

### Read-Only class

1. //A Java class which has only getter methods.
2. **public** **class** Student{
3. //private data member
4. **private** String college="AKG";
5. //getter method for college
6. **public** String getCollege(){
7. **return** college;
8. }
9. }

Now, you can't change the value of the college data member which is "AKG".

1. s.setCollege("KITE");//will render compile time error

### Write-Only class

1. //A Java class which has only setter methods.
2. **public** **class** Student{
3. //private data member
4. **private** String college;
5. //getter method for college
6. **public** **void** setCollege(String college){
7. **this**.college=college;
8. }
9. }

Now, you can't get the value of the college, you can only change the value of college data member.

1. System.out.println(s.getCollege());//Compile Time Error, because there is no such method
2. System.out.println(s.college);//Compile Time Error, because the college data member is private.
3. //So, it can't be accessed from outside the class

### Another Example of Encapsulation in Java

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

*File: Account.java*

1. //A Account class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. **class** Account {
4. //private data members
5. **private** **long** acc\_no;
6. **private** String name,email;
7. **private** **float** amount;
8. //public getter and setter methods
9. **public** **long** getAcc\_no() {
10. **return** acc\_no;
11. }
12. **public** **void** setAcc\_no(**long** acc\_no) {
13. **this**.acc\_no = acc\_no;
14. }
15. **public** String getName() {
16. **return** name;
17. }
18. **public** **void** setName(String name) {
19. **this**.name = name;
20. }
21. **public** String getEmail() {
22. **return** email;
23. }
24. **public** **void** setEmail(String email) {
25. **this**.email = email;
26. }
27. **public** **float** getAmount() {
28. **return** amount;
29. }
30. **public** **void** setAmount(**float** amount) {
31. **this**.amount = amount;
32. }
34. }

# Create a Java Method Enrichment Action

* Java Method Enrichment Examples

Create an event policy with a Java method enrichment action to run a Java method and enrich an event based on the method output values. The Java method must return a comma-separated list of properties and returned values as follows for the enrichment to work:

*propertyname*

,

*value*

,

*propertyname*

,

*value*

...

Java method enrichments require the following information:

* Method name and class path
* Values to use for the method parameters based on event properties
* Script output values to assign as enrichment output values

The method must exist on the SA Manager and on every connector system to which you want to deploy the event policy.

**Follow these steps:**

1. Create an event policy based on a search pattern, and select Enrich Event as the action type.

The Enrichment Configuration page opens.

1. Select Java method in the Type drop-down list and enter information in the following fields:

The text at the bottom of the page indicates if any required information is missing.

* + **Java Class Path**

Defines the full class path and jar file of the method to use for the enrichment.

**Example:**

<C:\Program Files\CA\SOI\lib\ivy\em.event-plus-catalog-4.2.0.jar>

* + **Class Name**

Defines the class name of the Java method, including the package, to use for the enrichment.

**Example:**

com.ca.eventplus.catalog.methods.CMDBEnrich

* + **User**

(Optional) Defines the user name to run the Java method, if necessary.

* + **Password**

(Optional) Defines the password for the specified Java method user name, if necessary.

If the method requires user authentication in its parameters, enter the credentials here and reference them on the following page to ensure that the data is protected.

* + **Method**

Defines the name of the Java method to run from the referenced class.

**Example:**

 performCMDBEnrichment\_v2

1. (Optional) Click Test.

A confirmation dialog opens.

1. (Optional) Click Yes.

The Java method connection is verified. The Configuration Test Result dialog indicates whether the connection was successful.

If you have to change this information after deploying the policy, restart the CA SAM Integration Services service on the connector system to ensure that the change takes effect. For information about how to configure enrichment value caching, see Configure Enrichment Cache Timeout.

1. Click Next.

The Enrichment Policy page opens. Right-click each column on this page for additional help information.

1. Enter the following in the Parameter Configuration table to determine how the input parameters to the enrichment process are assigned according to method parameter values and event properties:
   * **Input Parameter**

Defines placeholder names for each required method input parameter. The enrichment always reads the parameters sequentially; therefore, the names that you enter for each parameter can be anything (param1, param2, and so on). Create an entry for each required input parameter to ensure that the method runs successfully.

* + **Assigned Value**

Defines the event property or other value to use for the corresponding method parameter value. Use the right-click menu to assign the value of a property from any matching event pattern. The value for each method parameter can take any of the following forms:

* + - A full event property
    - Multiple combined event properties
    - Part of an event property
    - Modified event properties

Use the right-click menu to add provided functions to perform common data conversions on the search value to use for each method parameter.

To enter user credentials, use the following substitution characters to reference the credentials entered on the previous page:

${user}

${password}

Entering a password value manually on this page creates an unencrypted record of the password.

1. The Preview cell displays the result of the entered value based on the selected event in the Event Log table. You must run an event search before creating the policy to get its results in the Event Log table for previewing enrichment values based on existing event content.
2. Include all required parameters for the method to run. If the Java method does not run successfully based on the entered parameters or does not return a comma-separated list of properties and values, the enrichment does not occur for that event.
3. Enter the following in the Enrichment Property Assignment table to specify how enrichment output values are assigned to event properties, and click Next:
   * **Assigned Value**

Defines the Java method output property values to assign to the event properties in the Event Property column. This value determines the property value to use for the enrichment from the comma-separated list of properties and values that the method returns.

References to output properties must be in the following format: ${

*propertyname*

}, where propertyname is the name of the property in the comma-separated output list whose value you want to return. For example, for a method that returns the string 'user,

*value*

,role,

*value*

,department,

*value*

', ${role} uses the returned value from the role output property for the enrichment. Any values entered without this format appear directly in the event as written. You can add enrichments to as many event properties as necessary.

You can change the names of the User Attribute properties if you want them to accurately represent the enrichment properties that you assign to them. However, these properties appear under their original names in the Event Policy dialog, even if you renamed them. Assigning values to these original names properly displays the values under the renamed properties in the Operations Console.

The method property-based value can be a single property value, multiple values, or a modified property value. Use the right-click menu to add provided functions to perform common data conversions on the enrichment value before assigning it to the specified property. The return value cannot contain an embedded comma.

Only the properties that support enrichment value assignment appear in the Event Property column.

1. The Select Data Sources page opens.
2. Save or deploy the policy.

Java Method Enrichment Examples

The following examples show how you can use the enrich event action to enrich events with information from a Java method when search patterns match. The examples uses CA CMDB and CA Spectrum enrichments provided with the Mid-tier connector. The enrichments could be useful in CA SOI for situations such as the following:

* You want to use CI properties in CA CMDB or CA Spectrum in alert queue criteria or for escalation policy
* You are using custom properties in CA CMDB or CA Spectrum that are not imported into CA SOI
* You want to use information from CA CMDB or CA Spectrum in alerts or CIs that are not managed in CA CMDB or CA Spectrum
* You want to parse partial information from a property for use in a different context

The enrichments use .jar files that are only available with the Mid-tier connector. Deploy these provided enrichments on the Mid-tier connector only.

**Example: Enrich events with location information from CA CMDB**

This example enriches events with location information stored in CA CMDB. The information could help you create alert queues by location or add location-based criteria to escalation policy.

* Select Java Method on the Enrichment Configuration page, and select CMDB in the Templates drop-down list.
* Use the User and Password fields to enter valid credentials for the CA CMDB server, and leave the default values in all other fields.
* Do the following on the Enrichment Policy Configuration page:
  + Enter values for the provided method parameters in the Input Parameter column in the Assigned Values column:
    1. endpointref: http://<cmdbserver>:8080/axis/services/USD\_R11\_WebService?wsdl
    2. userid: ${user}
    3. password: ${password}
    4. propertylist: location.address,location.city

**Note:**

For CA CMDB r12 and above, use location.address1 instead of location.address.

* + 1. selectquery: dns\_name like "%s"
    2. node: ${pattern1.AlertedMdrProdInstance}

This parameter configuration queries the defined CA CMDB instance for CIs with a dns\_name property that matches the event AlertedMdrProdInstance property value and returns the location.address and location.city properties of the matching CI. It uses substitution strings for the required CA CMDB credentials (referencing the credentials entered on the previous page) to avoid entering the information unencrypted.

# Assertion:

Assertion is a statement in java. It can be used to test your assumptions about the program.

While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.

## Advantage of Assertion:

It provides an effective way to detect and correct programming errors.

## Syntax of using Assertion:

There are two ways to use assertion. First way is:

1. **assert** expression;

and second way is:

1. **assert** expression1 : expression2;

### Simple Example of Assertion in java:

1. **import** java.util.Scanner;
3. **class** AssertionExample{
4. **public** **static** **void** main( String args[] ){
6. Scanner scanner = **new** Scanner( System.in );
7. System.out.print("Enter ur age ");
9. **int** value = scanner.nextInt();
10. **assert** value>=18:" Not valid";
12. System.out.println("value is "+value);
13. }
14. }

|  |
| --- |
| If you use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, **-ea** or **-enableassertions** switch of java must be used. |
| Compile it by: **javac AssertionExample.java** |
| Run it by: **java -ea AssertionExample** |

Output: Enter ur age 11

Exception in thread "main" java.lang.AssertionError: Not valid

### Where not to use Assertion:

There are some situations where assertion should be avoid to use. They are:

1. According to Sun Specification, assertion should not be used to check arguments in the public methods because it should result in appropriate runtime exception e.g. IllegalArgumentException, NullPointerException etc.
2. Do not use assertion, if you don't want any error in any situation.

**Unit - 4:**

# Multithreading in Java

**Multithreading in Java** is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* Process-based Multitasking (Multiprocessing)
* Thread-based Multitasking (Multithreading)

### 1) Process-based Multitasking (Multiprocessing)

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading [registers](https://www.javatpoint.com/register-memory), memory maps, updating lists, etc.

### 2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* A thread is lightweight.
* Cost of communication between the thread is low.

#### Note: At least one process is required for each thread.

## What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the [OS](https://www.javatpoint.com/os-tutorial), and one process can have multiple threads.

#### Note: At a time one thread is executed only.

## Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides [constructors](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

## Java Thread Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Modifier and Type** | **Method** | **Description** |
| 1) | void | [start()](https://www.javatpoint.com/java-thread-start-method) | It is used to start the execution of the thread. |
| 2) | void | [run()](https://www.javatpoint.com/java-thread-run-method) | It is used to do an action for a thread. |
| 3) | static void | [sleep()](https://www.javatpoint.com/java-thread-sleep-method) | It sleeps a thread for the specified amount of time. |
| 4) | static Thread | [currentThread()](https://www.javatpoint.com/java-thread-currentthread-method) | It returns a reference to the currently executing thread object. |
| 5) | void | [join()](https://www.javatpoint.com/java-thread-join-method) | It waits for a thread to die. |
| 6) | int | [getPriority()](https://www.javatpoint.com/java-thread-getpriority-method) | It returns the priority of the thread. |
| 7) | void | [setPriority()](https://www.javatpoint.com/java-thread-setpriority-method) | It changes the priority of the thread. |
| 8) | String | [getName()](https://www.javatpoint.com/java-thread-getname-method) | It returns the name of the thread. |
| 9) | void | [setName()](https://www.javatpoint.com/java-thread-setname-method) | It changes the name of the thread. |
| 10) | long | [getId()](https://www.javatpoint.com/java-thread-getid-method) | It returns the id of the thread. |
| 11) | boolean | [isAlive()](https://www.javatpoint.com/java-thread-isalive-method) | It tests if the thread is alive. |
| 12) | static void | [yield()](https://www.javatpoint.com/java-thread-yield-method) | It causes the currently executing thread object to pause and allow other threads to execute temporarily. |
| 13) | void | [suspend()](https://www.javatpoint.com/java-thread-suspend-method) | It is used to suspend the thread. |
| 14) | void | [resume()](https://www.javatpoint.com/java-thread-resume-method) | It is used to resume the suspended thread. |
| 15) | void | [stop()](https://www.javatpoint.com/java-thread-stop-method) | It is used to stop the thread. |
| 16) | void | [destroy()](https://www.javatpoint.com/java-thread-destroy-method) | It is used to destroy the thread group and all of its subgroups. |
| 17) | boolean | [isDaemon()](https://www.javatpoint.com/java-thread-isdaemon-method) | It tests if the thread is a daemon thread. |
| 18) | void | [setDaemon()](https://www.javatpoint.com/java-thread-setdaemon-method) | It marks the thread as daemon or user thread. |
| 19) | void | [interrupt()](https://www.javatpoint.com/java-thread-interrupt-method) | It interrupts the thread. |
| 20) | boolean | [isinterrupted()](https://www.javatpoint.com/java-thread-isinterrupted-method) |  |

# Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

## Explanation of Different Thread States

**New:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

**Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

* **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.  
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
* **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

**Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

**Timed Waiting:** Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

**Terminated:** A thread reaches the termination state because of the following reasons:

* When a thread has finished its job, then it exists or terminates normally.
* **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.

## Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

1. **public** **static** **final** Thread.State NEW

It represents the first state of a thread that is the NEW state.

1. **public** **static** **final** Thread.State RUNNABLE

It represents the runnable state.It means a thread is waiting in the queue to run.

1. **public** **static** **final** Thread.State BLOCKED

It represents the blocked state. In this state, the thread is waiting to acquire a lock.

1. **public** **static** **final** Thread.State WAITING

It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

1. **public** **static** **final** Thread.State TIMED\_WAITING

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

# How to create a thread in Java

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

### Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

### Commonly used Constructors of Thread class:

* Thread()
* Thread(String name)
* Thread(Runnable r)
* Thread(Runnable r,String name)

### Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).

### Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

### Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

* A new thread starts(with new callstack).
* The thread moves from New state to the Runnable state.
* When the thread gets a chance to execute, its target run() method will run.

### 1) Java Thread Example by extending Thread class

**FileName:** Multi.java

1. **class** Multi **extends** Thread{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5. **public** **static** **void** main(String args[]){
6. Multi t1=**new** Multi();
7. t1.start();
8. }
9. }

**Output:**

thread is running...

### 2) Java Thread Example by implementing Runnable interface

**FileName:** Multi3.java

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);   // Using the constructor Thread(Runnable r)
9. t1.start();
10. }
11. }

**Output:**

thread is running...

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

### 3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

**FileName:** MyThread1.java

1. **public** **class** MyThread1
2. {
3. // Main method
4. **public** **static** **void** main(String argvs[])
5. {
6. // creating an object of the Thread class using the constructor Thread(String name)
7. Thread t= **new** Thread("My first thread");
9. // the start() method moves the thread to the active state
10. t.start();
11. // getting the thread name by invoking the getName() method
12. String str = t.getName();
13. System.out.println(str);
14. }
15. }

**Output:**

My first thread

### 4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

**FileName:** MyThread2.java

1. **public** **class** MyThread2 **implements** Runnable
2. {
3. **public** **void** run()
4. {
5. System.out.println("Now the thread is running ...");
6. }
8. // main method
9. **public** **static** **void** main(String argvs[])
10. {
11. // creating an object of the class MyThread2
12. Runnable r1 = **new** MyThread2();
14. // creating an object of the class Thread using Thread(Runnable r, String name)
15. Thread th1 = **new** Thread(r1, "My new thread");
17. // the start() method moves the thread to the active state
18. th1.start();
20. // getting the thread name by invoking the getName() method
21. String str = th1.getName();
22. System.out.println(str);
23. }
24. }

# Thread.sleep() in Java with Examples

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

### The sleep() Method Syntax:

Following are the syntax of the sleep() method.

1. **public** **static** **void** sleep(**long** mls) **throws** InterruptedException
2. **public** **static** **void** sleep(**long** mls, **int** n) **throws** InterruptedException

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

### Parameters:

The following are the parameters used in the sleep() method.

**mls:** The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

**n:** It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

### Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

### Example of the sleep() method in Java : on the custom thread

The following example shows how one can use the sleep() method on the custom thread.

**FileName:** TestSleepMethod1.java

1. **class** TestSleepMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. // the thread will sleep for the 500 milli seconds
5. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
6. System.out.println(i);
7. }
8. }
9. **public** **static** **void** main(String args[]){
10. TestSleepMethod1 t1=**new** TestSleepMethod1();
11. TestSleepMethod1 t2=**new** TestSleepMethod1();
13. t1.start();
14. t2.start();
15. }
16. }

Java join() method

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution. Suppose *th* be the object the class Thread whose thread is doing its execution currently, then the *th.join();* statement ensures that *th* is finished before the program does the execution of the next statement.

**Syntax:**

1. **public** **final** **synchronized** **void** join(**long** mls, **int** nanos) **throws** InterruptedException, where mls is in milliseconds.

## Example of join() Method in Java

The following program shows the usage of the join() method.

**FileName:** ThreadJoinExample.java

1. // A Java program for understanding
2. // the joining of threads
4. // import statement
5. **import** java.io.\*;
7. // The ThreadJoin class is the child class of the class Thread
8. **class** ThreadJoin **extends** Thread
9. {
10. // overriding the run method
11. **public** **void** run()
12. {
13. **for** (**int** j = 0; j < 2; j++)
14. {
15. **try**
16. {
17. // sleeping the thread for 300 milli seconds
18. Thread.sleep(300);
19. System.out.println("The current thread name is: " + Thread.currentThread().getName());
20. }
21. // catch block for catching the raised exception
22. **catch**(Exception e)
23. {
24. System.out.println("The exception has been caught: " + e);
25. }
26. System.out.println( j );
27. }
28. }
29. }
31. **public** **class** ThreadJoinExample
32. {
33. // main method
34. **public** **static** **void** main (String argvs[])
35. {
37. // creating 3 threads
38. ThreadJoin th1 = **new** ThreadJoin();
39. ThreadJoin th2 = **new** ThreadJoin();
40. ThreadJoin th3 = **new** ThreadJoin();
42. // thread th1 starts
43. th1.start();
45. // starting the second thread after when
46. // the first thread th1 has ended or died.
47. **try**
48. {
49. System.out.println("The current thread name is: "+ Thread.currentThread().getName());
51. // invoking the join() method
52. th1.join();
53. }
55. // catch block for catching the raised exception
56. **catch**(Exception e)
57. {
58. System.out.println("The exception has been caught " + e);
59. }
61. // thread th2 starts
62. th2.start();
64. // starting the th3 thread after when the thread th2 has ended or died.
65. **try**
66. {
67. System.out.println("The current thread name is: " + Thread.currentThread().getName());
68. th2.join();
69. }
71. // catch block for catching the raised exception
72. **catch**(Exception e)
73. {
74. System.out.println("The exception has been caught " + e);
75. }
77. // thread th3 starts
78. th3.start();
79. }
80. }

**Output:**

The current thread name is: main

The current thread name is: Thread - 0

0

The current thread name is: Thread - 0

1

The current thread name is: main

The current thread name is: Thread - 1

0

The current thread name is: Thread - 1

1

The current thread name is: Thread - 2

0

The current thread name is: Thread - 2

1

**Explanation:** The above program sh

# ava Thread isAlive() method

The **isAlive()** method of thread class tests if the thread is alive. A thread is considered alive when the start() method of thread class has been called and the thread is not yet dead. This method returns true if the thread is still running and not finished.

## Syntax

1. **public** **final** **boolean** isAlive()

## Return

This method will return true if the thread is alive otherwise returns false.

## Example

1. **public** **class** JavaIsAliveExp **extends** Thread
2. {
3. **public** **void** run()
4. {
5. **try**
6. {
7. Thread.sleep(300);
8. System.out.println("is run() method isAlive "+Thread.currentThread().isAlive());
9. }
10. **catch** (InterruptedException ie) {
11. }
12. }
13. **public** **static** **void** main(String[] args)
14. {
15. JavaIsAliveExp t1 = **new** JavaIsAliveExp();
16. System.out.println("before starting thread isAlive: "+t1.isAlive());
17. t1.start();
18. System.out.println("after starting thread isAlive: "+t1.isAlive());
19. }
20. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaIsAliveExp)

**Output:**

before starting thread isAlive: false

after starting thread isAlive: true

is run() method isAlive true

# Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

### Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

### Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

### Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

### Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

### Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

### Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

**TestSynchronization1.java**

1. **class** Table{
2. **void** printTable(**int** n){//method not synchronized
3. **for**(**int** i=1;i<=5;i++){
4. System.out.println(n\*i);
5. **try**{
6. Thread.sleep(400);
7. }**catch**(Exception e){System.out.println(e);}
8. }
10. }
11. }
13. **class** MyThread1 **extends** Thread{
14. Table t;
15. MyThread1(Table t){
16. **this**.t=t;
17. }
18. **public** **void** run(){
19. t.printTable(5);
20. }
22. }
23. **class** MyThread2 **extends** Thread{
24. Table t;
25. MyThread2(Table t){
26. **this**.t=t;
27. }
28. **public** **void** run(){
29. t.printTable(100);
30. }
31. }
33. **class** TestSynchronization1{
34. **public** **static** **void** main(String args[]){
35. Table obj = **new** Table();//only one object
36. MyThread1 t1=**new** MyThread1(obj);
37. MyThread2 t2=**new** MyThread2(obj);
38. t1.start();
39. t2.start();
40. }
41. }

**Output:**

5

100

10

200

15

300

20

400

25

500

### Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

**TestSynchronization2.java**

1. //example of java synchronized method
2. **class** Table{
3. **synchronized** **void** printTable(**int** n){//synchronized method
4. **for**(**int** i=1;i<=5;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){System.out.println(e);}
9. }
11. }
12. }
14. **class** MyThread1 **extends** Thread{
15. Table t;
16. MyThread1(Table t){
17. **this**.t=t;
18. }
19. **public** **void** run(){
20. t.printTable(5);
21. }
23. }
24. **class** MyThread2 **extends** Thread{
25. Table t;
26. MyThread2(Table t){
27. **this**.t=t;
28. }
29. **public** **void** run(){
30. t.printTable(100);
31. }
32. }
34. **public** **class** TestSynchronization2{
35. **public** **static** **void** main(String args[]){
36. Table obj = **new** Table();//only one object
37. MyThread1 t1=**new** MyThread1(obj);
38. MyThread2 t2=**new** MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }

**Output:**

5

10

15

20

25

100

200

300

400

500

# Deadlock in Java

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



### Example of Deadlock in Java

**TestDeadlockExample1.java**

1. **public** **class** TestDeadlockExample1 {
2. **public** **static** **void** main(String[] args) {
3. **final** String resource1 = "ratan jaiswal";
4. **final** String resource2 = "vimal jaiswal";
5. // t1 tries to lock resource1 then resource2
6. Thread t1 = **new** Thread() {
7. **public** **void** run() {
8. **synchronized** (resource1) {
9. System.out.println("Thread 1: locked resource 1");
11. **try** { Thread.sleep(100);} **catch** (Exception e) {}
13. **synchronized** (resource2) {
14. System.out.println("Thread 1: locked resource 2");
15. }
16. }
17. }
18. };
20. // t2 tries to lock resource2 then resource1
21. Thread t2 = **new** Thread() {
22. **public** **void** run() {
23. **synchronized** (resource2) {
24. System.out.println("Thread 2: locked resource 2");
26. **try** { Thread.sleep(100);} **catch** (Exception e) {}
28. **synchronized** (resource1) {
29. System.out.println("Thread 2: locked resource 1");
30. }
31. }
32. }
33. };

36. t1.start();
37. t2.start();
38. }
39. }

**Output:**

Thread 1: locked resource 1

Thread 2: locked resource 2

### More Complicated Deadlocks

A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

### How to avoid deadlock?

A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.

**DeadlockSolved.java**

1. **public** **class** DeadlockSolved {
3. **public** **static** **void** main(String ar[]) {
4. DeadlockSolved test = **new** DeadlockSolved();
6. **final** resource1 a = test.**new** resource1();
7. **final** resource2 b = test.**new** resource2();
9. // Thread-1
10. Runnable b1 = **new** Runnable() {
11. **public** **void** run() {
12. **synchronized** (b) {
13. **try** {
14. /\* Adding delay so that both threads can start trying to lock resources \*/
15. Thread.sleep(100);
16. } **catch** (InterruptedException e) {
17. e.printStackTrace();
18. }
19. // Thread-1 have resource1 but need resource2 also
20. **synchronized** (a) {
21. System.out.println("In block 1");
22. }
23. }
24. }
25. };
27. // Thread-2
28. Runnable b2 = **new** Runnable() {
29. **public** **void** run() {
30. **synchronized** (b) {
31. // Thread-2 have resource2 but need resource1 also
32. **synchronized** (a) {
33. System.out.println("In block 2");
34. }
35. }
36. }
37. };

40. **new** Thread(b1).start();
41. **new** Thread(b2).start();
42. }
44. // resource1
45. **private** **class** resource1 {
46. **private** **int** i = 10;
48. **public** **int** getI() {
49. **return** i;
50. }
52. **public** **void** setI(**int** i) {
53. **this**.i = i;
54. }
55. }
57. // resource2
58. **private** **class** resource2 {
59. **private** **int** i = 20;
61. **public** **int** getI() {
62. **return** i;
63. }
65. **public** **void** setI(**int** i) {
66. **this**.i = i;
67. }
68. }
69. }

**Output:**

In block 1

In block 2

In the above code, class DeadlockSolved solves the deadlock kind of situation. It will help in avoiding deadlocks, and if encountered, in resolving them.

### How to Avoid Deadlock in Java?

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

1. **Avoid Nested Locks**: We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
2. **Avoid Unnecessary Locks**: The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
3. **Using Thread Join**: A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use **join** with a maximum time that a thread will take.

# Inter-thread Communication in Java

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

* wait()
* notify()
* notifyAll()

### 1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final void wait()throws InterruptedException | It waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException | It waits for the specified amount of time. |

### 2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

**Syntax:**

1. **public** **final** **void** notify()

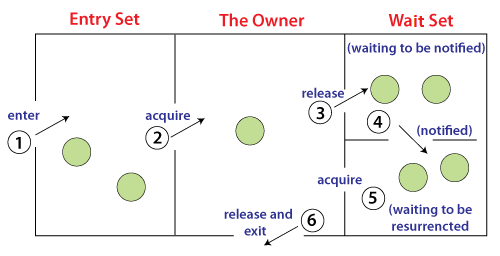
### 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

**Syntax:**

1. **public** **final** **void** notifyAll()

## Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

### Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

### Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| The wait() method releases the lock. | The sleep() method doesn't release the lock. |
| It is a method of Object class | It is a method of Thread class |
| It is the non-static method | It is the static method |
| It should be notified by notify() or notifyAll() methods | After the specified amount of time, sleep is completed. |

### Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

**Test.java**

1. **class** Customer{
2. **int** amount=10000;
4. **synchronized** **void** withdraw(**int** amount){
5. System.out.println("going to withdraw...");
7. **if**(**this**.amount<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. **try**{wait();}**catch**(Exception e){}
10. }
11. **this**.amount-=amount;
12. System.out.println("withdraw completed...");
13. }
15. **synchronized** **void** deposit(**int** amount){
16. System.out.println("going to deposit...");
17. **this**.amount+=amount;
18. System.out.println("deposit completed... ");
19. notify();
20. }
21. }
23. **class** Test{
24. **public** **static** **void** main(String args[]){
25. **final** Customer c=**new** Customer();
26. **new** Thread(){
27. **public** **void** run(){c.withdraw(15000);}
28. }.start();
29. **new** Thread(){
30. **public** **void** run(){c.deposit(10000);}
31. }.start();
33. }}

**Output:**

going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed

# Thread suspend() method

The **suspend()** method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

## Syntax

1. **public** **final** **void** suspend()

## Return

This method does not return any value.

## Exception

**SecurityException:** If the current thread cannot modify the thread.

## Example

1. **public** **class** JavaSuspendExp **extends** Thread
2. {
3. **public** **void** run()
4. {
5. **for**(**int** i=1; i<5; i++)
6. {
7. **try**
8. {
9. // thread to sleep for 500 milliseconds
10. sleep(500);
11. System.out.println(Thread.currentThread().getName());
12. }**catch**(InterruptedException e){System.out.println(e);}
13. System.out.println(i);
14. }
15. }
16. **public** **static** **void** main(String args[])
17. {
18. // creating three threads
19. JavaSuspendExp t1=**new** JavaSuspendExp ();
20. JavaSuspendExp t2=**new** JavaSuspendExp ();
21. JavaSuspendExp t3=**new** JavaSuspendExp ();
22. // call run() method
23. t1.start();
24. t2.start();
25. // suspend t2 thread
26. t2.suspend();
27. // call run() method
28. t3.start();
29. }
30. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaSuspendExp)

**Output:**

Thread-0

1

Thread-2

1

Thread-0

2

Thread-2

2

Thread-0

3

Thread-2

3

Thread-0

4

Thread-2

4

# Thread resume() method

The **resume()** method of thread class is only used with suspend() method. This method is used to resume a thread which was suspended using suspend() method. This method allows the suspended thread to start again.

## Syntax

1. **public** **final** **void** resume()

## Return value

This method does not return any value.

## Exception

**SecurityException:** If the current thread cannot modify the thread.

## Example

1. **public** **class** JavaResumeExp **extends** Thread
2. {
3. **public** **void** run()
4. {
5. **for**(**int** i=1; i<5; i++)
6. {
7. **try**
8. {
9. // thread to sleep for 500 milliseconds
10. sleep(500);
11. System.out.println(Thread.currentThread().getName());
12. }**catch**(InterruptedException e){System.out.println(e);}
13. System.out.println(i);
14. }
15. }
16. **public** **static** **void** main(String args[])
17. {
18. // creating three threads
19. JavaResumeExp t1=**new** JavaResumeExp ();
20. JavaResumeExp t2=**new** JavaResumeExp ();
21. JavaResumeExp t3=**new** JavaResumeExp ();
22. // call run() method
23. t1.start();
24. t2.start();
25. t2.suspend(); // suspend t2 thread
26. // call run() method
27. t3.start();
28. t2.resume(); // resume t2 thread
29. }
30. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaResumeExp)

**Output:**

Thread-0

1

Thread-2

1

Thread-1

1

Thread-0

2

Thread-2

2

Thread-1

2

Thread-0

3

Thread-2

3

Thread-1

3

Thread-0

4

Thread-2

4

Thread-1

4

# Reading and writing in the array using threads



Writing and reading elements in an array is a small problem where each element is first added to the array and then the entire array is read element by element and printed on the console. But when the number of elements is too large, it could take a lot of time. But this could be solved by dividing the writing and reading tasks into parts.

This could be done by using multi-threading where each core of the processor is used. In this case, two threads are used, where one thread is responsible for writing to the array and the other thread is responsible for reading the array. In this way, the performance of a program can be improved as well as the cores of the processor can be utilized. It is better to use one thread for each core. Although one can create as many threads as required for a better understanding of multi-threading.

This article focuses on writing and reading the elements of the array using the concept of multithreading.

**Approach:** This section states the algorithm that is followed to design a program to writing and read elements of an array using multithreading:

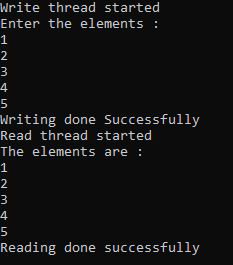
* In the first step, two threads will be created.
* One for writing operation and one for reading operation.
* Here the synchronized keyword is used with the array so that only one thread can access the array at a time.
* First, the write operation will be performed on the array.
* Then, the read operation is performed on the array.

Below is the Java program to implement the above approach-

* C++14
* Java
* Python3
* C#
* Javascript

|  |
| --- |
| #include <iostream>  #include <thread>  using namespace std;    int main()  {      // Array created for 5 elements      int a[5];        // Thread created for write operation      thread t1([]() {          // Here the array is being          // synchronized          lock\_guard<mutex> lock(m);          cout << "Enter the elements : " << endl;          for (int i = 0; i < 5; i++) {              cin >> a[i];          }          cout << "Writing done Successfully" << endl;      });        // Thread created for read operation      thread t2([]() {          // Here the array is being          // synchronized          lock\_guard<mutex> lock(m);          cout << "The elements are : " << endl;          for (int i = 0; i < 5; i++) {              cout << a[i] << endl;          }          cout << "Reading done successfully" << endl;      });        // Write thread is started      t1.join();        // Read thread is started      t2.join();        return 0;  } |

**Output:**



**Explanation:** Here, firstly the write thread is started and at that time read thread will not interfere as the array is synchronized. Similarly, during reading, write thread will not interfere.

**UNIT – 5:**

**Java Threads**

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

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

**Thread States in Java:**

A thread is a program in execution created to perform a specific task. Life cycle of a Java thread starts with its birth and ends on its death.

The start() method of the Thread class is used to initiate the execution of a thread and it goes into runnable state and the sleep() and wait() methods of the Thread class sends the thread into non runnable state.

After non runnable state, thread again comes into runnable state and starts its execution. The run() method of thread is very much important. After executing the run() method, the lifecycle of thread is completed.

**Interrupting a Thread:**

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

* public void interrupt()
* public static boolean interrupted()
* public booleanisInterrupted()

**Inter-Thread communication:**

Inter-thread communication in Java is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

Note: Inter-thread communication is also known as Cooperation in Java.

**Java Networking:**

Java Networking is a concept of connecting two or more computing devices together so that we can share resources.

Java socket programming provides facility to share data between different computing devices.

### Advantage of Java Networking

1. Sharing resources
2. Centralize software management

The java.net package supports two protocols,

1. **TCP:** Transmission Control Protocol provides reliable communication between the sender and receiver. TCP is used along with the Internet Protocol referred as TCP/IP.
2. **UDP:** User Datagram Protocol provides a connection-less protocol service by allowing packet of data to be transferred along two or more nodes

## Java Networking Terminology

The widely used Java networking terminologies are given below:

1. IP Address
2. Protocol
3. Port Number
4. MAC Address
5. Connection-oriented and connection-less protocol
6. Socket

### 1) IP Address

IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255.

It is a logical address that can be changed.

### 2) Protocol

A protocol is a set of rules basically that is followed for communication. For example:

* TCP
* FTP
* Telnet
* SMTP
* POP etc.

### 3) Port Number

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications.

The port number is associated with the IP address for communication between two applications.

### 4) MAC Address

MAC (Media Access Control) address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC address.

For example, an ethernet card may have a **MAC** address of 00:0d:83::b1:c0:8e.

### 5) Connection-oriented and connection-less protocol

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP.

But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP.

### 6) Socket

A socket is an endpoint between two way communications.

Visit next page for Java socket programming.

## java.net package

The java.net package can be divided into two sections:

1. **A Low-Level API:** It deals with the abstractions of addresses i.e. networking identifiers, Sockets i.e. bidirectional data communication mechanism and Interfaces i.e. network interfaces.
2. **A High Level API:** It deals with the abstraction of URIs i.e. Universal Resource Identifier, URLs i.e. Universal Resource Locator, and Connections i.e. connections to the resource pointed by URLs.

The java.net package provides many classes to deal with networking applications in Java. A list of these classes is given below:

* Authenticator
* CacheRequest
* CacheResponse
* ContentHandler
* CookieHandler
* CookieManager
* DatagramPacket
* DatagramSocket
* DatagramSocketImpl
* InterfaceAddress
* JarURLConnection
* MulticastSocket
* InetSocketAddress
* InetAddress
* Inet4Address
* Inet6Address
* IDN
* HttpURLConnection
* HttpCookie
* NetPermission
* NetworkInterface
* PasswordAuthentication
* Proxy
* ProxySelector
* ResponseCache
* SecureCacheResponse
* ServerSocket
* Socket
* SocketAddress
* SocketImpl
* SocketPermission
* StandardSocketOptions

# Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information: 1.1K

HTML Tutorial

1. IP Address of Server, and
2. Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.

## Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

### Important methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public InputStream getInputStream() | returns the InputStream attached with this socket. |
| 2) public OutputStream getOutputStream() | returns the OutputStream attached with this socket. |
| 3) public synchronized void close() | closes this socket |

## ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

### Important methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public Socket accept() | returns the socket and establish a connection between server and client. |
| 2) public synchronized void close() | closes the server socket. |

## Example of Java Socket Programming

**Creating Server:**

To create the server application, we need to create the instance of ServerSocket class. Here, we are using 6666 port number for the communication between the client and server. You may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

1. ServerSocket ss=**new** ServerSocket(6666);
2. Socket s=ss.accept();//establishes connection and waits for the client

**Creating Client:**

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

1. Socket s=**new** Socket("localhost",6666);

Let's see a simple of Java socket programming where client sends a text and server receives and prints it.

*File: MyServer.java*

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public** **class** MyServer {
4. **public** **static** **void** main(String[] args){
5. **try**{
6. ServerSocket ss=**new** ServerSocket(6666);
7. Socket s=ss.accept();//establishes connection
8. DataInputStream dis=**new** DataInputStream(s.getInputStream());
9. String  str=(String)dis.readUTF();
10. System.out.println("message= "+str);
11. ss.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

*File: MyClient.java*

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public** **class** MyClient {
4. **public** **static** **void** main(String[] args) {
5. **try**{
6. Socket s=**new** Socket("localhost",6666);
7. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
8. dout.writeUTF("Hello Server");
9. dout.flush();
10. dout.close();
11. s.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

# ava InetAddress class

**Java InetAddress** class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name for example www.java.com, www.google.com, www.facebook.com, etc.

An IP address is represented by 32-bit or 128-bit unsigned number. An instance of InetAddress represents the IP address with its corresponding host name. There are two types of addresses: Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

Moreover, InetAddress has a cache mechanism to store successful and unsuccessful host name resolutions.

## IP Address

* An IP address helps to identify a specific resource on the network using a numerical representation.
* Most networks combine IP with TCP (Transmission Control Protocol). It builds a virtual bridge among the destination and the source.

There are two versions of IP address:

### 1. IPv4

IPv4 is the primary Internet protocol. It is the first version of IP deployed for production in the ARAPNET in 1983. It is a widely used IP version to differentiate devices on network using an addressing scheme. A 32-bit addressing scheme is used to store 232addresses that is more than 4 million addresses.

**Features of IPv4:**

* It is a connectionless protocol.
* It utilizes less memory and the addresses can be remembered easily with the class based addressing scheme.
* It also offers video conferencing and libraries.

### 2. IPv6

IPv6 is the latest version of Internet protocol. It aims at fulfilling the need of more internet addresses. It provides solutions for the problems present in IPv4. It provides 128-bit address space that can be used to form a network of 340 undecillion unique IP addresses. IPv6 is also identified with a name IPng (Internet Protocol next generation).

**Features of IPv6:**

* It has a stateful and stateless both configurations.
* It provides support for quality of service (QoS).
* It has a hierarchical addressing and routing infrastructure.

## TCP/IP Protocol

* TCP/IP is a communication protocol model used connect devices over a network via internet.
* TCP/IP helps in the process of addressing, transmitting, routing and receiving the data packets over the internet.
* The two main protocols used in this communication model are:
  1. TCP i.e. Transmission Control Protocol. TCP provides the way to create a communication channel across the network. It also helps in transmission of packets at sender end as well as receiver end.
  2. IP i.e. Internet Protocol. IP provides the address to the nodes connected on the internet. It uses a gateway computer to check whether the IP address is correct and the message is forwarded correctly or not.

## Java InetAddress Class Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static InetAddress getByName(String host) throws UnknownHostException | It returns the instance of InetAddress containing LocalHost IP and name. |
| public static InetAddress getLocalHost() throws UnknownHostException | It returns the instance of InetAdddress containing local host name and address. |
| public String getHostName() | It returns the host name of the IP address. |
| public String getHostAddress() | It returns the IP address in string format. |

## Example of Java InetAddress Class

Let's see a simple example of InetAddress class to get ip address of www.java.com website.

**InetDemo.java**

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public** **class** InetDemo{
4. **public** **static** **void** main(String[] args){
5. **try**{
6. InetAddress ip=InetAddress.getByName("www.java.com");
8. System.out.println("Host Name: "+ip.getHostName());
9. System.out.println("IP Address: "+ip.getHostAddress());
10. }**catch**(Exception e){System.out.println(e);}
11. }
12. }

### TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable two-way, persistent, point-to-point streaming connections between hosts on the Internet. The Java I/O system can use sockets to connect to other programs on the local system or on other systems on the Internet. It is important to note that the applet establishes a reverse socket connection to the host on which the applet is loaded. This restriction exists because it is dangerous for applets loaded through a firewall to access arbitrary systems. There are two types of TCP sockets in Java.

One for the **server** and one for the **client**. The ServerSocket class is designed as a "listener", waiting for a client to connect before doing anything. So ServerSocket is for servers. The Socket class is for clients. It is designed to connect to a server socket and initiate a protocol exchange. This is because client sockets are most commonly used in Java applications. Creating a Socket object implicitly establishes a connection between the client and server. There is no method or constructor that explicitly exposes details about setting up this connection.

Here are the two constructors used to create a client socket:

1. **Socket(String hostName, int port) throws UnknownHostException, IOException:** Creates a socket connected to the specified host and port.
2. **Socket(InetAddress ipAddress, int port) throws IOException:** Creates a socket using a pre-existing InetAddress object and a port.

Socket defines multiple instance methods. For example, a Socket can always check for associated address and port information using the following methods:

1. **InetAddress getInetAddress( ):** It returns the InetAddress associated with the Socket object. It returns null if the socket is not connected.
2. **int getPort( ):** It returns the remote port to which the invoking Socket object is connected. It returns 0 if the socket is not connected.
3. **int getLocalPort( ):** Returns the local port to which the invoking Socket object is bound. It returns -1 if the socket is not bound.
4. **InputStream getInputStream( ) throws IOException:** Returns the InputStream associated with the invoking socket.
5. **OutputStream getOutputStream( ) throws IOException:** Returns the OutputStream associated with the invoking socket.
6. **connect( ):** Allows you to specify a new connection
7. **isConnected( ):** Returns true if the socket is connected to a server
8. **isBound( ):** Returns true if the socket is bound to an address
9. **isClosed( ):** Returns true if the socket is closed.

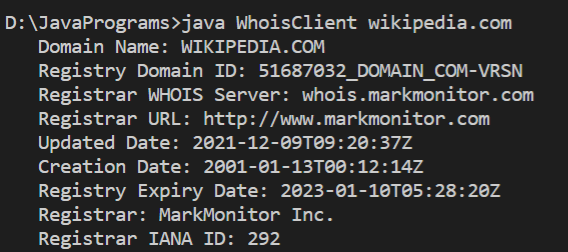
The following program provides a simple socket example. Opens a connection to a "whois" port (port 43) of the InterNIC server, sends command-line argument to the socket, and prints the returned data. The InterNIC will try find the argument by the registered Internet domain name, and then send back the IP address and contact information for that site.

### Example of Stream Socket

**WhoisClient.java**

1. **import** java.net.\*;
2. **import** java.io.\*;
3. **public** **class** WhoisClient {
4. **public** **static** **void** main(String[] args) {
5. // no arguments passed, simply return
6. **if** (args.length < 1)
7. **return**;
8. // initializing domainName with the name passed in the argument
9. String domainName = args[0];
10. // specifying the host name
11. String hostname = "whois.internic.net";
12. **int** port = 43;
13. **try** (Socket socket = **new** Socket(hostname, port)) {
14. // getOutputStream( ) returns the OutputStream
15. // associated with the invoking socket
16. OutputStream output = socket.getOutputStream();
17. PrintWriter writer = **new** PrintWriter(output, **true**);
18. // print the domain name
19. writer.println(domainName);
20. // getInputStream( ) returns the InputStream
21. // associated with the invoking socket
22. InputStream input = socket.getInputStream();
23. BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(input));
24. String line;
25. **while** ((line = reader.readLine()) != **null**) {
26. System.out.println(line);
27. }
28. }
29. **catch** (UnknownHostException ex) {
30. System.out.println("Server not found: " + ex.getMessage());
31. }
32. **catch** (IOException ex) {
33. System.out.println("I/O error: " + ex.getMessage());
34. }
35. }
36. }

**Output:**



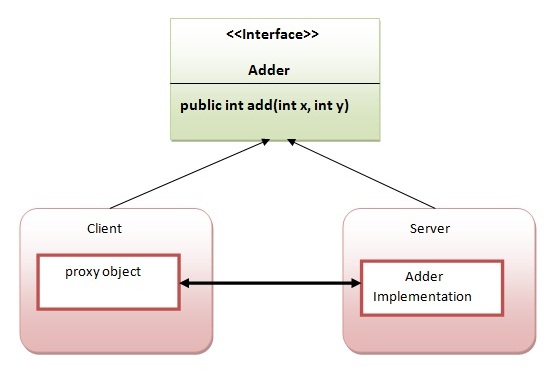
## Java RMI Example

The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

## RMI Example

In this example, we have followed all the 6 steps to create and run the rmi application. The client application need only two files, remote interface and client application. In the rmi application, both client and server interacts with the remote interface. The client application invokes methods on the proxy object, RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.



### 1) create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

1. **import** java.rmi.\*;
2. **public** **interface** Adder **extends** Remote{
3. **public** **int** add(**int** x,**int** y)**throws** RemoteException;
4. }

### 2) Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

* Either extend the UnicastRemoteObject class,
* or use the exportObject() method of the UnicastRemoteObject class

In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.

1. **import** java.rmi.\*;
2. **import** java.rmi.server.\*;
3. **public** **class** AdderRemote **extends** UnicastRemoteObject **implements** Adder{
4. AdderRemote()**throws** RemoteException{
5. **super**();
6. }
7. **public** **int** add(**int** x,**int** y){**return** x+y;}
8. }

### 3) create the stub and skeleton objects using the rmic tool.

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

1. rmic AdderRemote

### 4) Start the registry service by the rmiregistry tool

Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

1. rmiregistry 5000

### 5) Create and run the server application

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. The Naming class provides 5 methods.

|  |  |
| --- | --- |
| public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException; | It returns the reference of the remote object. |
| public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException; | It binds the remote object with the given name. |
| public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException; | It destroys the remote object which is bound with the given name. |
| public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException; | It binds the remote object to the new name. |
| public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException; | It returns an array of the names of the remote objects bound in the registry. |

In this example, we are binding the remote object by the name sonoo.

1. **import** java.rmi.\*;
2. **import** java.rmi.registry.\*;
3. **public** **class** MyServer{
4. **public** **static** **void** main(String args[]){
5. **try**{
6. Adder stub=**new** AdderRemote();
7. Naming.rebind("rmi://localhost:5000/sonoo",stub);
8. }**catch**(Exception e){System.out.println(e);}
9. }
10. }

### 6) Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

1. **import** java.rmi.\*;
2. **public** **class** MyClient{
3. **public** **static** **void** main(String args[]){
4. **try**{
5. Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
6. System.out.println(stub.add(34,4));
7. }**catch**(Exception e){}
8. }
9. }
10. For running **this** rmi example,
12. 1) compile all the java files
14. javac \*.java
16. 2)create stub and skeleton object by rmic tool
18. rmic AdderRemote
20. 3)start rmi registry in one command prompt
22. rmiregistry 5000
24. 4)start the server in another command prompt
26. java MyServer
28. 5)start the client application in another command prompt
30. java MyClient

Another Example

### Creating a Simple RMI application involves following steps

* Define a remote interface.
* Implementing remote interface.
* create and start remote application
* create and start client application

### Define a remote interface

A remote interface specifies the methods that can be invoked remotely by a client. Clients program communicate to remote interfaces, not to classes implementing it. To be a remote interface, a interface must extend the **Remote** interface of **java.rmi** package.

import java.rmi.\*;

public interface AddServerInterface extends Remote

{

public int sum(int a,int b);

}

Copy

### Implementation of remote interface

For implementation of remote interface, a class must either extend **UnicastRemoteObject** or use exportObject() method of **UnicastRemoteObject** class.

import java.rmi.\*;

import java.rmi.server.\*;

public class Adder extends UnicastRemoteObject implements AddServerInterface

{

Adder()throws RemoteException{

super();

}

public int sum(int a,int b)

{

return a+b;

}

}

Copy

### Create AddServer and host rmi service

You need to create a server application and host rmi service **Adder** in it. This is done using rebind() method of **java.rmi.Naming** class. rebind() method take two arguments, first represent the name of the object reference and second argument is reference to instance of **Adder**

import java.rmi.\*;

import java.rmi.registry.\*;

public class AddServer {

public static void main(String args[]) {

try {

AddServerInterface addService=new Adder();

Naming.rebind("AddService",addService); //addService object is hosted with name AddService

}

catch(Exception e) {

System.out.println(e);

}

}

}

Copy

### Create client application

Client application contains a java program that invokes the lookup() method of the **Naming** class. This method accepts one argument, the **rmi** URL and returns a reference to an object of type **AddServerInterface**. All remote method invocation is done on this object.

import java.rmi.\*;

public class Client {

public static void main(String args[]) {

try{

AddServerInterface st = (AddServerInterface)Naming.lookup("rmi://"+args[0]+"/AddService");

System.out.println(st.sum(25,8));

}

catch(Exception e) {

System.out.println(e);

}

}

}

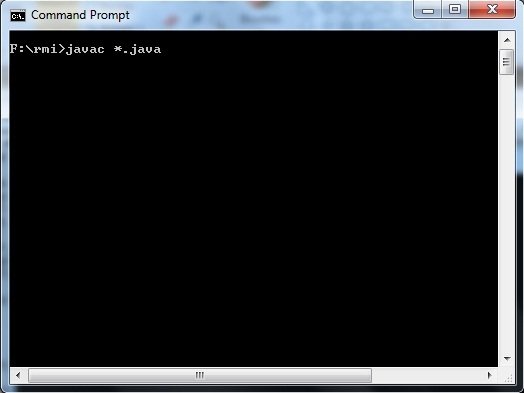
Copy

### Steps to run this RMI application

Save all the above java file into a directory and name it as "rmi"

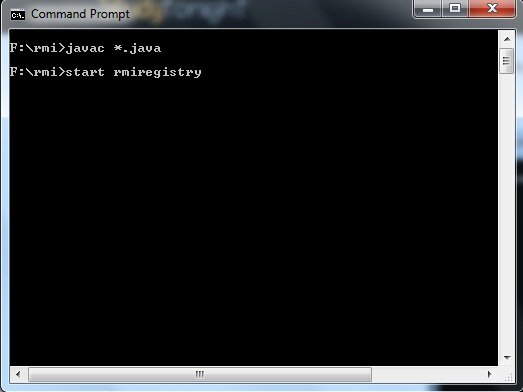
* compile all the java files

javac \*.java



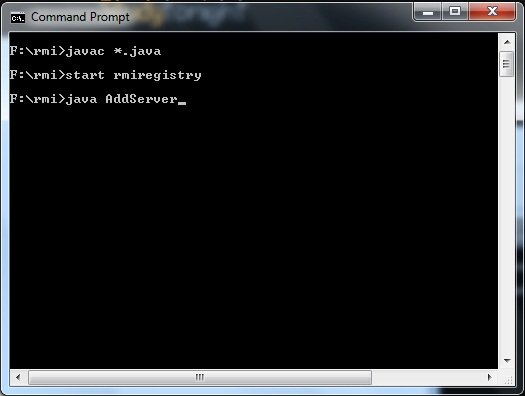
* Start RMI registry

start rmiregistry



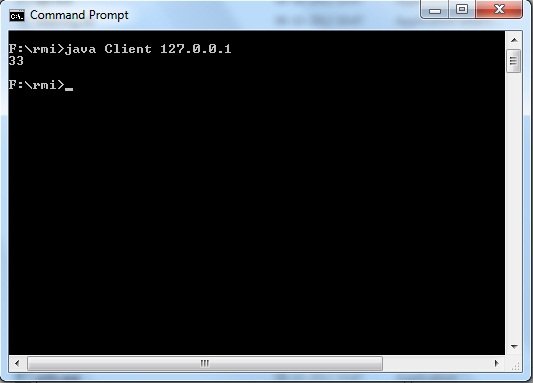
* Run Server file

java AddServer



* Run Client file in another command prompt abd pass local host port number at run time

java Client 127.0.0.1



### Example:

**Program: Power.java**

import java.rmi.\*;

public interface Power extends Remote

{

public int power1()throwsRemoteException;

}

Copy

**Program: PowerRemote.java**

import java.rmi.\*;

import java.rmi.server.\*;

import java.util.Scanner;

public class PowerRemote extends UnicastRemoteObject implements Power

{

PowerRemote()throws RemoteException

{

super();

}

public int power1(int z)

{

int z;

Scanner sc = new Scanner(System.in);

System.out.println("Enter the base number ::");

int x = sc.nextInt();

System.out.println("Enter the exponent number ::");

int y = sc.nextInt();

z=y^x;

System.out.println(z);

}

}

Copy

**MyServer.java**

import java.rmi.\*;

import java.rmi.registry.\*;

public class MyServer

{

public static void main(String args[])

{

try

{

Power stub=new PowerRemote();

Naming.rebind("rmi://localhost:1995/shristee",stub);

}

catch(Exception e)

{

System.out.println(e);

}

}

}

Copy

**MyClient.java**

import java.rmi.\*;

public class MyClient

{

public static void main(String args[])

{

try

{

Power stub=(Power)Naming.lookup("rmi://localhost:1995/shristee");

System.out.println(stub.power1());

}

catch(Exception e){}

}

}

Copy



### Output of this RMI exampleRMI RMI