Java-Doc.

Below document contains the information regarding the core Java, Multi-threading – framework , collection framework and executor framework and basic understanding of web-services in Java.—Amit.

Starting with Core-Java

JDK = JRE+JVM

JVM (Java virtual machine) 🡪 Converts Java language code to byte code which can be run on any machine and platform.

JVM is platform independent.

What is JVM?

It is the specification where working of Java machine is specified. But implementation provider is independent to choose the algorithm. Its implementation is provided by sun and other companies.

An Implementation: JVM implementation is known as JRE

Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does?

The JVM performs following operation:

Loads code, verifies code, executes code provides runtime information.

It also provides the information for the:

Memory area, class file format, register set, Garbage collected heap, Fatal error reporting etc.

JRE (Java runtime environment) 🡪 It provides the runtime environment for the Java application.

JDK (Java development Kit) =JRE+JVM

Variables : There are basically 3 types of variables in Java,

* Local variable
* Instance variable
* Static variable

Local variable: The variable that is declared inside the function(method) are called local variable.

Instance variable : The instance variable are the variable which are called inside the class , but outside the method is called the instance variable.

Static variable:

The variable which is declared as static is called static variable, they are not local variable.

Data types in Java:

* Primitive data types
* Non- primitive data types

Non-primitive data types : String, Array are the non – primitive data type.

Primitive data types: int, byte , Boolean, short, float, double etc are the primitive data types

|  |  |  |
| --- | --- | --- |
| **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 |

Java uses Unicode system.

**Operators in Java:**

Unary operator

Arithmetic operator

Bitwise operator

Shift operator

Relational operator

Logical operator

Ternary operator

Assignment operator

Java Operator Precedence

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | postfix | *expr*++ *expr*-- |
| prefix | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| Arithmetic | multiplicative | \* / % |
| additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instanceof |
| equality | == != |
| Bitwise | bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

Java if statements:

* if statement
* if-else statement
* nested if statement
* if-else-if ladder

Are the types of if statements.

Switch statements: Switch statements execute one statements from the multiple conditions.

There are different types of loops as well like while, for, do..While

Break statement: The break statement is specifly is used to break the flow of the program at the specified program, like inner loop.

Continue statement: Continue statement continues the flow of the program and skips remaining code at specified condition. In case of inner loop it continues only the inner loop.

Java have single line comments and multiline comments.

**Java Object oriented programming language:**

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

Object: Any entity that has the state and the behavior is the object.

Class: Collection of the objects is called the class, it is the logical entity.

Inheritance: when the child class is having all the properties and the behaviors of the parent class it is called inheritance.

Polymorphism: When one task is performed by different ways, it is called polymorphism, In Java we use method overloading and method overriding as a part of polymorphism.

Abstraction: Hiding internal details and showing functionality is known as abstraction, in Java we use interface and abstract class to achieve abstraction.

Encapsulation: Biding/Wrapping a code and data together into a single unit is known as encapsulation. In Java Java-bean is the example of encapsulation.

Advantages: Oops make development faster and easier, provides data hiding capabilities, able to simulate the real world event more effectively.

Difference between object oriented programming and Object based programming:

Object based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object based programming languages.

Java Naming convention:

|  |  |
| --- | --- |
| **`** | **Convention** |
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX\_PRIORITY etc. |

## CamelCase in java naming conventions

Java follows camelcase syntax for naming the class, interface, method and variable.

Object and class in Java:

Object is the physical as well logical entity, whereas the class is the logical entity.

The Object has the state, behavior, identity. It is the logical entity as well as physical entity.

* **state:** represents data (value) of an object.
* **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
* **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

Class is the template from which object can be prepared.

A variable which is created inside the class but outside the method, is known as instance variable. Instance variable doesn't get memory at compile time. It gets memory at run time when object(instance) is created. That is why, it is known as instance variable.

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

* By new keyword
* By newInstance() method
* By clone() method
* By deserialization
* By factory method etc.

We will learn these ways to create object later.

## Anonymous object

Anonymous simply means nameless. An object which has no reference is known as anonymous object. It can be used at the time of object creation only.

**Constructor in Java:**

Constructor in Java is basically called at the time of object creation.

Rules for creating Java constructor:

* The constructor name should be same as the name of the class.
* Constructor should not have return type.

Types of constructor:

The constructor are basically are of 2 types:

* Parameterized constructor
* Non-parameterized constructor.

If there is no constructor in the class, Compiler will automatically creates the default constructor, default constructor will assign the values of the null and 0 to the integer.

Parameterized constructor is basically used to assigned the different values to the objects.

Constructor overloading in Java:

Constructor overloading is basically done when the number of parameters are different while defining the constructor.

## Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| Constructor is used to initialize the state of an object. | Method is used to expose behaviour of an object. |
| Constructor must not have return type. | Method must have return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The java compiler provides a default constructor if you don't have any constructor. | Method is not provided by compiler in any case. |
| Constructor name must be same as the class name. | Method name may or may not be same as class name. |

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

### **Q) Does constructor return any value?**

**Ans:**yes, that is current class instance (You cannot use return type yet it returns a value).

**Java Static Keyword:**

In Java, Static keyword can be applied to variable, method, block and class.

Java static variable: If you declare any variable as static, it is known as static variable.

The static variable can be used to refer the common property of all objects (that is not unique for each object).

The static variable gets memory only once in class area at the time of class loading.

Advantages of static variable:

It makes your program memory efficient.

Java static property is shared to all objects.

**Java static method:**

If you apply static keyword to the method , It is known as static method.

A static method belongs to the class rather than object of a class.

A static method can be invoked without the need for creating an instance of a class.

Static method can access static data member and can change the value of it.

**Restrictions of Java static method:**

The static method cannot use non- static data member or call non- static method directly.

This and super cannot be used in the static context.

Why main method is static?

Because object is not required to call static method if it were non-static method.JVM create object first then call main() method that will lead the problem of extra memory allocation.

**Java static block:**

It is used to initialize the static data member.

It is executed before main method at the time of classloading.

One of the way is static block but in previous version of JDK not in JDK 1.7

**THIS Keyword in Java:**

There is lot of Usage of this Keyword , basic use of This Keyword is reference variable that refers to the current object.

#### It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

#### Rule: Call to this() must be the first statement in constructor.

**Inheritance of Java:**

Java supports single inheritance in the form of IS a relationship. In Java we child class can have all the properties of parent class.

Use of inheritance is basically used for:

Method overloading (so runtime polymorphism can be achieved).

For code reusability.

Java does not support multiple inheritance.

In Java extends keyword is used for inheritance.

This is –a relationship which is used for extending.

Aggregation in Java: If a class have an entity reference it is known as aggregation.

It also show the AS-A relationship in the class.

Aggregation is best used in case of the code –reusability.

The inheritance is maintained when we have is a relationship.

**Method overloading:**

When we have multiple methods with the same name it is known as method overloading. If we have the same operation, it increase the readability of the program.

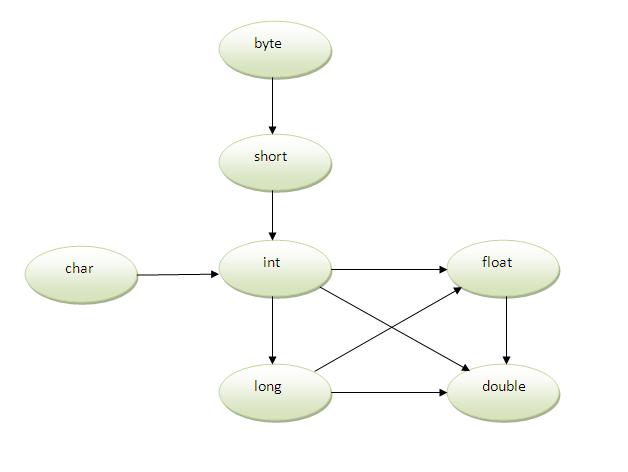
There are 2 ways to perform the method overloading:

1.)By changing the datatype.

2.)By changing number of arguments.

In Java, method overloading is not possible by just changing the return type of the method only.

Given below is the type of promotion that is possible.



#### One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

**Method overriding in Java:**

If the child class have the same method as the parent class it is known as method overriding,

Usage of Java method overriding:

1. Method overriding is used to provide specific implementation of a method that is already provided by its super class.
2. Method overriding is used for runtime polymorphism.

Rules to be followed while doing method overriding:

1. Method must have the same name as it is in the parent class.
2. It must have same number of parameters.
3. Must be the IS- A relationship(inheritance).

Static method can’t be overridden , because static method is bound to class whreas the instance method is bound to the object. Static method belong to class area whereas the static method belongs to heap area.

# **Covariant Return Type**

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

#### Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

### **Simple example of Covariant Return Type**

1. **class** A{
2. A get(){**return** **this**;}
3. }
5. **class** B1 **extends** A{
6. B1 get(){**return** **this**;}
7. **void** message(){System.out.println("welcome to covariant return type");}
9. **public** **static** **void** main(String args[]){
10. **new** B1().get().message();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=B1)

Output:welcome to covariant return type

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

Super keyword in Java:

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Usage of Super keyword:

Super can be used to refer immediate parent class instance method.

Super can be used to invoke immediate parent class method.

Super can be used to invoke immediate parent class constructor.

#### Note: super() is added in each class constructor automatically by compiler if there is no super() or this().

Instance initialize block:

Instance initialize block is used to initialize the instance data member It run each time when object of the class is created.

The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

There are three places in java where you can perform operations:

1. method
2. constructor
3. block

#### **Note: The java compiler copies the code of instance initializer block in every constructor.**

Rules for instance initializer block:

|  |
| --- |
| There are mainly three rules for the instance initializer block. They are as follows: |

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Final Keyword in Java:

The Final keyword in Java is used to restrict the user. The Java final keyword can be used to restrict the user. The Java final keyword can be used in many context. Final can be:

Variable

Method

Class.

Java Final variable :If you make any variable as final you can change the value of the variable.

Stop method overriding

Stop inheritance.

If we don’t give the value to the final variable at the time of initialization we can initialize such variable only in the constructor.

In the case of static final variable if the variable is not initialized we can give the value of the same in static block.

If we declare any parameter that is the method as final we can’t change the value of the same.

Java Final method: If you make any method as final it is not possible to override the same.

Final method is inherited, but we can’t override the same.

We can’t declare the constructor as final. As constructor is never inherited.

Java final class :If you make any class as final , It is not possible to inherit the same.

Polymorphism in Java:

Polymorphism means to do the single action in different ways. In Java there are 2 types of polymorphism compile time polymorphism and run – time polymorphism.

We can perform polymorphism either by method overloading or method overriding.

If we overload the static method in Java, it is an example of compile time polymorphism.

Runtime Polymorphism:

It is also known as Dynamic method dispatch is a process in which call to overridden method is resolved at the runtime rather than compile time.

In this process an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Up-casting :

When reference variable of parent class refers to the Object of child class, it is known as up-casting.

Method overridden is only possible in case of the methods, it is not a possible to override the data members.

So runtime polymorphism is not possible in case of data members.

**Static binding And Dynamic Binding:**

Connecting a method call to the method body is called binding ,There are 2 types of binding.

1.) Static binding

2.) Dynamic binding.

Static binding:

When type of object is determined at the compile time it is known as static binding. If there is any private, final or static method in a class , there is a static binding.

Dynamic binding:

When type of the object is determined at the runtime it is known as dynamic binding.(Upcasting).

Java instanceOf:

The Java instanceOf operator is basically used to check whether the object is an instance of specified type(class or subclass or interface).

The instanceOf in Java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceOf operator with any variable that has null value it returns false.

Down-casting is not possible in Java.

Abstract class in Java:

The class that is declared with the abstract keyword is known as abstract class in Java. It can have abstract and non-abstract methods(method with body).

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

In other words it hide the internal details and only shows important things.

It focus on what object does instead of how it does it.

Abstract class in less ( 0 to 100%) abstract.

While interface is (100 % )Abstract..

Abstract class :The class can be declared abstract. Once the class is declared abstract it can be either be extended and all of its method implemented.

It can’t be instantiated.

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

An abstract class can have data member, abstract method, method body, constructor and even main() method.

**Rule: If there is any abstract method in the class, then the class must be abstract.**

If we are extending any abstract class that have abstract method , you must either provide the implementation of the method or make the class abstract.

### **Another real scenario of abstract class**

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

1. **interface** A{
2. **void** a();
3. **void** b();
4. **void** c();
5. **void** d();
6. }
8. **abstract** **class** B **implements** A{
9. **public** **void** c(){System.out.println("I am C");}
10. }
12. **class** M **extends** B{
13. **public** **void** a(){System.out.println("I am a");}
14. **public** **void** b(){System.out.println("I am b");}
15. **public** **void** d(){System.out.println("I am d");}
16. }
18. **class** Test5{
19. **public** **static** **void** main(String args[]){
20. A a=**new** M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}

Output:I am a

I am b

I am c

I am d

Interface in Java:

An interface is a blueprint of class. It has static constants and abstract methods. It has a static constants and abstract methods. Interface is basically used to achieve multiple inheritance. It cannot be instantiated, it represents IS-A relationship.

Why use Java Interface?

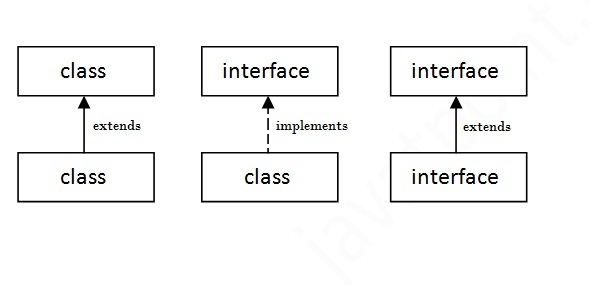
It is used to achieve abstraction

It is used to achieve loose coupling.

It is used to support multiple inheritance.

Since Java8 , we can have static and default methods in interface.

For interface, Java compiler adds public and abstract keywords before the methods and adds public, static and final keyword for the data members.



Since Java 8, we can have method body in interface. But we need to make it default method.

Since Java 8, we can have static method in interface.

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

Note: An interface can have another interface i.e. known as nested interface.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods.  Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | 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) **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

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

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.

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.

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

# **Java Static Import**

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

## Advantage of static import:

* Less coding is required if you have access any static member of a class oftenly.

## Disadvantage of static import:

* If you overuse the static import feature, it makes the program unreadable and unmaintainable.

### **What is the difference between import and static import?**

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

Access modifiers in Java:

Let's understand the access modifiers 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 |

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

**Encapsulation in Java:**

Encapsulation is the process of wrapping of data data and code into single unit.

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 fully encapsulated class.

### **Advantage of Encapsulation in java**

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

**Object class in Java:**

The Object class is the parent class in Java.

Following are the methods.

### **Methods of Object class**

|  |
| --- |
| The Object class provides many methods. They are as follows: |

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class  can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified  milliseconds, until another thread notifies (invokes notify() or  notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified  milliseconds and nanoseconds, until another thread  notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread  notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being  garbage collected. |

# **Object Cloning in Java**

The **object cloning** is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

1. **protected** Object clone() **throws** CloneNotSupportedException

### **Why use clone() method ?**

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

### **Advantage of Object cloning**

Less processing task.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

**Wrapper class in Java:**

Wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

**Call by Value and call by reference in Java:**

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

**Java StrictFP:**

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform.

To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

Example: javadoc M.java

Java Command line argument:

The java command-line argument is an argument i.e. passed at the time of running the java program.

# **Difference between object and class**

There are many differences between object and class. A list of differences between object and class are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **Object** | **Class** |
| 1) | Object is an **instance** of a class. | Class is a **blueprint or template**from which objects are created. |
| 2) | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a **group of similar objects**. |
| 3) | Object is a **physical** entity. | Class is a **logical** entity. |
| 4) | Object is created through **new keyword** mainly e.g. Student s1=new Student(); | Class is declared using **class keyword** e.g. class Student{} |
| 5) | Object is created **many times** as per requirement. | Class is declared **once**. |
| 6) | Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. |
| 7) | There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. |

# **Difference between method overloading and method overriding in java**

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | Method overloading is used to increase the readability of the program. | Method overriding is used to provide the  specific implementation of the method that is  already provided by its super class. |
| 2) | Method overloading is performed within class. | Method overriding occurs in two  classes that have IS-A (inheritance) relationship. |
| 3) | In case of method overloading, parameter must be different. | In case of method overriding, parameter must be same. |
| 4) | Method overloading is the example of compile time polymorphism. | Method overriding is the example of run  time polymorphism. |
| 5) | In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter. | Return type must be same or covariant  in method overriding. |

**Java String:**

String is basically an object that represents sequence of char values. An array of characters works same as Java String.

Java .lang.String class implements serializable, comparable and charSequence interfaces.

Java String class provides a lot of methods to perform operations on String like,

|  |  |  |
| --- | --- | --- |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string with given locale |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | returns a joined string |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | returns a joined string |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with object |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatinates specified string |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified char value |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified CharSequence |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex and limit |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | returns interned string |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index starting with given index |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index starting with given index |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | converts given type into string. It is overloaded. |

CharSequence

String

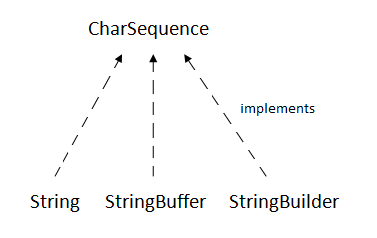
Comparable

Serializable

implements

## CharSequence Interface

The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.



The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.

### **String Literal**

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance

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

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

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

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

# **Immutable String in Java**

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

# **Difference between String and StringBuffer**

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **StringBuffer** |
| 1) | String class is immutable. | StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less  memory when you cancat strings. |
| 3) | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the  equals() method of Object class. |

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e.  not thread safe. It means two threads can call  the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

The class is immutable as:

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There is no setter methods i.e. we have no option to change the value of the instance variable.

These points makes this class as immutable.

If you want to represents any object as a String it can be done by toString () method.

If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation.

#### StringTokenizer class is deprecated now. It is recommended to use split() method of String class or regex (Regular Expression).

### **1) How many objects will be created in the following code?**

String s1="javatpoint";

String s2="javatpoint";

**Answer:** Only one.

### **2) What is the difference between equals() method and == operator?**

The equals() method matches content of the strings whereas == operator matches object or reference of the strings.

### **3) Is String class final?**

**Answer:** Yes.

### [**4) How to reverse String in java?**](https://www.javatpoint.com/how-to-reverse-string-in-java)

Input:

this is javatpoint

Output:

tnioptavaj si siht

### [**5) How to check Palindrome String in java?**](https://www.javatpoint.com/how-to-check-palindrome-string-in-java)

Input:

nitin

Output:

true

Input:

jatin

Output:

false

### [**6) Write a java program to capitalize each word in string?**](https://www.javatpoint.com/java-program-to-capitalize-each-word-in-string)

Input:

this is javatpoint

Output:

This Is Javatpoint

### [**7) Write a java program to reverse each word in string?**](https://www.javatpoint.com/java-program-to-reverse-each-word-in-string)

Input:

this is javatpoint

Output:

siht si tnioptavaj

### [**8) Write a java program to tOGGLE each word in string?**](https://www.javatpoint.com/java-program-to-toggle-each-word-in-string)

Input:

this is javatpoint

Output:

tHIS iS jAVATPOINT

### [**9) Write a java program reverse tOGGLE each word in string?**](https://www.javatpoint.com/java-program-to-reverse-toggle-each-word-in-string)

Input:

this is javatpoint

Output:

sIHT sI tNIOPTAVAJ

[**10) What is the difference between String and StringBuffer in java?**](https://www.javatpoint.com/difference-between-string-and-stringbuffer)

[**11) What is the difference between StringBuffer and StringBuilder in java?**](https://www.javatpoint.com/difference-between-stringbuffer-and-stringbuilder)

[**12) What does intern() method in java?**](https://www.javatpoint.com/java-string-intern)

[**13) How to convert String to int in java?**](https://www.javatpoint.com/java-string-to-int)

[**14) How to convert int to String in java?**](https://www.javatpoint.com/java-int-to-string)

[**15) How to convert String to Date in java?**](https://www.javatpoint.com/java-string-to-date)

[**16) How to Optimize Java String Creation?**](https://www.javatpoint.com/how-to-optimize-java-string-creation)

[**17) Java Program to check whether two Strings are anagram or not**](https://www.javatpoint.com/java-program-to-check-whether-two-strings-are-anagram-or-not)

[**18) Java program to find the percentage of uppercase, lowercase, digits and special characters in a String**](https://www.javatpoint.com/java-program-to-find-percentage-of-uppercase-lowercase-digits-and-special-characters)

[**19) How to convert String to Integer and Integer to String in Java**](https://www.javatpoint.com/how-to-convert-string-to-integer-and-integer-to-string-in-java)

[**20) Java Program to find duplicate characters in a String**](https://www.javatpoint.com/java-program-to-find-duplicate-characters-in-a-string)

[**21) Java Program to prove that strings are immutable in java**](https://www.javatpoint.com/java-program-to-prove-that-strings-are-immutable-in-java)

[**22) Java Program to remove all white spaces from a String**](https://www.javatpoint.com/java-program-to-remove-all-white-spaces-from-a-string)

[**23) Java Program to check whether one String is a rotation of another**](https://www.javatpoint.com/java-program-to-check-whether-one-string-is-a-rotation-of-another)

### [**24) Java Program to count the number of words in a String**](https://www.javatpoint.com/java-program-to-count-the-number-of-words-in-a-string)

### [**25) Java Program to reverse a given String with preserving the position of space**](https://www.javatpoint.com/java-program-to-reverse-a-string-preserving-the-position-of-space)

### [**26) How to swap two String variables without third variable**](https://www.javatpoint.com/how-to-swap-two-string-variables-without-using-third-variable-in-java)

### [**27) How to remove a particular character from a String**](https://www.javatpoint.com/how-to-remove-a-particular-character-from-a-string)

Revise of the string function.

## Java String Methods

[String charAt()](https://www.javatpoint.com/java-string-charat)

[String compareTo()](https://www.javatpoint.com/java-string-compareto)

[String concat()](https://www.javatpoint.com/java-string-concat)

[String contains()](https://www.javatpoint.com/java-string-contains)

[String endsWith()](https://www.javatpoint.com/java-string-endswith)

[String equals()](https://www.javatpoint.com/java-string-equals)

[equalsIgnoreCase()](https://www.javatpoint.com/java-string-equalsignorecase)

[String format()](https://www.javatpoint.com/java-string-format)

[String getBytes()](https://www.javatpoint.com/java-string-getbytes)

[String getChars()](https://www.javatpoint.com/java-string-getchars)

[String indexOf()](https://www.javatpoint.com/java-string-indexof)

[String intern()](https://www.javatpoint.com/java-string-intern)

[String isEmpty()](https://www.javatpoint.com/java-string-isempty)

[String join()](https://www.javatpoint.com/java-string-join)

[String lastIndexOf()](https://www.javatpoint.com/java-string-lastindexof)

[String length()](https://www.javatpoint.com/java-string-length)

[String replace()](https://www.javatpoint.com/java-string-replace)

[String replaceAll()](https://www.javatpoint.com/java-string-replaceall)

[String split()](https://www.javatpoint.com/java-string-split)

[String startsWith()](https://www.javatpoint.com/java-string-startswith)

[String substring()](https://www.javatpoint.com/java-string-substring)

[String toCharArray()](https://www.javatpoint.com/java-string-tochararray)

[String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase)

[String toUpperCase()](https://www.javatpoint.com/java-string-touppercase)

[String trim()](https://www.javatpoint.com/java-string-trim)

[String valueOf()](https://www.javatpoint.com/java-string-valueof)

# **Java Regex**

The **Java Regex** or Regular Expression is an API to *define pattern for searching or manipulating strings*.

It is widely used to define constraint on strings such as password and email validation. After learning java regex tutorial, you will be able to test your own regular expressions by the Java Regex Tester Tool.

Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.

#### **java.util.regex package**

It provides following classes and interface for regular expressions. The Matcher and Pattern classes are widely used in java regular expression.

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class

## Matcher class

It implements **MatchResult** interface. It is a *regex engine* i.e. used to perform match operations on a character sequence.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | boolean matches() | test whether the regular expression matches the pattern. |
| 2 | boolean find() | finds the next expression that matches the pattern. |
| 3 | boolean find(int start) | finds the next expression that matches the pattern from the given start number. |
| 4 | String group() | returns the matched subsequence. |
| 5 | int start() | returns the starting index of the matched subsequence. |
| 6 | int end() | returns the ending index of the matched subsequence. |
| 7 | int groupCount() | returns the total number of the matched subsequence. |

## Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | static Pattern compile(String regex) | compiles the given regex and return the instance of pattern. |
| 2 | Matcher matcher(CharSequence input) | creates a matcher that matches the given input with pattern. |
| 3 | static boolean matches(String regex, CharSequence input) | It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern. |
| 4 | String[] split(CharSequence input) | splits the given input string around matches of given pattern. |
| 5 | String pattern() | returns the regex pattern. |

## Exception handling in Java:

## Exception handling in Java is basically used to maintain the flow of the program even though it is failing in the program.

## hierarchy of exception handling

## In Java there are mainly 2 types of exception.

## 1.) Checked exceptions.

## 2.) Unchecked exception

## Checked exceptions:

## The classes that extend Throwable class except RuntimeException and error are known as checked exceptions. This are basically checked at compile time .

## Unchecked Execptions:

## The classes that extend RunTimeException are known as unchecked exception. This exception is not checked at compile time, But checked runtime.

## They are basically of 3 types: Arithmetic Exception, NULL pointer exception, ArrayIndexOutOfBound Exception.

## Error is irrecoverable.

### **Common scenarios where exceptions may occur**

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

### **1) Scenario where ArithmeticException occurs**

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

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

### **2) Scenario where NullPointerException occurs**

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

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

### **3) Scenario where NumberFormatException occurs**

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

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

### **4) Scenario where ArrayIndexOutOfBoundsException occurs**

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

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

## Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

## Java Try catch:

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

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

## Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

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

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 exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

#### Rule: At a time only one Exception is occured and at a time only one catch block is executed.

#### Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

# **Java Nested try block**

The try block within a try block is known as nested try block in java.

### **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.

# **Java finally block**

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

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

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

## Why use java finally

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

#### Rule: For each try block there can be zero or more catch blocks, but only one finally block.

#### Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

## Java throw Keyword:

## Java throw keyword is explicitly used to throw the exception, It can be checked, unchecked or custom exception.

# **Java Exception propagation**

|  |
| --- |
| An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation. |

#### Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

#### Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

## Java Throws keyword:

## The Java throws keyword is used to declare the exception. It gives information to the programmer that there may occur the exception so it is better for the programmer to provide the default the handling for the same.

## Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

## It help to propagate the checked exceptions.

## We can re-throw the exception in catch block.

# **Difference between throw and throws in Java**

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple  exceptions e.g. public void method()throws IOException,SQLException. |

# **Difference between final, finally and finalize**

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **final** | **finally** | **finalize** |
| 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform  clean up processing just before  object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

* **If the superclass method does not declare an exception**
  + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
* **If the superclass method declares an exception**
  + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

#### 1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

#### 2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

### **If the superclass method declares an exception**

#### 1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

1. **class** InvalidAgeException **extends** Exception{
2. InvalidAgeException(String s){
3. **super**(s);
4. }
5. }
6. **class** TestCustomException1{
8. **static** **void** validate(**int** age)**throws** InvalidAgeException{
9. **if**(age<18)
10. **throw** **new** InvalidAgeException("not valid");
11. **else**
12. System.out.println("welcome to vote");
13. }
15. **public** **static** **void** main(String args[]){
16. **try**{
17. validate(13);
18. }**catch**(Exception m){System.out.println("Exception occured: "+m);}
20. System.out.println("rest of the code...");
21. }
22. }

Output:Exception occured: InvalidAgeException:not valid

rest of the code...

# **ava Anonymous inner class**

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

### **Java anonymous inner class example using class**

1. **abstract** **class** Person{
2. **abstract** **void** eat();
3. }
4. **class** TestAnonymousInner{
5. **public** **static** **void** main(String args[]){
6. Person p=**new** Person(){
7. **void** eat(){System.out.println("nice fruits");}
8. };
9. p.eat();
10. }
11. }

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

Output:

nice fruits

## Internal working of given code

1. Person p=**new** Person(){
2. **void** eat(){System.out.println("nice fruits");}
3. };
4. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
5. An object of Anonymous class is created that is referred by p reference variable of Person type.

## Internal class generated by the compiler

1. **import** java.io.PrintStream;
2. **static** **class** TestAnonymousInner$1 **extends** Person
3. {
4. TestAnonymousInner$1(){}
5. **void** eat()
6. {
7. System.out.println("nice fruits");
8. }
9. }

## Java anonymous inner class example using interface

1. **interface** Eatable{
2. **void** eat();
3. }
4. **class** TestAnnonymousInner1{
5. **public** **static** **void** main(String args[]){
6. Eatable e=**new** Eatable(){
7. **public** **void** eat(){System.out.println("nice fruits");}
8. };
9. e.eat();
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnonymousInner1)

Output:

nice fruits

### **Internal working of given code**

It performs two main tasks behind this code:

1. Eatable p=**new** Eatable(){
2. **void** eat(){System.out.println("nice fruits");}
3. };
4. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
5. An object of Anonymous class is created that is referred by p reference variable of Eatable type.

### **Internal class generated by the compiler**

1. **import** java.io.PrintStream;
2. **static** **class** TestAnonymousInner1$1 **implements** Eatable
3. {
4. TestAnonymousInner1$1(){}
5. **void** eat(){System.out.println("nice fruits");}
6. }

# **Java Local inner class**

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

#### Rule: Local variable can't be private, public or protected.

## Rules for Java Local Inner class

#### 1) Local inner class cannot be invoked from outside the method.

#### 2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.

# **Java static nested class**

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

* It can access static data members of outer class including private.
* Static nested class cannot access non-static (instance) data member or method.

## Java static nested class example with static method

If you have the static member inside static nested class, you don't need to create instance of static nested class.

# **Java Nested Interface**

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

### **Points to remember for nested interfaces**

There are given some points that should be remembered by the java programmer.

* Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
* Nested interfaces are declared static implicitly.

### **Can we define a class inside the interface?**

Yes, If we define a class inside the interface, java compiler creates a static nested class. Let's see how can we define a class within the interface:

## Multithreading in Java:

## Muliti- threading is executing multiple threads simultaneously.

## Thread is the light weight sub-process, a smallest unit of processing. Multi-processing and multi-threading both are used to achieve multi-tasking.

## Each thread can have different path for execution, as they are independent.

## The thread are preferred over process because they share the memory area, they don’t allocate different memory.

## Advantages:

## It does not block the user because thread are independent and you can perform multiple operations at same time.

## Threads are in independent so it does not affect other thread if exception occur in the single thread.

## Multitasking can be achieved by 2 ways.

## Process based multitasking.

## Thread based multitasking.

## Process based multitasking:

## Each process have its own memory.

## They are heavy weight.

## Switching from one processes to another is high, as cost of communication between the processes is high.

## Thread-based multitasking:

## Thread shares same address space.

## They are light weight process.

## The switching from one thread is not costly as the cost of communication between the processes is low.

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

## Life cycle of thread:

## The thread can have basically 5 stages.

## New

## Runnable

## Running

## Non-Running(blocked)

## Terminated.

## New: The Thread is in new state if you create an instance of thread class but before the invocation of the start method.

## Runnable:

## When the thread is in the runnable state after the invocation of start method , but the thread scheduler has not selected it to be the running thread.

## Running: When the thread is in the running thread and scheduler has selected the thread for the execution.

## Non-Running :When the thread is alive, but currently not eligible to run.

## Terminated: when the thread is terminated and run method is exits.

## Creation of thread:

## In Java we can create thread by 2 ways,

## By implementing the runnable interface.

## By extending the Thread class.

## Thread class:

## Thread class provide constructors and method 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

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).
18. **public boolean isDaemon():**tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread.
20. **public void interrupt():**interrupts the thread.
21. **public boolean isInterrupted():**tests if the thread has been interrupted.
22. **public static boolean interrupted():**tests if the current thread has been interrupted.

## Runnable interface:

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

## Public void run(): is used to perform action for a thread.

## Starting a thread:

## Start() method of thread class is used to start a newly created start. It performs 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.

## 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 Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

## Thread scheduler:

## Thread scheduler in Java is a part of the JVM that decides which thread would run.

## There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

## Only one thread at a time will run in a single process. The thread scheduler mainly uses preemptive or time slicing to schedule the threads.

## Sleep method in Java:

## The sleep method of thread class is used to sleep a thread for the specified amount of time.

## Public static void sleep(long millisec) throws interrupted exception.

## Public static void sleep(long millisec,int nanos) throws interrupted exception.

## Starting thread twice:

## We can’t start thread once started. If the thread is already running ,we can’t start the thread already running.

## It gives IllegalThreadException.

## If we call run() method directly instead of start() method, The process doesn’t get the new stack, instead all the variables and method get into the stack of the currently running thread.

## As when the new thread is started with the start method, Each thread starts in a separate call stack.

Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

## Join():

## The join method waits for the thread to die,It causes currently running thread to stop executing until the thread it joins with completes it task.

## In other words , we will able to sequential arrange the running of thread with the help of join method.

|  |
| --- |
| public void join()throws InterruptedException |
| public void join(long milliseconds)throws InterruptedException |

|  |
| --- |
| public String getName() |
| public void setName(String name) |
| public long getId() |

### **The currentThread() method:**

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

## By default each thread have name like thread -0 , thread- 1 we can change the name of the thread by setName() method

## And we can get the name of the method by getName().

## Public String getName()

## Public void setName(String name)

## Public static Thread currentThread():

## This method references current thread.

## Priority of the Thread:There are 3 types of priority assigned to the thread.

## 1.)Min\_priority

## 2.)Norm\_priority

## 3.)Max\_priority.

## Normally each thread have the norm\_prioroty.

## The min\_priority is 1 and max\_priority is 10.

## Demon thread in Java:

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

## JVM terminates Deomon thread if there is no other thread running.

 java.lang.Thread class provides two methods for java daemon thread.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread as daemon thread or user thread. |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

#### Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

## Java Thread pool:

## Java thread pool represents the a group of worker threads that are the use many times and waiting for the job.

## In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

## Advantages :

## Better performance It saves time because there is no need to create new thread.

#### **Example of Java Thread Pool**

Let's see a simple example of java thread pool using ExecutorService and Executors.

*File: WorkerThread.java*

1. **import** java.util.concurrent.ExecutorService;
2. **import** java.util.concurrent.Executors;
3. **class** WorkerThread **implements** Runnable {
4. **private** String message;
5. **public** WorkerThread(String s){
6. **this**.message=s;
7. }
8. **public** **void** run() {
9. System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
10. processmessage();//call processmessage method that sleeps the thread for 2 seconds
11. System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
12. }
13. **private** **void** processmessage() {
14. **try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) { e.printStackTrace(); }
15. }
16. }

*File: JavaThreadPoolExample.java*

1. **public** **class** TestThreadPool {
2. **public** **static** **void** main(String[] args) {
3. ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads
4. **for** (**int** i = 0; i < 10; i++) {
5. Runnable worker = **new** WorkerThread("" + i);
6. executor.execute(worker);//calling execute method of ExecutorService
7. }
8. executor.shutdown();
9. **while** (!executor.isTerminated()) {   }
11. System.out.println("Finished all threads");
12. }
13. }

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Java thread group is implemented by java.lang.ThreadGroup class.

## Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

|  |  |  |
| --- | --- | --- |
| **No.** | **Constructor** | **Description** |
| 1) | ThreadGroup(String name) | creates a thread group with given name. |
| 2) | ThreadGroup(ThreadGroup parent, String name) | creates a thread group with given parent group and name. |

## Important methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | int activeCount() | returns no. of threads running in current group. |
| 2) | int activeGroupCount() | returns a no. of active group in this thread group. |
| 3) | void destroy() | destroys this thread group and all its sub groups. |
| 4) | String getName() | returns the name of this group. |
| 5) | ThreadGroup getParent() | returns the parent of this group. |
| 6) | void interrupt() | interrupts all threads of this group. |
| 7) | void list() | prints information of this group to standard console. |

Let's see a code to group multiple threads.

1. ThreadGroup tg1 = **new** ThreadGroup("Group A");
2. Thread t1 = **new** Thread(tg1,**new** MyRunnable(),"one");
3. Thread t2 = **new** Thread(tg1,**new** MyRunnable(),"two");
4. Thread t3 = **new** Thread(tg1,**new** MyRunnable(),"three");

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

1. Thread.currentThread().getThreadGroup().interrupt();

## ThreadGroup Example

*File: ThreadGroupDemo.java*

1. **public** **class** ThreadGroupDemo **implements** Runnable{
2. **public** **void** run() {
3. System.out.println(Thread.currentThread().getName());
4. }
5. **public** **static** **void** main(String[] args) {
6. ThreadGroupDemo runnable = **new** ThreadGroupDemo();
7. ThreadGroup tg1 = **new** ThreadGroup("Parent ThreadGroup");
9. Thread t1 = **new** Thread(tg1, runnable,"one");
10. t1.start();
11. Thread t2 = **new** Thread(tg1, runnable,"two");
12. t2.start();
13. Thread t3 = **new** Thread(tg1, runnable,"three");
14. t3.start();
16. System.out.println("Thread Group Name: "+tg1.getName());
17. tg1.list();
19. }
20. }

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

### **When does the JVM shut down?**

The JVM shuts down when:

* user presses ctrl+c on the command prompt
* System.exit(int) method is invoked
* user logoff
* user shutdown etc.

#### **The addShutdownHook(Thread hook) method**

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine. Syntax:

1. **public** **void** addShutdownHook(Thread hook){}

The object of Runtime class can be obtained by calling the static factory method getRuntime(). For example:

Runtime r = Runtime.getRuntime();

#### **Factory method**

The method that returns the instance of a class is known as factory method.

### **Simple example of Shutdown Hook**

1. **class** MyThread **extends** Thread{
2. **public** **void** run(){
3. System.out.println("shut down hook task completed..");
4. }
5. }
7. **public** **class** TestShutdown1{
8. **public** **static** **void** main(String[] args)**throws** Exception {
10. Runtime r=Runtime.getRuntime();
11. r.addShutdownHook(**new** MyThread());
13. System.out.println("Now main sleeping... press ctrl+c to exit");
14. **try**{Thread.sleep(3000);}**catch** (Exception e) {}
15. }
16. }

#### Note: The shutdown sequence can be stopped by invoking the halt(int) method of Runtime class.

### **Same example of Shutdown Hook by annonymous class:**

1. **public** **class** TestShutdown2{
2. **public** **static** **void** main(String[] args)**throws** Exception {
4. Runtime r=Runtime.getRuntime();
6. r.addShutdownHook(**new** Thread(){
7. **public** **void** run(){
8. System.out.println("shut down hook task completed..");
9. }
10. }
11. );
13. System.out.println("Now main sleeping... press ctrl+c to exit");
14. **try**{Thread.sleep(3000);}**catch** (Exception e) {}
15. }
16. }

# **How to perform single task by multiple threads?**

|  |
| --- |
| If you have to perform single task by many threads, have only one run() method.For example: |

***Program of performing single task by multiple threads***

1. **class** TestMultitasking1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("task one");
4. }
5. **public** **static** **void** main(String args[]){
6. TestMultitasking1 t1=**new** TestMultitasking1();
7. TestMultitasking1 t2=**new** TestMultitasking1();
8. TestMultitasking1 t3=**new** TestMultitasking1();
10. t1.start();
11. t2.start();
12. t3.start();
13. }
14. }

***Program of performing single task by multiple threads***

1. **class** TestMultitasking2 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("task one");
4. }
6. **public** **static** **void** main(String args[]){
7. Thread t1 =**new** Thread(**new** TestMultitasking2());//passing annonymous object of TestMultitasking2 class
8. Thread t2 =**new** Thread(**new** TestMultitasking2());
10. t1.start();
11. t2.start();
13. }
14. }

### **How to perform multiple tasks by multiple threads (multitasking in multithreading)?**

|  |
| --- |
| If you have to perform multiple tasks by multiple threads,have multiple run() methods.For example: |

***Program of performing two tasks by two threads***

1. **class** Simple1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("task one");
4. }
5. }
7. **class** Simple2 **extends** Thread{
8. **public** **void** run(){
9. System.out.println("task two");
10. }
11. }
13. **class** TestMultitasking3{
14. **public** **static** **void** main(String args[]){
15. Simple1 t1=**new** Simple1();
16. Simple2 t2=**new** Simple2();
18. t1.start();
19. t2.start();
20. }
21. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultitasking3)

Output:task one

task two

### **Same example as above by annonymous class that extends Thread class:**

***Program of performing two tasks by two threads***

1. **class** TestMultitasking4{
2. **public** **static** **void** main(String args[]){
3. Thread t1=**new** Thread(){
4. **public** **void** run(){
5. System.out.println("task one");
6. }
7. };
8. Thread t2=**new** Thread(){
9. **public** **void** run(){
10. System.out.println("task two");
11. }
12. };

15. t1.start();
16. t2.start();
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultitasking4)

Output:task one

task two

### **Same example as above by annonymous class that implements Runnable interface:**

***Program of performing two tasks by two threads***

1. **class** TestMultitasking5{
2. **public** **static** **void** main(String args[]){
3. Runnable r1=**new** Runnable(){
4. **public** **void** run(){
5. System.out.println("task one");
6. }
7. };
9. Runnable r2=**new** Runnable(){
10. **public** **void** run(){
11. System.out.println("task two");
12. }
13. };
15. Thread t1=**new** Thread(r1);
16. Thread t2=**new** Thread(r2);
18. t1.start();
19. t2.start();
20. }
21. }

# **Java Garbage Collection**

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### **Advantage of Garbage Collection**

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## How can an object be unreferenced?

There are many ways:

* By nulling the reference
* By assigning a reference to another
* By annonymous object etc.

### **1) By nulling a reference:**

1. Employee e=**new** Employee();
2. e=**null**;

### **2) By assigning a reference to another:**

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

### **3) By annonymous object:**

1. **new** Employee();

## finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. **protected** **void** finalize(){}

#### Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. **public** **static** **void** gc(){}

#### Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

### **Simple Example of garbage collection in java**

1. **public** **class** TestGarbage1{
2. **public** **void** finalize(){System.out.println("object is garbage collected");}
3. **public** **static** **void** main(String args[]){
4. TestGarbage1 s1=**new** TestGarbage1();
5. TestGarbage1 s2=**new** TestGarbage1();
6. s1=**null**;
7. s2=**null**;
8. System.gc();
9. }
10. }

#### Note: Neither finalization nor garbage collection is guaranteed.

**Java Runtime** class is used to interact with java runtime environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

## Important methods of Java Runtime class

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public static Runtime getRuntime() | returns the instance of Runtime class. |
| 2) | public void exit(int status) | terminates the current virtual machine. |
| 3) | public void addShutdownHook(Thread hook) | registers new hook thread. |
| 4) | public Process exec(String command)throws IOException | executes given command in a separate process. |
| 5) | public int availableProcessors() | returns no. of available processors. |
| 6) | public long freeMemory() | returns amount of free memory in JVM. |
| 7) | public long totalMemory() | returns amount of total memory in JVM. |

## Java Runtime exec() method

1. **public** **class** Runtime1{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("notepad");//will open a new notepad
4. }
5. }

## How to shutdown system in Java

You can use shutdown -s command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. c:\\Windows\\System32\\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system and -t switch to specify time delay.

1. **public** **class** Runtime2{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("shutdown -s -t 0");
4. }
5. }

## How to shutdown windows system in Java

1. **public** **class** Runtime2{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");
4. }
5. }

## How to restart system in Java

1. **public** **class** Runtime3{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("shutdown -r -t 0");
4. }
5. }

## Java Runtime availableProcessors()

1. **public** **class** Runtime4{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. System.out.println(Runtime.getRuntime().availableProcessors());
4. }
5. }

## Java Runtime freeMemory() and totalMemory() method

In the given program, after creating 10000 instance, free memory will be less than the previous free memory. But after gc() call, you will get more free memory.

1. **public** **class** MemoryTest{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime r=Runtime.getRuntime();
4. System.out.println("Total Memory: "+r.totalMemory());
5. System.out.println("Free Memory: "+r.freeMemory());
7. **for**(**int** i=0;i<10000;i++){
8. **new** MemoryTest();
9. }
10. System.out.println("After creating 10000 instance, Free Memory: "+r.freeMemory());
11. System.gc();
12. System.out.println("After gc(), Free Memory: "+r.freeMemory());
13. }
14. }

# **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. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

### **Concept of Lock in Java**

Synchronization is built around an internal entity known as the lock or monitor. Every object has an 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:

1. Class Table{
3. **void** printTable(**int** n){//method not synchronized
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. **class** TestSynchronization1{
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

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.

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

### **Example of synchronized method by using annonymous class**

In this program, we have created the two threads by annonymous class, so less coding is required.

1. //Program of synchronized method by using annonymous class
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. **public** **class** TestSynchronization3{
15. **public** **static** **void** main(String args[]){
16. **final** Table obj = **new** Table();//only one object
18. Thread t1=**new** Thread(){
19. **public** **void** run(){
20. obj.printTable(5);
21. }
22. };
23. Thread t2=**new** Thread(){
24. **public** **void** run(){
25. obj.printTable(100);
26. }
27. };
29. t1.start();
30. t2.start();
31. }
32. }

Output: 5

10

15

20

25

100

200

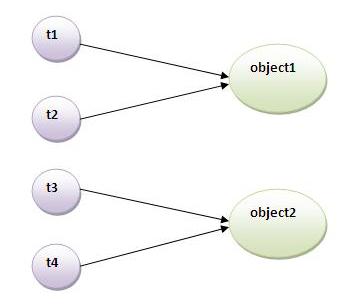
300

400

500

# **Static synchronization**

If you make any static method as synchronized, the lock will be on the class not on object.



### **Problem without static synchronization**

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

### **Example of static synchronization**

In this example we are applying synchronized keyword on the static method to perform static synchronization.

1. **class** Table{
3. **synchronized** **static** **void** printTable(**int** n){
4. **for**(**int** i=1;i<=10;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){}
9. }
10. }
11. }
13. **class** MyThread1 **extends** Thread{
14. **public** **void** run(){
15. Table.printTable(1);
16. }
17. }
19. **class** MyThread2 **extends** Thread{
20. **public** **void** run(){
21. Table.printTable(10);
22. }
23. }
25. **class** MyThread3 **extends** Thread{
26. **public** **void** run(){
27. Table.printTable(100);
28. }
29. }



34. **class** MyThread4 **extends** Thread{
35. **public** **void** run(){
36. Table.printTable(1000);
37. }
38. }
40. **public** **class** TestSynchronization4{
41. **public** **static** **void** main(String t[]){
42. MyThread1 t1=**new** MyThread1();
43. MyThread2 t2=**new** MyThread2();
44. MyThread3 t3=**new** MyThread3();
45. MyThread4 t4=**new** MyThread4();
46. t1.start();
47. t2.start();
48. t3.start();
49. t4.start();
50. }
51. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSynchronization4)

Output: 1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

### **Same example of static synchronization by annonymous class**

In this example, we are using annonymous class to create the threads.

1. **class** Table{
3. **synchronized** **static**  **void** printTable(**int** n){
4. **for**(**int** i=1;i<=10;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){}
9. }
10. }
11. }
13. **public** **class** TestSynchronization5 {
14. **public** **static** **void** main(String[] args) {
16. Thread t1=**new** Thread(){
17. **public** **void** run(){
18. Table.printTable(1);
19. }
20. };
22. Thread t2=**new** Thread(){
23. **public** **void** run(){
24. Table.printTable(10);
25. }
26. };
28. Thread t3=**new** Thread(){
29. **public** **void** run(){
30. Table.printTable(100);
31. }
32. };
34. Thread t4=**new** Thread(){
35. **public** **void** run(){
36. Table.printTable(1000);
37. }
38. };
39. t1.start();
40. t2.start();
41. t3.start();
42. t4.start();
44. }
45. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSynchronization5)

Output: 1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

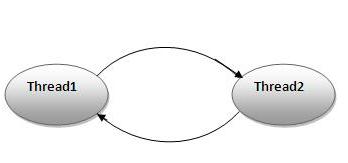
### **Synchronized block on a class lock:**

The block synchronizes on the lock of the object denoted by the reference .class name .class. A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

1. **static** **void** printTable(**int** n) {
2. **synchronized** (Table.**class**) {       // Synchronized block on class A
3. // ...
4. }
5. }

# **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**

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. }

# **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**

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

### **2) 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:

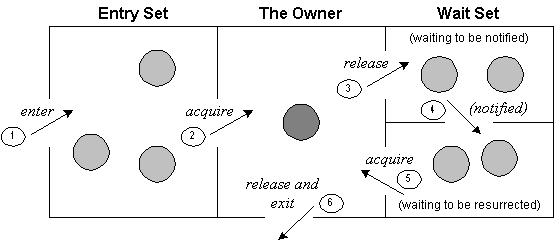
public final void notify()

### **3) notifyAll() method**

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

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()** |
| wait() method releases the lock | sleep() method doesn't release the lock. |
| is the method of Object class | is the method of Thread class |
| is the non-static method | is the static method |
| is the non-static method | is the static method |
| 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.

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. }}

# **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 boolean isInterrupted()** |

## Example of interrupting a thread that stops working

|  |
| --- |
| In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception. |

1. **class** TestInterruptingThread1 **extends** Thread{
2. **public** **void** run(){
3. **try**{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }**catch**(InterruptedException e){
7. **throw** **new** RuntimeException("Thread interrupted..."+e);
8. }
10. }
12. **public** **static** **void** main(String args[]){
13. TestInterruptingThread1 t1=**new** TestInterruptingThread1();
14. t1.start();
15. **try**{
16. t1.interrupt();
17. }**catch**(Exception e){System.out.println("Exception handled "+e);}
19. }
20. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterruptingThread1)

[download this example](https://www.javatpoint.com/src/multi/interrupt1.zip)

Output:Exception in thread-0

java.lang.RuntimeException: Thread interrupted...

java.lang.InterruptedException: sleep interrupted

at A.run(A.java:7)

## Example of interrupting a thread that doesn't stop working

|  |
| --- |
| In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working. |

1. **class** TestInterruptingThread2 **extends** Thread{
2. **public** **void** run(){
3. **try**{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }**catch**(InterruptedException e){
7. System.out.println("Exception handled "+e);
8. }
9. System.out.println("thread is running...");
10. }
12. **public** **static** **void** main(String args[]){
13. TestInterruptingThread2 t1=**new** TestInterruptingThread2();
14. t1.start();
16. t1.interrupt();
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterruptingThread2)

[download this example](https://www.javatpoint.com/src/multi/interrupt2.zip)

Output:Exception handled

java.lang.InterruptedException: sleep interrupted

thread is running...

## Example of interrupting thread that behaves normally

|  |
| --- |
| If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later. |

1. **class** TestInterruptingThread3 **extends** Thread{
3. **public** **void** run(){
4. **for**(**int** i=1;i<=5;i++)
5. System.out.println(i);
6. }
8. **public** **static** **void** main(String args[]){
9. TestInterruptingThread3 t1=**new** TestInterruptingThread3();
10. t1.start();
12. t1.interrupt();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterruptingThread3)

Output:1

2

3

4

5

## What about isInterrupted and interrupted method?

|  |
| --- |
| The isInterrupted() method returns the interrupted flag either true or false. The static interrupted() method returns the interrupted flag afterthat it sets the flag to false if it is true. |

1. **public** **class** TestInterruptingThread4 **extends** Thread{
3. **public** **void** run(){
4. **for**(**int** i=1;i<=2;i++){
5. **if**(Thread.interrupted()){
6. System.out.println("code for interrupted thread");
7. }
8. **else**{
9. System.out.println("code for normal thread");
10. }
12. }//end of for loop
13. }
15. **public** **static** **void** main(String args[]){
17. TestInterruptingThread4 t1=**new** TestInterruptingThread4();
18. TestInterruptingThread4 t2=**new** TestInterruptingThread4();
20. t1.start();
21. t1.interrupt();
23. t2.start();
25. }
26. }

According to Sun Microsystems, **Java monitors are reentrant** means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

#### **Advantage of Reentrant Monitor**

It eliminates the possibility of single thread deadlocking

Let's understand the java reentrant monitor by the example given below:

1. **class** Reentrant {
2. **public** **synchronized** **void** m() {
3. n();
4. System.out.println("this is m() method");
5. }
6. **public** **synchronized** **void** n() {
7. System.out.println("this is n() method");
8. }
9. }

In this class, m and n are the synchronized methods. The m() method internally calls the n() method.

Now let's call the m() method on a thread. In the class given below, we are creating thread using annonymous class.

1. **public** **class** ReentrantExample{
2. **public** **static** **void** main(String args[]){
3. **final** ReentrantExample re=**new** ReentrantExample();
5. Thread t1=**new** Thread(){
6. **public** **void** run(){
7. re.m();//calling method of Reentrant class
8. }
9. };
10. t1.start();
11. }}

|  |  |
| --- | --- |
| [**BlockingDeque**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingDeque.html)<E> | A [**Deque**](https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html) that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element. |
| [**BlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html)<E> | A [**Queue**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element. |
| [**Callable**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html)<V> | A task that returns a result and may throw an exception. |
| [**CompletionService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CompletionService.html)<V> | A service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks. |
| [**ConcurrentMap**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html)<K,V> | A [**Map**](https://docs.oracle.com/javase/7/docs/api/java/util/Map.html) providing additional atomic putIfAbsent, remove, and replace methods. |
| [**ConcurrentNavigableMap**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentNavigableMap.html)<K,V> | A [**ConcurrentMap**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html) supporting [**NavigableMap**](https://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html) operations, and recursively so for its navigable sub-maps. |
| [**Delayed**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Delayed.html) | A mix-in style interface for marking objects that should be acted upon after a given delay. |
| [**Executor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) | An object that executes submitted [**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) tasks. |
| [**ExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) | An [**Executor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) that provides methods to manage termination and methods that can produce a [**Future**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html) for tracking progress of one or more asynchronous tasks. |
| [**ForkJoinPool.ForkJoinWorkerThreadFactory**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.ForkJoinWorkerThreadFactory.html) | Factory for creating new [**ForkJoinWorkerThread**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinWorkerThread.html)s. |
| [**ForkJoinPool.ManagedBlocker**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html) | Interface for extending managed parallelism for tasks running in [**ForkJoinPool**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html)s. |
| [**Future**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html)<V> | A Future represents the result of an asynchronous computation. |
| [**RejectedExecutionHandler**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RejectedExecutionHandler.html) | A handler for tasks that cannot be executed by a [**ThreadPoolExecutor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html). |
| [**RunnableFuture**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RunnableFuture.html)<V> | A [**Future**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html) that is [**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html). |
| [**RunnableScheduledFuture**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RunnableScheduledFuture.html)<V> | A [**ScheduledFuture**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledFuture.html) that is [**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html). |
| [**ScheduledExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html) | An [**ExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) that can schedule commands to run after a given delay, or to execute periodically. |
| [**ScheduledFuture**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledFuture.html)<V> | A delayed result-bearing action that can be cancelled. |
| [**ThreadFactory**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadFactory.html) | An object that creates new threads on demand. |
| [**TransferQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TransferQueue.html)<E> | A [**BlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) in which producers may wait for consumers to receive elements. |

|  |  |
| --- | --- |
| **Class Summary** | |
| **Class** | **Description** |
| [**AbstractExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/AbstractExecutorService.html) | Provides default implementations of [**ExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) execution methods. |
| [**ArrayBlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html)<E> | A bounded [**blocking queue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) backed by an array. |
| [**ConcurrentHashMap**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html)<K,V> | A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates. |
| [**ConcurrentLinkedDeque**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html)<E> | An unbounded concurrent [**deque**](https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html) based on linked nodes. |
| [**ConcurrentLinkedQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html)<E> | An unbounded thread-safe [**queue**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) based on linked nodes. |
| [**ConcurrentSkipListMap**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html)<K,V> | A scalable concurrent [**ConcurrentNavigableMap**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentNavigableMap.html) implementation. |
| [**ConcurrentSkipListSet**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html)<E> | A scalable concurrent [**NavigableSet**](https://docs.oracle.com/javase/7/docs/api/java/util/NavigableSet.html) implementation based on a [**ConcurrentSkipListMap**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html). |
| [**CopyOnWriteArrayList**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html)<E> | A thread-safe variant of [**ArrayList**](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html) in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array. |
| [**CopyOnWriteArraySet**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArraySet.html)<E> | A [**Set**](https://docs.oracle.com/javase/7/docs/api/java/util/Set.html) that uses an internal [**CopyOnWriteArrayList**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) for all of its operations. |
| [**CountDownLatch**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html) | A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. |
| [**CyclicBarrier**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html) | A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. |
| [**DelayQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/DelayQueue.html)<E extends [**Delayed**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Delayed.html)> | An unbounded [**blocking queue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) of Delayed elements, in which an element can only be taken when its delay has expired. |
| [**Exchanger**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Exchanger.html)<V> | A synchronization point at which threads can pair and swap elements within pairs. |
| [**ExecutorCompletionService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorCompletionService.html)<V> | A [**CompletionService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CompletionService.html) that uses a supplied [**Executor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) to execute tasks. |
| [**Executors**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html) | Factory and utility methods for [**Executor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html), [**ExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html), [**ScheduledExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html), [**ThreadFactory**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadFactory.html), and [**Callable**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html) classes defined in this package. |
| [**ForkJoinPool**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html) | An [**ExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) for running [**ForkJoinTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html)s. |
| [**ForkJoinTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html)<V> | Abstract base class for tasks that run within a [**ForkJoinPool**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html). |
| [**ForkJoinWorkerThread**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinWorkerThread.html) | A thread managed by a [**ForkJoinPool**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html), which executes [**ForkJoinTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html)s. |
| [**FutureTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html)<V> | A cancellable asynchronous computation. |
| [**LinkedBlockingDeque**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingDeque.html)<E> | An optionally-bounded [**blocking deque**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingDeque.html) based on linked nodes. |
| [**LinkedBlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html)<E> | An optionally-bounded [**blocking queue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) based on linked nodes. |
| [**LinkedTransferQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedTransferQueue.html)<E> | An unbounded [**TransferQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TransferQueue.html) based on linked nodes. |
| [**Phaser**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Phaser.html) | A reusable synchronization barrier, similar in functionality to [**CyclicBarrier**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html) and [**CountDownLatch**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html) but supporting more flexible usage. |
| [**PriorityBlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html)<E> | An unbounded [**blocking queue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) that uses the same ordering rules as class [**PriorityQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html) and supplies blocking retrieval operations. |
| [**RecursiveAction**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RecursiveAction.html) | A recursive resultless [**ForkJoinTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html). |
| [**RecursiveTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RecursiveTask.html)<V> | A recursive result-bearing [**ForkJoinTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html). |
| [**ScheduledThreadPoolExecutor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html) | A [**ThreadPoolExecutor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html) that can additionally schedule commands to run after a given delay, or to execute periodically. |
| [**Semaphore**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html) | A counting semaphore. |
| [**SynchronousQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html)<E> | A [**blocking queue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. |
| [**ThreadLocalRandom**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadLocalRandom.html) | A random number generator isolated to the current thread. |
| [**ThreadPoolExecutor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html) | An [**ExecutorService**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) that executes each submitted task using one of possibly several pooled threads, normally configured using [**Executors**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html) factory methods. |
| [**ThreadPoolExecutor.AbortPolicy**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.AbortPolicy.html) | A handler for rejected tasks that throws a RejectedExecutionException. |
| [**ThreadPoolExecutor.CallerRunsPolicy**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.CallerRunsPolicy.html) | A handler for rejected tasks that runs the rejected task directly in the calling thread of the execute method, unless the executor has been shut down, in which case the task is discarded. |
| [**ThreadPoolExecutor.DiscardOldestPolicy**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.DiscardOldestPolicy.html) | A handler for rejected tasks that discards the oldest unhandled request and then retries execute, unless the executor is shut down, in which case the task is discarded. |
| [**ThreadPoolExecutor.DiscardPolicy**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.DiscardPolicy.html) | A handler for rejected tasks that silently discards the rejected task. |

|  |  |
| --- | --- |
| **Enum Summary** | |
| **Enum** | **Description** |
| [**TimeUnit**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html) | A TimeUnit represents time durations at a given unit of granularity and provides utility methods to convert across units, and to perform timing and delay operations in these units. |

|  |  |
| --- | --- |
| **Exception Summary** | |
| **Exception** | **Description** |
| [**BrokenBarrierException**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BrokenBarrierException.html) | Exception thrown when a thread tries to wait upon a barrier that is in a broken state, or which enters the broken state while the thread is waiting. |
| [**CancellationException**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CancellationException.html) | Exception indicating that the result of a value-producing task, such as a [**FutureTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html), cannot be retrieved because the task was cancelled. |
| [**ExecutionException**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutionException.html) | Exception thrown when attempting to retrieve the result of a task that aborted by throwing an exception. |
| [**RejectedExecutionException**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RejectedExecutionException.html) | Exception thrown by an [**Executor**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) when a task cannot be accepted for execution. |
| [**TimeoutException**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeoutException.html) | Exception thrown when a blocking operation times out. |

## Package java.util.concurrent Description

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the [java.util.concurrent.locks](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/package-summary.html) and [java.util.concurrent.atomic](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html) packages.

## Executors

**Interfaces.** [Executor](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) is a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous IO, and lightweight task frameworks. Depending on which concrete Executor class is being used, tasks may execute in a newly created thread, an existing task-execution thread, or the thread calling [execute](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html#execute(java.lang.Runnable)), and may execute sequentially or concurrently. [ExecutorService](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) provides a more complete asynchronous task execution framework. An ExecutorService manages queuing and scheduling of tasks, and allows controlled shutdown. The [ScheduledExecutorService](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html) subinterface and associated interfaces add support for delayed and periodic task execution. ExecutorServices provide methods arranging asynchronous execution of any function expressed as [Callable](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html), the result-bearing analog of [Runnable](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html). A [Future](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html) returns the results of a function, allows determination of whether execution has completed, and provides a means to cancel execution. A [RunnableFuture](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RunnableFuture.html) is a Future that possesses a run method that upon execution, sets its results.

**Implementations.** Classes [ThreadPoolExecutor](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html) and [ScheduledThreadPoolExecutor](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html) provide tunable, flexible thread pools. The [Executors](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html) class provides factory methods for the most common kinds and configurations of Executors, as well as a few utility methods for using them. Other utilities based on Executors include the concrete class [FutureTask](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html) providing a common extensible implementation of Futures, and [ExecutorCompletionService](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorCompletionService.html), that assists in coordinating the processing of groups of asynchronous tasks.

Class [ForkJoinPool](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html) provides an Executor primarily designed for processing instances of [ForkJoinTask](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html) and its subclasses. These classes employ a work-stealing scheduler that attains high throughput for tasks conforming to restrictions that often hold in computation-intensive parallel processing.

## Queues

The [ConcurrentLinkedQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html) class supplies an efficient scalable thread-safe non-blocking FIFO queue.

Five implementations in java.util.concurrent support the extended [BlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) interface, that defines blocking versions of put and take: [LinkedBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [ArrayBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [SynchronousQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html), [PriorityBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html), and[DelayQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/DelayQueue.html). The different classes cover the most common usage contexts for producer-consumer, messaging, parallel tasking, and related concurrent designs.

Extended interface [TransferQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TransferQueue.html), and implementation [LinkedTransferQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedTransferQueue.html) introduce a synchronous transfer method (along with related features) in which a producer may optionally block awaiting its consumer.

The [BlockingDeque](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingDeque.html) interface extends BlockingQueue to support both FIFO and LIFO (stack-based) operations. Class [LinkedBlockingDeque](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingDeque.html) provides an implementation.

## Timing

The [TimeUnit](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html) class provides multiple granularities (including nanoseconds) for specifying and controlling time-out based operations. Most classes in the package contain operations based on time-outs in addition to indefinite waits. In all cases that time-outs are used, the time-out specifies the minimum time that the method should wait before indicating that it timed-out. Implementations make a "best effort" to detect time-outs as soon as possible after they occur. However, an indefinite amount of time may elapse between a time-out being detected and a thread actually executing again after that time-out. All methods that accept timeout parameters treat values less than or equal to zero to mean not to wait at all. To wait "forever", you can use a value of Long.MAX\_VALUE.

## Synchronizers

Five classes aid common special-purpose synchronization idioms.

* [Semaphore](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html) is a classic concurrency tool.
* [CountDownLatch](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html) is a very simple yet very common utility for blocking until a given number of signals, events, or conditions hold.
* A [CyclicBarrier](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html) is a resettable multiway synchronization point useful in some styles of parallel programming.
* A [Phaser](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Phaser.html) provides a more flexible form of barrier that may be used to control phased computation among multiple threads.
* An [Exchanger](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Exchanger.html) allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs.

## Concurrent Collections

Besides Queues, this package supplies Collection implementations designed for use in multithreaded contexts: [ConcurrentHashMap](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html), [ConcurrentSkipListMap](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html), [ConcurrentSkipListSet](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html), [CopyOnWriteArrayList](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html), and [CopyOnWriteArraySet](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArraySet.html). When many threads are expected to access a given collection, a ConcurrentHashMap is normally preferable to a synchronized HashMap, and a ConcurrentSkipListMap is normally preferable to a synchronized TreeMap. A CopyOnWriteArrayList is preferable to a synchronized ArrayList when the expected number of reads and traversals greatly outnumber the number of updates to a list.

The "Concurrent" prefix used with some classes in this package is a shorthand indicating several differences from similar "synchronized" classes. For example java.util.Hashtable and Collections.synchronizedMap(new HashMap()) are synchronized. But [ConcurrentHashMap](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html) is "concurrent". A concurrent collection is thread-safe, but not governed by a single exclusion lock. In the particular case of ConcurrentHashMap, it safely permits any number of concurrent reads as well as a tunable number of concurrent writes. "Synchronized" classes can be useful when you need to prevent all access to a collection via a single lock, at the expense of poorer scalability. In other cases in which multiple threads are expected to access a common collection, "concurrent" versions are normally preferable. And unsynchronized collections are preferable when either collections are unshared, or are accessible only when holding other locks.

Most concurrent Collection implementations (including most Queues) also differ from the usual java.util conventions in that their Iterators provide weakly consistent rather than fast-fail traversal. A weakly consistent iterator is thread-safe, but does not necessarily freeze the collection while iterating, so it may (or may not) reflect any updates since the iterator was created.

## Memory Consistency Properties

Chapter 17 of The Java™ Language Specification defines the *happens-before* relation on memory operations such as reads and writes of shared variables. The results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation *happens-before* the read operation. The synchronized and volatile constructs, as well as the Thread.start() and Thread.join() methods, can form *happens-before* relationships. In particular:

* Each action in a thread *happens-before* every action in that thread that comes later in the program's order.
* An unlock (synchronized block or method exit) of a monitor *happens-before* every subsequent lock (synchronized block or method entry) of that same monitor. And because the *happens-before* relation is transitive, all actions of a thread prior to unlocking *happen-before* all actions subsequent to any thread locking that monitor.
* A write to a volatile field *happens-before* every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.
* A call to start on a thread *happens-before* any action in the started thread.
* All actions in a thread *happen-before* any other thread successfully returns from a join on that thread.

The methods of all classes in java.util.concurrent and its subpackages extend these guarantees to higher-level synchronization. In particular:

* Actions in a thread prior to placing an object into any concurrent collection *happen-before* actions subsequent to the access or removal of that element from the collection in another thread.
* Actions in a thread prior to the submission of a Runnable to an Executor *happen-before* its execution begins. Similarly for Callables submitted to an ExecutorService.
* Actions taken by the asynchronous computation represented by a Future *happen-before* actions subsequent to the retrieval of the result via Future.get() in another thread.
* Actions prior to "releasing" synchronizer methods such as Lock.unlock, Semaphore.release, and CountDownLatch.countDown *happen-before* actions subsequent to a successful "acquiring" method such as Lock.lock, Semaphore.acquire, Condition.await, and CountDownLatch.await on the same synchronizer object in another thread.
* For each pair of threads that successfully exchange objects via an Exchanger, actions prior to the exchange() in each thread *happen-before* those subsequent to the corresponding exchange() in another thread.
* Actions prior to calling CyclicBarrier.await and Phaser.awaitAdvance (as well as its variants) *happen-before* actions performed by the barrier action, and actions performed by the barrier action *happen-before* actions subsequent to a successful return from the corresponding await in other threads.

# **Java I/O Tutorial**

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

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

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

## Stream

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

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

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

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

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

Let's see the code to print **output and error** message to the console.

1. System.out.println("simple message");
2. System.err.println("error message");

Let's see the code to get **input** from console.

1. **int** i=System.in.read();//returns ASCII code of 1st character
2. System.out.println((**char**)i);//will print the character

## OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

### **OutputStream**

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

### **InputStream**

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

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

Java IO

## OutputStream class

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

### **Useful methods of OutputStream**

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public void write(int)throws IOException | is used to write a byte to the current output stream. |
| 2) public void write(byte[])throws IOException | is used to write an array of byte to the current output stream. |
| 3) public void flush()throws IOException | flushes the current output stream. |
| 4) public void close()throws IOException | is used to close the current output stream. |

### **OutputStream Hierarchy**

Java output stream hierarchy

## InputStream class

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

### **Useful methods of InputStream**

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

### **InputStream Hierarchy**

Java input stream hierarchy

# **Java FileOutputStream Class**

Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutStream.

## FileOutputStream class declaration

Let's see the declaration for Java.io.FileOutputStream class:

1. **public** **class** FileOutputStream **extends** OutputStream

## FileOutputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| protected void finalize() | It is sued to clean up the connection with the file output stream. |
| void write(byte[] ary) | It is used to write **ary.length** bytes from the byte array to the file output stream. |
| void write(byte[] ary, int off, int len) | It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream. |
| void write(int b) | It is used to write the specified byte to the file output stream. |
| FileChannel getChannel() | It is used to return the file channel object associated with the file output stream. |
| FileDescriptor getFD() | It is used to return the file descriptor associated with the stream. |
| void close() | It is used to closes the file output stream. |

## Java FileOutputStream Example 1: write byte

1. **import** java.io.FileOutputStream;
2. **public** **class** FileOutputStreamExample {
3. **public** **static** **void** main(String args[]){
4. **try**{
5. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");
6. fout.write(65);
7. fout.close();
8. System.out.println("success...");
9. }**catch**(Exception e){System.out.println(e);}
10. }
11. }

Output:

Success...

The content of a text file **testout.txt** is set with the data **A**.

testout.txt

A

## Java FileOutputStream example 2: write string

1. **import** java.io.FileOutputStream;
2. **public** **class** FileOutputStreamExample {
3. **public** **static** **void** main(String args[]){
4. **try**{
5. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");
6. String s="Welcome to javaTpoint.";
7. **byte** b[]=s.getBytes();//converting string into byte array
8. fout.write(b);
9. fout.close();
10. System.out.println("success...");
11. }**catch**(Exception e){System.out.println(e);}
12. }
13. }

Output:

Success...

The content of a text file **testout.txt** is set with the data **Welcome to javaTpoint.**

testout.txt

Welcome to javaTpoint.

# **Java FileInputStream Class**

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

## Java FileInputStream class declaration

Let's see the declaration for java.io.FileInputStream class:

1. **public** **class** FileInputStream **extends** InputStream

## Java FileInputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| int available() | It is used to return the estimated number of bytes that can be read from the input stream. |
| int read() | It is used to read the byte of data from the input stream. |
| int read(byte[] b) | It is used to read up to **b.length** bytes of data from the input stream. |
| int read(byte[] b, int off, int len) | It is used to read up to **len** bytes of data from the input stream. |
| long skip(long x) | It is used to skip over and discards x bytes of data from the input stream. |
| FileChannel getChannel() | It is used to return the unique FileChannel object associated with the file input stream. |
| FileDescriptor getFD() | It is used to return the FileDescriptor object. |
| protected void finalize() | It is used to ensure that the close method is call when there is no more reference to the file input stream. |
| void close() | It is used to closes the stream. |

## Java FileInputStream example 1: read single character

1. **import** java.io.FileInputStream;
2. **public** **class** DataStreamExample {
3. **public** **static** **void** main(String args[]){
4. **try**{
5. FileInputStream fin=**new** FileInputStream("D:\\testout.txt");
6. **int** i=fin.read();
7. System.out.print((**char**)i);
9. fin.close();
10. }**catch**(Exception e){System.out.println(e);}
11. }
12. }

**Note:** Before running the code, a text file named as **"testout.txt"**is required to be created. In this file, we are having following content:

Welcome to javatpoint.

After executing the above program, you will get a single character from the file which is 87 (in byte form). To see the text, you need to convert it into character.

Output:

W

## Java FileInputStream example 2: read all characters

1. **package** com.javatpoint;
3. **import** java.io.FileInputStream;
4. **public** **class** DataStreamExample {
5. **public** **static** **void** main(String args[]){
6. **try**{
7. FileInputStream fin=**new** FileInputStream("D:\\testout.txt");
8. **int** i=0;
9. **while**((i=fin.read())!=-1){
10. System.out.print((**char**)i);
11. }
12. fin.close();
13. }**catch**(Exception e){System.out.println(e);}
14. }
15. }

Output:

Welcome to javaTpoint

# **Java BufferedOutputStream Class**

Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:

1. OutputStream os= **new** BufferedOutputStream(**new** FileOutputStream("D:\\IO Package\\testout.txt"));

## Java BufferedOutputStream class declaration

Let's see the declaration for Java.io.BufferedOutputStream class:

1. **public** **class** BufferedOutputStream **extends** FilterOutputStream

## Java BufferedOutputStream class constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| BufferedOutputStream(OutputStream os) | It creates the new buffered output stream which is used for writing the data to the specified output stream. |
| BufferedOutputStream(OutputStream os, int size) | It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size. |

## Java BufferedOutputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| void write(int b) | It writes the specified byte to the buffered output stream. |
| void write(byte[] b, int off, int len) | It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset |
| void flush() | It flushes the buffered output stream. |

## Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

1. **package** com.javatpoint;
2. **import** java.io.\*;
3. **public** **class** BufferedOutputStreamExample{
4. **public** **static** **void** main(String args[])**throws** Exception{
5. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");
6. BufferedOutputStream bout=**new** BufferedOutputStream(fout);
7. String s="Welcome to javaTpoint.";
8. **byte** b[]=s.getBytes();
9. bout.write(b);
10. bout.flush();
11. bout.close();
12. fout.close();
13. System.out.println("success");
14. }
15. }

Output:

Success

testout.txt

Welcome to javaTpoint.

# **Java BufferedInputStream Class**

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

* When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
* When a BufferedInputStream is created, an internal buffer array is created.

## Java BufferedInputStream class declaration

Let's see the declaration for Java.io.BufferedInputStream class:

1. **public** **class** BufferedInputStream **extends** FilterInputStream

## Java BufferedInputStream class constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| BufferedInputStream(InputStream IS) | It creates the BufferedInputStream and saves it argument, the input stream IS, for later use. |
| BufferedInputStream(InputStream IS, int size) | It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use. |

## Java BufferedInputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| int available() | It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream. |
| int read() | It read the next byte of data from the input stream. |
| int read(byte[] b, int off, int ln) | It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset. |
| void close() | It closes the input stream and releases any of the system resources associated with the stream. |
| void reset() | It repositions the stream at a position the mark method was last called on this input stream. |
| void mark(int readlimit) | It sees the general contract of the mark method for the input stream. |
| long skip(long x) | It skips over and discards x bytes of data from the input stream. |
| boolean markSupported() | It tests for the input stream to support the mark and reset methods. |

### **Example of Java BufferedInputStream**

Let's see the simple example to read data of file using BufferedInputStream:

1. **package** com.javatpoint;
3. **import** java.io.\*;
4. **public** **class** BufferedInputStreamExample{
5. **public** **static** **void** main(String args[]){
6. **try**{
7. FileInputStream fin=**new** FileInputStream("D:\\testout.txt");
8. BufferedInputStream bin=**new** BufferedInputStream(fin);
9. **int** i;
10. **while**((i=bin.read())!=-1){
11. System.out.print((**char**)i);
12. }
13. bin.close();
14. fin.close();
15. }**catch**(Exception e){System.out.println(e);}
16. }
17. }

Here, we are assuming that you have following data in **"testout.txt"** file:

javaTpoint

Output:

javaTpoint

# **Java SequenceInputStream Class**

Java SequenceInputStream class is used to read data from multiple streams. It reads data sequentially (one by one).

## Java SequenceInputStream Class declaration

Let's see the declaration for Java.io.SequenceInputStream class:

1. **public** **class** SequenceInputStream **extends** InputStream

## Constructors of SequenceInputStream class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| SequenceInputStream(InputStream s1, InputStream s2) | creates a new input stream by reading the data of two input stream in order, first s1 and then s2. |
| SequenceInputStream(Enumeration e) | creates a new input stream by reading the data of an enumeration whose type is InputStream. |

## Methods of SequenceInputStream class

|  |  |
| --- | --- |
| **Method** | **Description** |
| int read() | It is used to read the next byte of data from the input stream. |
| int read(byte[] ary, int off, int len) | It is used to read len bytes of data from the input stream into the array of bytes. |
| int available() | It is used to return the maximum number of byte that can be read from an input stream. |
| void close() | It is used to close the input stream. |

## Java SequenceInputStream Example

In this example, we are printing the data of two files testin.txt and testout.txt.

1. **package** com.javatpoint;
3. **import** java.io.\*;
4. **class** InputStreamExample {
5. **public** **static** **void** main(String args[])**throws** Exception{
6. FileInputStream input1=**new** FileInputStream("D:\\testin.txt");
7. FileInputStream input2=**new** FileInputStream("D:\\testout.txt");
8. SequenceInputStream inst=**new** SequenceInputStream(input1, input2);
9. **int** j;
10. **while**((j=inst.read())!=-1){
11. System.out.print((**char**)j);
12. }
13. inst.close();
14. input1.close();
15. input2.close();
16. }
17. }

Here, we are assuming that you have two files: testin.txt and testout.txt which have following information:

testin.txt:

Welcome to Java IO Programming.

testout.txt:

It is the example of Java SequenceInputStream class.

After executing the program, you will get following output:

Output:

Welcome to Java IO Programming. It is the example of Java SequenceInputStream class.

## Example that reads the data from two files and writes into another file

In this example, we are writing the data of two files **testin1.txt** and **testin2.txt** into another file named **testout.txt.**

1. **package** com.javatpoint;
3. **import** java.io.\*;
4. **class** Input1{
5. **public** **static** **void** main(String args[])**throws** Exception{
6. FileInputStream fin1=**new** FileInputStream("D:\\testin1.txt");
7. FileInputStream fin2=**new** FileInputStream("D:\\testin2.txt");
8. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");
9. SequenceInputStream sis=**new** SequenceInputStream(fin1,fin2);
10. **int** i;
11. **while**((i=sis.read())!=-1)
12. {
13. fout.write(i);
14. }
15. sis.close();
16. fout.close();
17. fin1.close();
18. fin2.close();
19. System.out.println("Success..");
20. }
21. }

Output:

Succeess...

testout.txt:

1. Welcome to Java IO Programming. It is the example of Java SequenceInputStream **class**.

## SequenceInputStream example that reads data using enumeration

If we need to read the data from more than two files, we need to use Enumeration. Enumeration object can be obtained by calling elements() method of the Vector class. Let's see the simple example where we are reading the data from 4 files: a.txt, b.txt, c.txt and d.txt.

1. **package** com.javatpoint;
2. **import** java.io.\*;
3. **import** java.util.\*;
4. **class** Input2{
5. **public** **static** **void** main(String args[])**throws** IOException{
6. //creating the FileInputStream objects for all the files
7. FileInputStream fin=**new** FileInputStream("D:\\a.txt");
8. FileInputStream fin2=**new** FileInputStream("D:\\b.txt");
9. FileInputStream fin3=**new** FileInputStream("D:\\c.txt");
10. FileInputStream fin4=**new** FileInputStream("D:\\d.txt");
11. //creating Vector object to all the stream
12. Vector v=**new** Vector();
13. v.add(fin);
14. v.add(fin2);
15. v.add(fin3);
16. v.add(fin4);
17. //creating enumeration object by calling the elements method
18. Enumeration e=v.elements();
19. //passing the enumeration object in the constructor
20. SequenceInputStream bin=**new** SequenceInputStream(e);
21. **int** i=0;
22. **while**((i=bin.read())!=-1){
23. System.out.print((**char**)i);
24. }
25. bin.close();
26. fin.close();
27. fin2.close();
28. }
29. }

The a.txt, b.txt, c.txt and d.txt have following information:

a.txt:

Welcome

b.txt:

to

c.txt:

java

d.txt:

programming

Output:

Welcometojavaprogramming

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Serialization in Java**  1. [Serialization](https://www.javatpoint.com/serialization-in-java#serialization) 2. [Serializable Interface](https://www.javatpoint.com/serialization-in-java#serializable) 3. [Example of Serialization](https://www.javatpoint.com/serialization-in-java#serializationex) 4. [Deserialization](https://www.javatpoint.com/serialization-in-java#deserialization) 5. [Example of Deserialization](https://www.javatpoint.com/serialization-in-java#deserializationex) 6. [Serialization with Inheritance](https://www.javatpoint.com/serialization-in-java#serializationinheritance) 7. [Externalizable interface](https://www.javatpoint.com/serialization-in-java#serializationextern) 8. [Serialization and static datamember](https://www.javatpoint.com/serialization-in-java#serializationstatic)   **Serialization in java** is a mechanism of writing the state of an object into a byte stream.  It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.  The reverse operation of serialization is called deserialization. **Advantage of Java Serialization** It is mainly used to travel object's state on the network (known as marshaling).  java serialization java.io.Serializable interface Serializable is a marker interface (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability. The Cloneable and Remote are also marker interfaces.  It must be implemented by the class whose object you want to persist.  The String class and all the wrapper classes implements java.io.Serializable interface by default.  Let's see the example given below:   1. **import** java.io.Serializable; 2. **public** **class** Student **implements** Serializable{ 3. **int** id; 4. String name; 5. **public** Student(**int** id, String name) { 6. **this**.id = id; 7. **this**.name = name; 8. } 9. }   In the above example, Student class implements Serializable interface. Now its objects can be converted into stream. **ObjectOutputStream class** The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams. **Constructor**  |  | | --- | | 1) public ObjectOutputStream(OutputStream out) throws IOException {}creates an ObjectOutputStream that writes to the specified OutputStream. |  **Important Methods**  |  |  | | --- | --- | | **Method** | **Description** | | 1) public final void writeObject(Object obj) throws IOException {} | writes the specified object to the ObjectOutputStream. | | 2) public void flush() throws IOException {} | flushes the current output stream. | | 3) public void close() throws IOException {} | closes the current output stream. |  Example of Java Serialization In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.   1. **import** java.io.\*; 2. **class** Persist{ 3. **public** **static** **void** main(String args[])**throws** Exception{ 4. Student s1 =**new** Student(211,"ravi"); 6. FileOutputStream fout=**new** FileOutputStream("f.txt"); 7. ObjectOutputStream out=**new** ObjectOutputStream(fout); 9. out.writeObject(s1); 10. out.flush(); 11. System.out.println("success"); 12. } 13. }   success  [download this example of serialization](https://www.javatpoint.com/src/serialization/serialization.zip) Deserialization in java Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization. **ObjectInputStream class** An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream. **Constructor**  |  |  | | --- | --- | | **1) public ObjectInputStream(InputStream in) throws IOException {}** | creates an ObjectInputStream that reads from the specified InputStream. |  **Important Methods**  |  |  | | --- | --- | | **Method** | **Description** | | 1) public final Object readObject() throws IOException, ClassNotFoundException{} | reads an object from the input stream. | | 2) public void close() throws IOException {} | closes ObjectInputStream. |  **Example of Java Deserialization**  1. **import** java.io.\*; 2. **class** Depersist{ 3. **public** **static** **void** main(String args[])**throws** Exception{ 5. ObjectInputStream in=**new** ObjectInputStream(**new** FileInputStream("f.txt")); 6. Student s=(Student)in.readObject(); 7. System.out.println(s.id+" "+s.name); 9. in.close(); 10. } 11. }   211 ravi  [download this example of deserialization](https://www.javatpoint.com/src/serialization/deserialization.zip) Java Serialization with Inheritance (IS-A Relationship) If a class implements serializable then all its sub classes will also be serializable. Let's see the example given below:   1. **import** java.io.Serializable; 2. **class** Person **implements** Serializable{ 3. **int** id; 4. String name; 5. Person(**int** id, String name) { 6. **this**.id = id; 7. **this**.name = name; 8. } 9. } 10. **class** Student **extends** Person{ 11. String course; 12. **int** fee; 13. **public** Student(**int** id, String name, String course, **int** fee) { 14. **super**(id,name); 15. **this**.course=course; 16. **this**.fee=fee; 17. } 18. }   Now you can serialize the Student class object that extends the Person class which is Serializable.Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be. Java Serialization with Aggregation (HAS-A Relationship) If a class has a reference of another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, *NotSerializableException* is thrown at runtime.   1. **class** Address{ 2. String addressLine,city,state; 3. **public** Address(String addressLine, String city, String state) { 4. **this**.addressLine=addressLine; 5. **this**.city=city; 6. **this**.state=state; 7. } 8. } 9. **import** java.io.Serializable; 10. **public** **class** Student **implements** Serializable{ 11. **int** id; 12. String name; 13. Address address;//HAS-A 14. **public** Student(**int** id, String name) { 15. **this**.id = id; 16. **this**.name = name; 17. } 18. }   Since Address is not Serializable, you can not serialize the instance of Student class. Note: All the objects within an object must be Serializable.Java Serialization with static data member If there is any static data member in a class, it will not be serialized because static is the part of class not object.   1. **class** Employee **implements** Serializable{ 2. **int** id; 3. String name; 4. **static** String company="SSS IT Pvt Ltd";//it won't be serialized 5. **public** Student(**int** id, String name) { 6. **this**.id = id; 7. **this**.name = name; 8. } 9. }  Java Serialization with array or collection Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization will be failed. Externalizable in java The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.  The Externalizable interface provides two methods:   * **public void writeExternal(ObjectOutput out) throws IOException** * **public void readExternal(ObjectInput in) throws IOException** |

## Java Transient Keyword

If you don't want to serialize any data member of a class, you can mark it as transient.

Visit next page for more details.

# **Java Transient Keyword**

**Java transient** keyword is used in serialization. If you define any data member as transient, it will not be serialized.

Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

### **Example of Java Transient Keyword**

In this example, we have created the two classes Student and PersistExample. The age data member of the Student class is declared as transient, its value will not be serialized.

If you deserialize the object, you will get the default value for transient variable.

Let's create a class with transient variable.

1. **import** java.io.Serializable;
2. **public** **class** Student **implements** Serializable{
3. **int** id;
4. String name;
5. **transient** **int** age;//Now it will not be serialized
6. **public** Student(**int** id, String name,**int** age) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.age=age;
10. }
11. }

Now write the code to serialize the object.

1. **import** java.io.\*;
2. **class** PersistExample{
3. **public** **static** **void** main(String args[])**throws** Exception{
4. Student s1 =**new** Student(211,"ravi",22);//creating object
5. //writing object into file
6. FileOutputStream f=**new** FileOutputStream("f.txt");
7. ObjectOutputStream out=**new** ObjectOutputStream(f);
8. out.writeObject(s1);
9. out.flush();
11. out.close();
12. f.close();
13. System.out.println("success");
14. }
15. }

Output:

success

Now write the code for deserialization.

1. **import** java.io.\*;
2. **class** DePersist{
3. **public** **static** **void** main(String args[])**throws** Exception{
4. ObjectInputStream in=**new** ObjectInputStream(**new** FileInputStream("f.txt"));
5. Student s=(Student)in.readObject();
6. System.out.println(s.id+" "+s.name+" "+s.age);
7. in.close();
8. }
9. }

211 ravi 0

As you can see, printing age of the student returns 0 because value of age was not serialized.

# **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

Do You Know ?

* How to perform connection-oriented Socket Programming in networking ?
* How to display the data of any online web page ?
* How to get the IP address of any host name e.g. www.google.com ?
* How to perform connection-less socket programming in networking ?

## 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.

### **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 communication.

Visit next page for java socket programming.

# **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. IP Address of Server, and
2. Port number.

## 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

Let's see a simple of java socket programming in which client sends a text and server receives 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. }

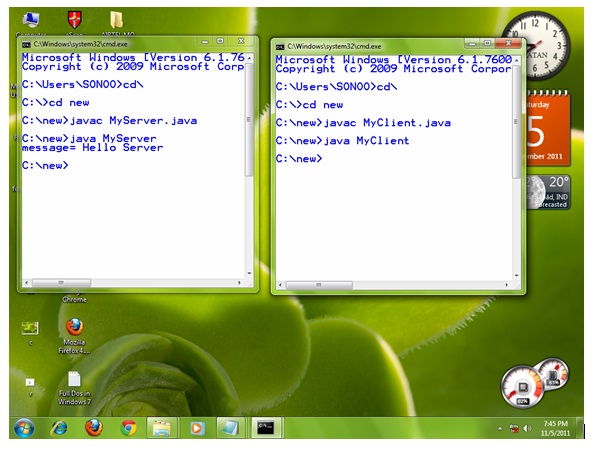
*File: MyClient.java*

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. }

[download this example](https://www.javatpoint.com/src/networking/socket.zip)

To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.



## Example of Java Socket Programming (Read-Write both side)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.

*File: MyServer.java*

1. **import** java.net.\*;
2. **import** java.io.\*;
3. **class** MyServer{
4. **public** **static** **void** main(String args[])**throws** Exception{
5. ServerSocket ss=**new** ServerSocket(3333);
6. Socket s=ss.accept();
7. DataInputStream din=**new** DataInputStream(s.getInputStream());
8. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
9. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
11. String str="",str2="";
12. **while**(!str.equals("stop")){
13. str=din.readUTF();
14. System.out.println("client says: "+str);
15. str2=br.readLine();
16. dout.writeUTF(str2);
17. dout.flush();
18. }
19. din.close();
20. s.close();
21. ss.close();
22. }}

*File: MyClient.java*

1. **import** java.net.\*;
2. **import** java.io.\*;
3. **class** MyClient{
4. **public** **static** **void** main(String args[])**throws** Exception{
5. Socket s=**new** Socket("localhost",3333);
6. DataInputStream din=**new** DataInputStream(s.getInputStream());
7. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
8. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
10. String str="",str2="";
11. **while**(!str.equals("stop")){
12. str=br.readLine();
13. dout.writeUTF(str);
14. dout.flush();
15. str2=din.readUTF();
16. System.out.println("Server says: "+str2);
17. }
19. dout.close();
20. s.close();
21. }}

# **Java URL**

The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

1. http://www.javatpoint.com/java-tutorial

A URL contains many information:

1. **Protocol:** In this case, http is the protocol.
2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.
3. **Port Number:** It is an optional attribute. If we write http//ww.javatpoint.com:80/sonoojaiswal/ , 80 is the port number. If port number is not mentioned in the URL, it returns -1.
4. **File Name or directory name:** In this case, index.jsp is the file name.

## Commonly used methods of Java URL class

The java.net.URL class provides many methods. The important methods of URL class are given below.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public String getProtocol() | it returns the protocol of the URL. |
| public String getHost() | it returns the host name of the URL. |
| public String getPort() | it returns the Port Number of the URL. |
| public String getFile() | it returns the file name of the URL. |
| public URLConnection openConnection() | it returns the instance of URLConnection i.e. associated with this URL. |

## Example of Java URL class

1. //URLDemo.java
2. **import** java.io.\*;
3. **import** java.net.\*;
4. **public** **class** URLDemo{
5. **public** **static** **void** main(String[] args){
6. **try**{
7. URL url=**new** URL("http://www.javatpoint.com/java-tutorial");
9. System.out.println("Protocol: "+url.getProtocol());
10. System.out.println("Host Name: "+url.getHost());
11. System.out.println("Port Number: "+url.getPort());
12. System.out.println("File Name: "+url.getFile());
14. }**catch**(Exception e){System.out.println(e);}
15. }
16. }

# **Java URLConnection class**

The **Java URLConnection** class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

## How to get the object of URLConnection class

The openConnection() method of URL class returns the object of URLConnection class. Syntax:

1. **public** URLConnection openConnection()**throws** IOException{}

## Displaying source code of a webpage by URLConnecton class

The URLConnection class provides many methods, we can display all the data of a webpage by using the getInputStream() method. The getInputStream() method returns all the data of the specified URL in the stream that can be read and displayed.

### **Example of Java URLConnecton class**

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public** **class** URLConnectionExample {
4. **public** **static** **void** main(String[] args){
5. **try**{
6. URL url=**new** URL("http://www.javatpoint.com/java-tutorial");
7. URLConnection urlcon=url.openConnection();
8. InputStream stream=urlcon.getInputStream();
9. **int** i;
10. **while**((i=stream.read())!=-1){
11. System.out.print((**char**)i);
12. }
13. }**catch**(Exception e){System.out.println(e);}
14. }
15. }

# **Java HttpURLConnection class**

The **Java HttpURLConnection** class is http specific URLConnection. It works for HTTP protocol only.

By the help of HttpURLConnection class, you can information of any HTTP URL such as header information, status code, response code etc.

The java.net.HttpURLConnection is subclass of URLConnection class.

## How to get the object of HttpURLConnection class

The openConnection() method of URL class returns the object of URLConnection class. Syntax:

1. **public** URLConnection openConnection()**throws** IOException{}

You can typecast it to HttpURLConnection type as given below.

1. URL url=**new** URL("http://www.javatpoint.com/java-tutorial");
2. HttpURLConnection huc=(HttpURLConnection)url.openConnection();

## Java HttpURLConnecton Example

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public** **class** HttpURLConnectionDemo{
4. **public** **static** **void** main(String[] args){
5. **try**{
6. URL url=**new** URL("http://www.javatpoint.com/java-tutorial");
7. HttpURLConnection huc=(HttpURLConnection)url.openConnection();
8. **for**(**int** i=1;i<=8;i++){
9. System.out.println(huc.getHeaderFieldKey(i)+" = "+huc.getHeaderField(i));
10. }
11. huc.disconnect();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

# **Java 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.javatpoint.com, www.google.com, www.facebook.com etc.

## Commonly used methods of InetAddress class

|  |  |
| --- | --- |
| **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.javatpoint.com website.

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.javatpoint.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. }

# **Java DatagramSocket and DatagramPacket**

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

## Java DatagramSocket class

**Java DatagramSocket** class represents a connection-less socket for sending and receiving datagram packets.

A datagram is basically an information but there is no guarantee of its content, arrival or arrival time.

## Commonly used Constructors of DatagramSocket class

* **DatagramSocket() throws SocketEeption:**it creates a datagram socket and binds it with the available Port Number on the localhost machine.
* **DatagramSocket(int port) throws SocketEeption:**it creates a datagram socket and binds it with the given Port Number.
* **DatagramSocket(int port, InetAddress address) throws SocketEeption:**it creates a datagram socket and binds it with the specified port number and host address.

## Java DatagramPacket class

**Java DatagramPacket** is a message that can be sent or received. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.

## Commonly used Constructors of DatagramPacket class

* **DatagramPacket(byte[] barr, int length):**it creates a datagram packet. This constructor is used to receive the packets.
* **DatagramPacket(byte[] barr, int length, InetAddress address, int port):**it creates a datagram packet. This constructor is used to send the packets.

### **Example of Sending DatagramPacket by DatagramSocket**

1. //DSender.java
2. **import** java.net.\*;
3. **public** **class** DSender{
4. **public** **static** **void** main(String[] args) **throws** Exception {
5. DatagramSocket ds = **new** DatagramSocket();
6. String str = "Welcome java";
7. InetAddress ip = InetAddress.getByName("127.0.0.1");
9. DatagramPacket dp = **new** DatagramPacket(str.getBytes(), str.length(), ip, 3000);
10. ds.send(dp);
11. ds.close();
12. }
13. }

### **Example of Receiving DatagramPacket by DatagramSocket**

1. //DReceiver.java
2. **import** java.net.\*;
3. **public** **class** DReceiver{
4. **public** **static** **void** main(String[] args) **throws** Exception {
5. DatagramSocket ds = **new** DatagramSocket(3000);
6. **byte**[] buf = **new** **byte**[1024];
7. DatagramPacket dp = **new** DatagramPacket(buf, 1024);
8. ds.receive(dp);
9. String str = **new** String(dp.getData(), 0, dp.getLength());
10. System.out.println(str);
11. ds.close();
12. }
13. }

# **Collections in Java**

1. [Java Collection Framework](https://www.javatpoint.com/collections-in-java)
2. [Hierarchy of Collection Framework](https://www.javatpoint.com/collections-in-java#collectionhierarchy)
3. [Collection interface](https://www.javatpoint.com/collections-in-java#collectionmethods)
4. [Iterator interface](https://www.javatpoint.com/collections-in-java#collectioniterator)

**Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

#### **What is Collection in java**

Collection represents a single unit of objects i.e. a group.

#### **What is framework in java**

* provides readymade architecture.
* represents set of classes and interface.
* is optional.

#### **What is Collection framework**

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

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

Do You Know ?

* What are the two ways to iterate the elements of a collection ?
* What is the difference between ArrayList and LinkedList classes in collection framework?
* What is the difference between ArrayList and Vector classes in collection framework?
* What is the difference between HashSet and HashMap classes in collection framework?
* What is the difference between HashMap and Hashtable class?
* What is the difference between Iterator and Enumeration interface in collection framework?
* How can we sort the elements of an object. What is the difference between Comparable and Comparator interfaces?
* What does the hashcode() method ?
* What is the difference between java collection and java collections ?

### **Hierarchy of Collection Framework**

Let us see the hierarchy of collection framework.The **java.util** package contains all the classes and interfaces for Collection framework.

hierarchy of collection framework

### **Methods of Collection interface**

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

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(Object element) | is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection c) | is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | is used to delete an element from this collection. |
| 4 | public boolean removeAll(Collection c) | is used to delete all the elements of specified collection from the invoking collection. |
| 5 | public boolean retainAll(Collection c) | is used to delete all the elements of invoking collection except the specified collection. |
| 6 | public int size() | return the total number of elements in the collection. |
| 7 | public void clear() | removes the total no of element from the collection. |
| 8 | public boolean contains(Object element) | is used to search an element. |
| 9 | public boolean containsAll(Collection c) | is used to search the specified collection in this collection. |
| 10 | public Iterator iterator() | returns an iterator. |
| 11 | public Object[] toArray() | converts collection into array. |
| 12 | public boolean isEmpty() | checks if collection is empty. |
| 13 | public boolean equals(Object element) | matches two collection. |
| 14 | public int hashCode() | returns the hashcode number for collection. |

### **Iterator interface**

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

#### **Methods of Iterator interface**

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

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

What we are going to learn in Java Collections Framework

1. [ArrayList class](https://www.javatpoint.com/java-arraylist)
2. [LinkedList class](https://www.javatpoint.com/java-linkedlist)
3. [List interface](https://www.javatpoint.com/java-list)
4. [HashSet class](https://www.javatpoint.com/java-hashset)
5. [LinkedHashSet class](https://www.javatpoint.com/java-linkedhashset)
6. [TreeSet class](https://www.javatpoint.com/java-treeset)
7. [PriorityQueue class](https://www.javatpoint.com/java-priorityqueue)
8. [Map interface](https://www.javatpoint.com/java-map)
9. [HashMap class](https://www.javatpoint.com/java-hashmap)
10. [LinkedHashMap class](https://www.javatpoint.com/java-linkedhashmap)
11. [TreeMap class](https://www.javatpoint.com/TreeMap-class-in-collection-framework)
12. [Hashtable class](https://www.javatpoint.com/Hashtable-class-in-collection-framework)
13. [Sorting](https://www.javatpoint.com/Sorting-in-collection-framework)
14. [Comparable interface](https://www.javatpoint.com/Comparable-interface-in-collection-framework)
15. [Comparator interface](https://www.javatpoint.com/Comparator-interface-in-collection-framework)
16. [Properties class in Java](https://www.javatpoint.com/properties-class-in-java)

# **Java ArrayList class**

Java ArrayList class hierarchy

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

### **Hierarchy of ArrayList class**

As shown in above diagram, Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.

### **ArrayList class declaration**

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

1. **public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

### **Constructors of Java ArrayList**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| ArrayList() | It is used to build an empty array list. |
| ArrayList(Collection c) | It is used to build an array list that is initialized with the elements of the collection c. |
| ArrayList(int capacity) | It is used to build an array list that has the specified initial capacity. |

### **Methods of Java ArrayList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| void clear() | It is used to remove all of the elements from this list. |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| Object[] toArray(Object[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean addAll(int index, Collection c) | It is used to insert all of the elements in the specified collection into this list, starting at the specified position. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |

### **Java Non-generic Vs Generic Collection**

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

1. ArrayList al=**new** ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

1. ArrayList<String> al=**new** ArrayList<String>();//creating new generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.

For more information of java generics, click here [Java Generics Tutorial](https://www.javatpoint.com/generics-in-java).

### **Java ArrayList Example**

1. **import** java.util.\*;
2. **class** TestCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection1)

Ravi

Vijay

Ravi

Ajay

### **Two ways to iterate the elements of collection in java**

There are two ways to traverse collection elements:

1. By Iterator interface.
2. By for-each loop.

In the above example, we have seen traversing ArrayList by Iterator. Let's see the example to traverse ArrayList elements using for-each loop.

### **Iterating Collection through for-each loop**

1. **import** java.util.\*;
2. **class** TestCollection2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. **for**(String obj:al)
10. System.out.println(obj);
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection2)

Ravi

Vijay

Ravi

Ajay

### **User-defined class objects in Java ArrayList**

Let's see an example where we are storing Student class object in array list.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }
11. **import** java.util.\*;
12. **public** **class** TestCollection3{
13. **public** **static** **void** main(String args[]){
14. //Creating user-defined class objects
15. Student s1=**new** Student(101,"Sonoo",23);
16. Student s2=**new** Student(102,"Ravi",21);
17. Student s2=**new** Student(103,"Hanumat",25);
18. //creating arraylist
19. ArrayList<Student> al=**new** ArrayList<Student>();
20. al.add(s1);//adding Student class object
21. al.add(s2);
22. al.add(s3);
23. //Getting Iterator
24. Iterator itr=al.iterator();
25. //traversing elements of ArrayList object
26. **while**(itr.hasNext()){
27. Student st=(Student)itr.next();
28. System.out.println(st.rollno+" "+st.name+" "+st.age);
29. }
30. }
31. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection3)

101 Sonoo 23

102 Ravi 21

103 Hanumat 25

### **Example of addAll(Collection c) method**

1. **import** java.util.\*;
2. **class** TestCollection4{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Sonoo");
10. al2.add("Hanumat");
11. al.addAll(al2);//adding second list in first list
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection4)

Ravi

Vijay

Ajay

Sonoo

Hanumat

### **Example of removeAll() method**

1. **import** java.util.\*;
2. **class** TestCollection5{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Ravi");
10. al2.add("Hanumat");
11. al.removeAll(al2);
12. System.out.println("iterating the elements after removing the elements of al2...");
13. Iterator itr=al.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection5)

iterating the elements after removing the elements of al2...

Vijay

Ajay

### **Example of retainAll() method**

1. **import** java.util.\*;
2. **class** TestCollection6{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Ravi");
10. al2.add("Hanumat");
11. al.retainAll(al2);
12. System.out.println("iterating the elements after retaining the elements of al2...");
13. Iterator itr=al.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection6)

iterating the elements after retaining the elements of al2...

Ravi

### **Java ArrayList Example: Book**

Let's see an ArrayList example where we are adding books to list and printing all the books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

# **Java LinkedList class**

Java LinkedList class hierarchy

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

The important points about Java LinkedList are:

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

### **Hierarchy of LinkedList class**

As shown in above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

### **Doubly Linked List**

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

java LinkedList class using doubly linked list

### **LinkedList class declaration**

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

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

### **Constructors of Java LinkedList**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedList() | It is used to construct an empty list. |
| LinkedList(Collection c) | It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |

### **Methods of Java LinkedList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| void addFirst(Object o) | It is used to insert the given element at the beginning of a list. |
| void addLast(Object o) | It is used to append the given element to the end of a list. |
| int size() | It is used to return the number of elements in a list |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean contains(Object o) | It is used to return true if the list contains a specified element. |
| boolean remove(Object o) | It is used to remove the first occurence of the specified element in a list. |
| Object getFirst() | It is used to return the first element in a list. |
| Object getLast() | It is used to return the last element in a list. |
| int indexOf(Object o) | It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element. |
| int lastIndexOf(Object o) | It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element. |

### **Java LinkedList Example**

1. **import** java.util.\*;
2. **public** **class** TestCollection7{
3. **public** **static** **void** main(String args[]){
5. LinkedList<String> al=**new** LinkedList<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection7)

Output:Ravi

Vijay

Ravi

Ajay

### **Java LinkedList Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** LinkedListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** LinkedList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

# **Difference between ArrayList and LinkedList**

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

But there are many differences between ArrayList and LinkedList classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

### **Example of ArrayList and LinkedList in Java**

Let's see a simple example where we are using ArrayList and LinkedList both.

1. **import** java.util.\*;
2. **class** TestArrayLinked{
3. **public** **static** **void** main(String args[]){
5. List<String> al=**new** ArrayList<String>();//creating arraylist
6. al.add("Ravi");//adding object in arraylist
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. List<String> al2=**new** LinkedList<String>();//creating linkedlist

# al2.add(**Java List Interface**

List Interface is the subinterface of Collection.It contains methods to insert and delete elements in index basis.It is a factory of ListIterator interface.

### **List Interface declaration**

1. **public** **interface** List<E> **extends** Collection<E>

### **Methods of Java List Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index,Object element) | It is used to insert element into the invoking list at the index passed in the index. |
| boolean addAll(int index,Collection c) | It is used to insert all elements of c into the invoking list at the index passed in the index. |
| object get(int index) | It is used to return the object stored at the specified index within the invoking collection. |
| object set(int index,Object element) | It is used to assign element to the location specified by index within the invoking list. |
| object remove(int index) | It is used to remove the element at position index from the invoking list and return the deleted element. |
| ListIterator listIterator() | It is used to return an iterator to the start of the invoking list. |
| ListIterator listIterator(int index) | It is used to return an iterator to the invoking list that begins at the specified index. |

### **Java List Example**

1. **import** java.util.\*;
2. **public** **class** ListExample{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. System.out.println("Element at 2nd position: "+al.get(2));
10. **for**(String s:al){
11. System.out.println(s);
12. }
13. }
14. }

Output:

Element at 2nd position: Vijay

Amit

Sachin

Vijay

Kumar

## Java ListIterator Interface

ListIterator Interface is used to traverse the element in backward and forward direction.

### **ListIterator Interface declaration**

1. **public** **interface** ListIterator<E> **extends** Iterator<E>

### **Methods of Java ListIterator Interface:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean hasNext() | This method return true if the list iterator has more elements when traversing the list in the forward direction. |
| Object next() | This method return the next element in the list and advances the cursor position. |
| boolean hasPrevious() | This method return true if this list iterator has more elements when traversing the list in the reverse direction. |
| Object previous() | This method return the previous element in the list and moves the cursor position backwards. |

### **Example of ListIterator Interface**

1. **import** java.util.\*;
2. **public** **class** TestCollection8{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. System.out.println("element at 2nd position: "+al.get(2));
10. ListIterator<String> itr=al.listIterator();
11. System.out.println("traversing elements in forward direction...");
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. System.out.println("traversing elements in backward direction...");
16. **while**(itr.hasPrevious()){
17. System.out.println(itr.previous());
18. }
19. }
20. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection8)

Output:

element at 2nd position: Vijay

traversing elements in forward direction...

Amit

Sachin

Vijay

Kumar

traversing elements in backward direction...

Kumar

Vijay

Sachin

Amit

### **Example of ListIterator Interface: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }
32. "James");//adding object in linkedlist
33. al2.add("Serena");
34. al2.add("Swati");
35. al2.add("Junaid");
37. System.out.println("arraylist: "+al);
38. System.out.println("linkedlist: "+al2);
39. }
40. }

# **Java HashSet class**

Java HashSet class hierarchy

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.

## Difference between List and Set

List can contain duplicate elements whereas Set contains unique elements only.

### **Hierarchy of HashSet class**

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

### **HashSet class declaration**

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

1. **public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

### **Constructors of Java HashSet class:**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| HashSet(int capacity) | It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet. |

### **Methods of Java HashSet class:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the elements from this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean add(Object o) | It is used to adds the specified element to this set if it is not already present. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| Object clone() | It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| Iterator iterator() | It is used to return an iterator over the elements in this set. |
| int size() | It is used to return the number of elements in this set. |

### **Java HashSet Example**

1. **import** java.util.\*;
2. **class** TestCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection9)

Ajay

Vijay

Ravi

### **Java HashSet Example: Book**

Let's see a HashSet example where we are adding books to set and printing all the books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** HashSetExample {
15. **public** **static** **void** main(String[] args) {
16. HashSet<Book> set=**new** HashSet<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to HashSet
22. set.add(b1);
23. set.add(b2);
24. set.add(b3);
25. //Traversing HashSet
26. **for**(Book b:set){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

# **Java LinkedHashSet class**

Java HashSet class hierarchy

Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Contains unique elements only like HashSet.
* Provides all optional set operations, and permits null elements.
* Maintains insertion order.

## Hierarchy of LinkedHashSet class

The LinkedHashSet class extends HashSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

### **LinkedHashSet class declaration**

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

1. **public** **class** LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>, Cloneable, Serializable

### **Constructors of Java LinkedHashSet class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| LinkedHashSet(int capacity) | It is used initialize the capacity of the linkedhashset to the given integer value capacity. |
| LinkedHashSet(int capacity, float fillRatio) | It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument. |

### **Example of LinkedHashSet class:**

1. **import** java.util.\*;
2. **class** TestCollection10{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> al=**new** LinkedHashSet<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection10)

Ravi

Vijay

Ajay

### **Java LinkedHashSet Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** LinkedHashSetExample {
15. **public** **static** **void** main(String[] args) {
16. LinkedHashSet<Book> hs=**new** LinkedHashSet<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to hash table
22. hs.add(b1);
23. hs.add(b2);
24. hs.add(b3);
25. //Traversing hash table
26. **for**(Book b:hs){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

# **Java TreeSet class**

TreeSet class hierarchy

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order.

### **Hierarchy of TreeSet class**

As shown in above diagram, Java TreeSet class implements NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

### **TreeSet class declaration**

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

1. **public** **class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializable

### **Constructors of Java TreeSet class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeSet() | It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set. |
| TreeSet(Collection c) | It is used to build a new tree set that contains the elements of the collection c. |
| TreeSet(Comparator comp) | It is used to construct an empty tree set that will be sorted according to given comparator. |
| TreeSet(SortedSet ss) | It is used to build a TreeSet that contains the elements of the given SortedSet. |

### **Methods of Java TreeSet class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean addAll(Collection c) | It is used to add all of the elements in the specified collection to this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void add(Object o) | It is used to add the specified element to this set if it is not already present. |
| void clear() | It is used to remove all of the elements from this set. |
| Object clone() | It is used to return a shallow copy of this TreeSet instance. |
| Object first() | It is used to return the first (lowest) element currently in this sorted set. |
| Object last() | It is used to return the last (highest) element currently in this sorted set. |
| int size() | It is used to return the number of elements in this set. |

### **Java TreeSet Example**

1. **import** java.util.\*;
2. **class** TestCollection11{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> al=**new** TreeSet<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection11)

Output:

Ajay

Ravi

Vijay

### **Java TreeSet Example: Book**

Let's see a TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement Comparable interface.

1. **import** java.util.\*;
2. **class** Book **implements** Comparable<Book>{
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. **public** **int** compareTo(Book b) {
14. **if**(id>b.id){
15. **return** 1;
16. }**else** **if**(id<b.id){
17. **return** -1;
18. }**else**{
19. **return** 0;
20. }
21. }
22. }
23. **public** **class** TreeSetExample {
24. **public** **static** **void** main(String[] args) {
25. Set<Book> set=**new** TreeSet<Book>();
26. //Creating Books
27. Book b1=**new** Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=**new** Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=**new** Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30. //Adding Books to TreeSet
31. set.add(b1);
32. set.add(b2);
33. set.add(b3);
34. //Traversing TreeSet
35. **for**(Book b:set){
36. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
37. }
38. }
39. }

# **Java Queue Interface**

Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

### **Queue Interface declaration**

1. **public** **interface** Queue<E> **extends** Collection<E>

### **Methods of Java Queue Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this queue and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this queue. |
| Object remove() | It is used to retrieves and removes the head of this queue. |
| Object poll() | It is used to retrieves and removes the head of this queue, or returns null if this queue is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this queue. |
| Object peek() | It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

## PriorityQueue class

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

### **PriorityQueue class declaration**

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

1. **public** **class** PriorityQueue<E> **extends** AbstractQueue<E> **implements** Serializable

### **Java PriorityQueue Example**

1. **import** java.util.\*;
2. **class** TestCollection12{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit");
6. queue.add("Vijay");
7. queue.add("Karan");
8. queue.add("Jai");
9. queue.add("Rahul");
10. System.out.println("head:"+queue.element());
11. System.out.println("head:"+queue.peek());
12. System.out.println("iterating the queue elements:");
13. Iterator itr=queue.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. queue.remove();
18. queue.poll();
19. System.out.println("after removing two elements:");
20. Iterator<String> itr2=queue.iterator();
21. **while**(itr2.hasNext()){
22. System.out.println(itr2.next());
23. }
24. }
25. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection12)

Output:head:Amit

head:Amit

iterating the queue elements:

Amit

Jai

Karan

Vijay

Rahul

after removing two elements:

Karan

Rahul

Vijay

### **Java PriorityQueue Example: Book**

Let's see a PriorityQueue example where we are adding books to queue and printing all the books. The elements in PriorityQueue must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in PriorityQueue, you need to implement Comparable interface.

1. **import** java.util.\*;
2. **class** Book **implements** Comparable<Book>{
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. **public** **int** compareTo(Book b) {
14. **if**(id>b.id){
15. **return** 1;
16. }**else** **if**(id<b.id){
17. **return** -1;
18. }**else**{
19. **return** 0;
20. }
21. }
22. }
23. **public** **class** LinkedListExample {
24. **public** **static** **void** main(String[] args) {
25. Queue<Book> queue=**new** PriorityQueue<Book>();
26. //Creating Books
27. Book b1=**new** Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=**new** Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=**new** Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30. //Adding Books to the queue
31. queue.add(b1);
32. queue.add(b2);
33. queue.add(b3);
34. System.out.println("Traversing the queue elements:");
35. //Traversing queue elements
36. **for**(Book b:queue){
37. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
38. }
39. queue.remove();
40. System.out.println("After removing one book record:");
41. **for**(Book b:queue){
42. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
43. }
44. }
45. }

# **Java Deque Interface**

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

## Deque Interface declaration

1. **public** **interface** Deque<E> **extends** Queue<E>

### **Methods of Java Deque Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this deque and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this deque. |
| Object remove() | It is used to retrieves and removes the head of this deque. |
| Object poll() | It is used to retrieves and removes the head of this deque, or returns null if this deque is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this deque. |
| Object peek() | It is used to retrieves, but does not remove, the head of this deque, or returns null if this deque is empty. |

java arraydeque hierarchy

## ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

* Unlike Queue, we can add or remove elements from both sides.
* Null elements are not allowed in the ArrayDeque.
* ArrayDeque is not thread safe, in the absence of external synchronization.
* ArrayDeque has no capacity restrictions.
* ArrayDeque is faster than LinkedList and Stack.

### **ArrayDeque Hierarchy**

The hierarchy of ArrayDeque class is given in the figure displayed at the right side of the page.

### **ArrayDeque class declaration**

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

1. **public** **class** ArrayDeque<E> **extends** AbstractCollection<E> **implements** Deque<E>, Cloneable, Serializable

## Java ArrayDeque Example

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

Output:

Ravi

Vijay

Ajay

## Java ArrayDeque Example: offerFirst() and pollLast()

1. **import** java.util.\*;
2. **public** **class** DequeExample {
3. **public** **static** **void** main(String[] args) {
4. Deque<String> deque=**new** ArrayDeque<String>();
5. deque.offer("arvind");
6. deque.offer("vimal");
7. deque.add("mukul");
8. deque.offerFirst("jai");
9. System.out.println("After offerFirst Traversal...");
10. **for**(String s:deque){
11. System.out.println(s);
12. }
13. //deque.poll();
14. //deque.pollFirst();//it is same as poll()
15. deque.pollLast();
16. System.out.println("After pollLast() Traversal...");
17. **for**(String s:deque){
18. System.out.println(s);
19. }
20. }
21. }

Output:

After offerFirst Traversal...

jai

arvind

vimal

mukul

After pollLast() Traversal...

jai

arvind

vimal

## Java ArrayDeque Example: Book

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayDequeExample {
15. **public** **static** **void** main(String[] args) {
16. Deque<Book> set=**new** ArrayDeque<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to Deque
22. set.add(b1);
23. set.add(b2);
24. set.add(b3);
25. //Traversing ArrayDeque
26. **for**(Book b:set){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

# **Java Map Interface**

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

Map is useful if you have to search, update or delete elements on the basis of key.

### **Useful methods of Map interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object put(Object key, Object value) | It is used to insert an entry in this map. |
| void putAll(Map map) | It is used to insert the specified map in this map. |
| Object remove(Object key) | It is used to delete an entry for the specified key. |
| Object get(Object key) | It is used to return the value for the specified key. |
| boolean containsKey(Object key) | It is used to search the specified key from this map. |
| Set keySet() | It is used to return the Set view containing all the keys. |
| Set entrySet() | It is used to return the Set view containing all the keys and values. |

## Map.Entry Interface

Entry is the sub interface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

### **Methods of Map.Entry interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object getKey() | It is used to obtain key. |
| Object getValue() | It is used to obtain value. |

### **Java Map Example: Generic (New Style)**

1. **import** java.util.\*;
2. **class** MapInterfaceExample{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. **for**(Map.Entry m:map.entrySet()){
9. System.out.println(m.getKey()+" "+m.getValue());
10. }
11. }
12. }

Output:

102 Rahul

100 Amit

101 Vijay

### **Java Map Example: Non-Generic (Old Style)**

1. //Non-generic
2. **import** java.util.\*;
3. **public** **class** MapExample1 {
4. **public** **static** **void** main(String[] args) {
5. Map map=**new** HashMap();
6. //Adding elements to map
7. map.put(1,"Amit");
8. map.put(5,"Rahul");
9. map.put(2,"Jai");
10. map.put(6,"Amit");
11. //Traversing Map
12. Set set=map.entrySet();//Converting to Set so that we can traverse
13. Iterator itr=set.iterator();
14. **while**(itr.hasNext()){
15. //Converting to Map.Entry so that we can get key and value separately
16. Map.Entry entry=(Map.Entry)itr.next();
17. System.out.println(entry.getKey()+" "+entry.getValue());
18. }
19. }
20. }

## Class HashMap<K,V>

* [java.lang.Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)
  + [java.util.AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<K,V>
    - java.util.HashMap<K,V>
* **Type Parameters:**

K - the type of keys maintained by this map

V - the type of mapped values

**All Implemented Interfaces:**

[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<K,V>

**Direct Known Subclasses:**

[LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html), [PrinterStateReasons](https://docs.oracle.com/javase/8/docs/api/javax/print/attribute/standard/PrinterStateReasons.html)

public class **HashMap<K,V>**

extends [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<K,V>

implements [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<K,V>, [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html), this class may use comparison order among keys to help break ties.

**Note that this implementation is not synchronized.** If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the [Collections.synchronizedMap](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedMap-java.util.Map-) method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

Map m = Collections.synchronizedMap(new HashMap(...));

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html). Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the [Java Collections Framework](https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html).

**Since:**

1.2

**See Also:**

[Object.hashCode()](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--), [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html), [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html), [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html), [Hashtable](https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html), [Serialized Form](https://docs.oracle.com/javase/8/docs/api/serialized-form.html#java.util.HashMap)

### *Nested Class Summary*

### Nested classes/interfaces inherited from class java.util.[AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)

[AbstractMap.SimpleEntry](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleEntry.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleEntry.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleEntry.html)>, [AbstractMap.SimpleImmutableEntry](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleImmutableEntry.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleImmutableEntry.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleImmutableEntry.html)>

### Nested classes/interfaces inherited from interface java.util.[Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)

[Map.Entry](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html)>

### *Constructor Summary*

|  |
| --- |
| **Constructors** |
| **Constructor and Description** |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap--)()  Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75). |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-int-)(int initialCapacity)  Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75). |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-int-float-)(int initialCapacity, float loadFactor)  Constructs an empty HashMap with the specified initial capacity and load factor. |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-java.util.Map-)([**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<? extends [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> m)  Constructs a new HashMap with the same mappings as the specified Map. |

### *Method Summary*

|  |  |
| --- | --- |
| **All Methods**[**Instance Methods**](javascript:show(2);)[**Concrete Methods**](javascript:show(8);) | |
| **Modifier and Type** | **Method and Description** |
| void | [**clear**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#clear--)()  Removes all of the mappings from this map. |
| [**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) | [**clone**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#clone--)()  Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**compute**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#compute-K-java.util.function.BiFunction-)([**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [**BiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<? super [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> remappingFunction)  Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**computeIfAbsent**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#computeIfAbsent-K-java.util.function.Function-)([**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> mappingFunction)  If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**computeIfPresent**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#computeIfPresent-K-java.util.function.BiFunction-)([**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [**BiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<? super [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> remappingFunction)  If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value. |
| boolean | [**containsKey**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#containsKey-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key)  Returns true if this map contains a mapping for the specified key. |
| boolean | [**containsValue**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#containsValue-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) value)  Returns true if this map maps one or more keys to the specified value. |
| [**Set**](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)<[**Map.Entry**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html)<[**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>> | [**entrySet**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#entrySet--)()  Returns a [**Set**](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) view of the mappings contained in this map. |
| void | [**forEach**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#forEach-java.util.function.BiConsumer-)([**BiConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html)<? super [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> action)  Performs the given action for each entry in this map until all entries have been processed or the action throws an exception. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**get**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#get-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key)  Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**getOrDefault**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#getOrDefault-java.lang.Object-V-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key, [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) defaultValue)  Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key. |
| boolean | [**isEmpty**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#isEmpty--)()  Returns true if this map contains no key-value mappings. |
| [**Set**](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)<[**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> | [**keySet**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#keySet--)()  Returns a [**Set**](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) view of the keys contained in this map. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**merge**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#merge-K-V-java.util.function.BiFunction-)([**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value, [**BiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<? super [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> remappingFunction)  If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**put**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#put-K-V-)([**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value)  Associates the specified value with the specified key in this map. |
| void | [**putAll**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#putAll-java.util.Map-)([**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<? extends [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> m)  Copies all of the mappings from the specified map to this map. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**putIfAbsent**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#putIfAbsent-K-V-)([**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value)  If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returnsnull, else returns the current value. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**remove**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#remove-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key)  Removes the mapping for the specified key from this map if present. |
| boolean | [**remove**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#remove-java.lang.Object-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key, [**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) value)  Removes the entry for the specified key only if it is currently mapped to the specified value. |
| [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) | [**replace**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#replace-K-V-)([**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value)  Replaces the entry for the specified key only if it is currently mapped to some value. |
| boolean | [**replace**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#replace-K-V-V-)([**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) oldValue, [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) newValue)  Replaces the entry for the specified key only if currently mapped to the specified value. |
| void | [**replaceAll**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#replaceAll-java.util.function.BiFunction-)([**BiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<? super [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> function)  Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| int | [**size**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#size--)()  Returns the number of key-value mappings in this map. |
| [**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<[**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> | [**values**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#values--)()  Returns a [**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) view of the values contained in this map. |

### Methods inherited from class java.util.[AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)

[equals](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#equals-java.lang.Object-), [hashCode](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#hashCode--), [toString](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#toString--)

### Methods inherited from class java.lang.[Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

[finalize](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#finalize--), [getClass](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass--), [notify](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notify--), [notifyAll](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notifyAll--), [wait](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait--), [wait](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait-long-), [wait](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait-long-int-)

### Methods inherited from interface java.util.[Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)

[equals](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#equals-java.lang.Object-), [hashCode](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#hashCode--)

### *Constructor Detail*

#### HashMap

* + - public HashMap(int initialCapacity,

float loadFactor)

Constructs an empty HashMap with the specified initial capacity and load factor.

**Parameters:**

initialCapacity - the initial capacity

loadFactor - the load factor

**Throws:**

[IllegalArgumentException](https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalArgumentException.html) - if the initial capacity is negative or the load factor is nonpositive

#### HashMap

public HashMap(int initialCapacity)

Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

**Parameters:**

initialCapacity - the initial capacity.

**Throws:**

[IllegalArgumentException](https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalArgumentException.html) - if the initial capacity is negative.

#### HashMap

public HashMap()

Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

#### HashMap

public HashMap([Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<? extends [K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> m)

Constructs a new HashMap with the same mappings as the specified Map. The HashMap is created with default load factor (0.75) and an initial capacity sufficient to hold the mappings in the specified Map.

**Parameters:**

m - the map whose mappings are to be placed in this map

**Throws:**

[NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) - if the specified map is null

### *Method Detail*

#### size

public int size()

Returns the number of key-value mappings in this map.

**Specified by:**

[size](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#size--) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[size](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#size--) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Returns:**

the number of key-value mappings in this map

#### isEmpty

public boolean isEmpty()

Returns true if this map contains no key-value mappings.

**Specified by:**

[isEmpty](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#isEmpty--) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[isEmpty](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#isEmpty--) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Returns:**

true if this map contains no key-value mappings

#### get

public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) get([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key)

Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

More formally, if this map contains a mapping from a key k to a value v such that (key==null ? k==null : key.equals(k)), then this method returns v; otherwise it returns null. (There can be at most one such mapping.)

A return value of null does not *necessarily* indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to null. The [containsKey](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#containsKey-java.lang.Object-) operation may be used to distinguish these two cases.

**Specified by:**

[get](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#get-java.lang.Object-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[get](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#get-java.lang.Object-) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - the key whose associated value is to be returned

**Returns:**

the value to which the specified key is mapped, or null if this map contains no mapping for the key

**See Also:**

[put(Object, Object)](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#put-K-V-)

#### containsKey

public boolean containsKey([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key)

Returns true if this map contains a mapping for the specified key.

**Specified by:**

[containsKey](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#containsKey-java.lang.Object-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[containsKey](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#containsKey-java.lang.Object-) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - The key whose presence in this map is to be tested

**Returns:**

true if this map contains a mapping for the specified key.

#### put

* + - public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) put([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key,

[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value)

Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.

**Specified by:**

[put](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#put-K-V-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[put](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#put-K-V-) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the specified value is to be associated

value - value to be associated with the specified key

**Returns:**

the previous value associated with key, or null if there was no mapping for key. (A null return can also indicate that the map previously associated null with key.)

#### putAll

public void putAll([Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<? extends [K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> m)

Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.

**Specified by:**

[putAll](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#putAll-java.util.Map-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[putAll](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#putAll-java.util.Map-) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

m - mappings to be stored in this map

**Throws:**

[NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) - if the specified map is null

#### remove

public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) remove([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key)

Removes the mapping for the specified key from this map if present.

**Specified by:**

[remove](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#remove-java.lang.Object-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[remove](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#remove-java.lang.Object-) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key whose mapping is to be removed from the map

**Returns:**

the previous value associated with key, or null if there was no mapping for key. (A null return can also indicate that the map previously associated null with key.)

#### clear

public void clear()

Removes all of the mappings from this map. The map will be empty after this call returns.

**Specified by:**

[clear](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#clear--) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[clear](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#clear--) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

#### containsValue

public boolean containsValue([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) value)

Returns true if this map maps one or more keys to the specified value.

**Specified by:**

[containsValue](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#containsValue-java.lang.Object-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[containsValue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#containsValue-java.lang.Object-) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

value - value whose presence in this map is to be tested

**Returns:**

true if this map maps one or more keys to the specified value

#### keySet

public [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> keySet()

Returns a [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAlloperations.

**Specified by:**

[keySet](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#keySet--) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[keySet](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#keySet--) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Returns:**

a set view of the keys contained in this map

#### values

public [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> values()

Returns a [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa. If the map is modified while an iteration over the collection is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Collection.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

**Specified by:**

[values](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#values--) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Overrides:**

[values](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#values--) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Returns:**

a view of the values contained in this map

#### entrySet

public [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)<[Map.Entry](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>> entrySet()

Returns a [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation, or through the setValue operation on a map entry returned by the iterator) the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

**Specified by:**

[entrySet](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#entrySet--) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Specified by:**

[entrySet](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#entrySet--) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Returns:**

a set view of the mappings contained in this map

#### getOrDefault

* + - public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) getOrDefault([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key,

[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) defaultValue)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#getOrDefault-java.lang.Object-V-)

Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.

**Specified by:**

[getOrDefault](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#getOrDefault-java.lang.Object-V-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - the key whose associated value is to be returned

defaultValue - the default mapping of the key

**Returns:**

the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key

#### putIfAbsent

* + - public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) putIfAbsent([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key,

[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#putIfAbsent-K-V-)

If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

**Specified by:**

[putIfAbsent](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#putIfAbsent-K-V-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the specified value is to be associated

value - value to be associated with the specified key

**Returns:**

the previous value associated with the specified key, or null if there was no mapping for the key. (A null return can also indicate that the map previously associated null with the key, if the implementation supports null values.)

#### remove

* + - public boolean remove([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key,

[Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) value)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#remove-java.lang.Object-java.lang.Object-)

Removes the entry for the specified key only if it is currently mapped to the specified value.

**Specified by:**

[remove](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#remove-java.lang.Object-java.lang.Object-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the specified value is associated

value - value expected to be associated with the specified key

**Returns:**

true if the value was removed

#### replace

* + - public boolean replace([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key,
    - [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) oldValue,

[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) newValue)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#replace-K-V-V-)

Replaces the entry for the specified key only if currently mapped to the specified value.

**Specified by:**

[replace](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#replace-K-V-V-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the specified value is associated

oldValue - value expected to be associated with the specified key

newValue - value to be associated with the specified key

**Returns:**

true if the value was replaced

#### replace

* + - public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) replace([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key,

[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#replace-K-V-)

Replaces the entry for the specified key only if it is currently mapped to some value.

**Specified by:**

[replace](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#replace-K-V-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the specified value is associated

value - value to be associated with the specified key

**Returns:**

the previous value associated with the specified key, or null if there was no mapping for the key. (A null return can also indicate that the map previously associated null with the key, if the implementation supports null values.)

#### computeIfAbsent

* + - public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) computeIfAbsent([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key,

[Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> mappingFunction)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#computeIfAbsent-K-java.util.function.Function-)

If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.

If the function returns null no mapping is recorded. If the function itself throws an (unchecked) exception, the exception is rethrown, and no mapping is recorded. The most common usage is to construct a new object serving as an initial mapped value or memoized result, as in:

map.computeIfAbsent(key, k -> new Value(f(k)));

Or to implement a multi-value map, Map<K,Collection<V>>, supporting multiple values per key:

map.computeIfAbsent(key, k -> new HashSet<V>()).add(v);

**Specified by:**

[computeIfAbsent](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#computeIfAbsent-K-java.util.function.Function-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the specified value is to be associated

mappingFunction - the function to compute a value

**Returns:**

the current (existing or computed) value associated with the specified key, or null if the computed value is null

#### computeIfPresent

* + - public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) computeIfPresent([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key,

[BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<? super [K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> remappingFunction)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#computeIfPresent-K-java.util.function.BiFunction-)

If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

If the function returns null, the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

**Specified by:**

[computeIfPresent](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#computeIfPresent-K-java.util.function.BiFunction-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the specified value is to be associated

remappingFunction - the function to compute a value

**Returns:**

the new value associated with the specified key, or null if none

#### compute

* + - public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) compute([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key,

[BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<? super [K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> remappingFunction)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#compute-K-java.util.function.BiFunction-)

Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). For example, to either create or append a Stringmsg to a value mapping:

map.compute(key, (k, v) -> (v == null) ? msg : v.concat(msg))

(Method [merge()](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#merge-K-V-java.util.function.BiFunction-) is often simpler to use for such purposes.)

If the function returns null, the mapping is removed (or remains absent if initially absent). If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

**Specified by:**

[compute](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#compute-K-java.util.function.BiFunction-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the specified value is to be associated

remappingFunction - the function to compute a value

**Returns:**

the new value associated with the specified key, or null if none

#### merge

* + - public [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) merge([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key,
    - [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value,

[BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<? super [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> remappingFunction)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#merge-K-V-java.util.function.BiFunction-)

If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null. This method may be of use when combining multiple mapped values for a key. For example, to either create or append a String msg to a value mapping:

map.merge(key, msg, String::concat)

If the function returns null the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

**Specified by:**

[merge](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#merge-K-V-java.util.function.BiFunction-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

key - key with which the resulting value is to be associated

value - the non-null value to be merged with the existing value associated with the key or, if no existing value or a null value is associated with the key, to be associated with the key

remappingFunction - the function to recompute a value if present

**Returns:**

the new value associated with the specified key, or null if no value is associated with the key

#### forEach

public void forEach([BiConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html)<? super [K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> action)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#forEach-java.util.function.BiConsumer-)

Performs the given action for each entry in this map until all entries have been processed or the action throws an exception. Unless otherwise specified by the implementing class, actions are performed in the order of entry set iteration (if an iteration order is specified.) Exceptions thrown by the action are relayed to the caller.

**Specified by:**

[forEach](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#forEach-java.util.function.BiConsumer-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

action - The action to be performed for each entry

#### replaceAll

public void replaceAll([BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<? super [K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? super [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> function)

**Description copied from interface:**[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#replaceAll-java.util.function.BiFunction-)

Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. Exceptions thrown by the function are relayed to the caller.

**Specified by:**

[replaceAll](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#replaceAll-java.util.function.BiFunction-) in interface [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Parameters:**

function - the function to apply to each entry

#### clone

public [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) clone()

Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.

**Overrides:**

[clone](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html#clone--) in class [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>

**Returns:**

a shallow copy of this map

**See Also:**

[Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html)

# **Java HashMap class**

Java HashMap class hierarchy

Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.

The important points about Java HashMap class are:

* A HashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains no order.

### **Hierarchy of HashMap class**

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.

### **HashMap class declaration**

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

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

### **HashMap class Parameters**

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

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

### **Constructors of Java HashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashMap() | It is used to construct a default HashMap. |
| HashMap(Map m) | It is used to initializes the hash map by using the elements of the given Map object m. |
| HashMap(int capacity) | It is used to initializes the capacity of the hash map to the given integer value, capacity. |
| HashMap(int capacity, float fillRatio) | It is used to initialize both the capacity and fill ratio of the hash map by using its arguments. |

### **Methods of Java HashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| Object put(Object key, Object value) | It is used to associate the specified value with the specified key in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |
| Collection values() | It is used to return a collection view of the values contained in this map. |

### **Java HashMap Example**

1. **import** java.util.\*;
2. **class** TestCollection13{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(101,"Vijay");
7. hm.put(102,"Rahul");
8. **for**(Map.Entry m:hm.entrySet()){
9. System.out.println(m.getKey()+" "+m.getValue());
10. }
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection13)

Output:102 Rahul

100 Amit

101 Vijay

### **Java HashMap Example: remove()**

1. **import** java.util.\*;
2. **public** **class** HashMapExample {
3. **public** **static** **void** main(String args[]) {
4. // create and populate hash map
5. HashMap<Integer, String> map = **new** HashMap<Integer, String>();
6. map.put(101,"Let us C");
7. map.put(102, "Operating System");
8. map.put(103, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

Output:

Values before remove: {102=Operating System, 103=Data Communication and Networking, 101=Let us C}

Values after remove: {103=Data Communication and Networking, 101=Let us C}

### **Difference between HashSet and HashMap**

HashSet contains only values whereas HashMap contains entry(key and value).

### **Java HashMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** HashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(1,b1);
24. map.put(2,b2);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

# **Java LinkedHashMap class**

Java LinkedHashMap class hierarchy

Java LinkedHashMap class is Hash table and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

The important points about Java LinkedHashMap class are:

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead maintains insertion order.

### **LinkedHashMap class declaration**

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

1. **public** **class** LinkedHashMap<K,V> **extends** HashMap<K,V> **implements** Map<K,V>

### **LinkedHashMap class Parameters**

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

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

### **Constructors of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedHashMap() | It is used to construct a default LinkedHashMap. |
| LinkedHashMap(int capacity) | It is used to initialize a LinkedHashMap with the given capacity. |
| LinkedHashMap(int capacity, float fillRatio) | It is used to initialize both the capacity and the fillRatio. |
| LinkedHashMap(Map m) | It is used to initialize the LinkedHashMap with the elements from the given Map class m. |

### **Methods of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object get(Object key) | It is used to return the value to which this map maps the specified key. |
| void clear() | It is used to remove all mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map maps one or more keys to the specified value. |

### **Java LinkedHashMap Example**

1. **import** java.util.\*;
2. **class** TestCollection14{
3. **public** **static** **void** main(String args[]){
5. LinkedHashMap<Integer,String> hm=**new** LinkedHashMap<Integer,String>();
7. hm.put(100,"Amit");
8. hm.put(101,"Vijay");
9. hm.put(102,"Rahul");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection14)

Output:100 Amit

101 Vijay

102 Rahul

### **Java LinkedHashMap Example:remove()**

1. **import** java.util.\*;
2. **public** **class** LinkedHashMapExample {
3. **public** **static** **void** main(String args[]) {
4. // Create and populate linked hash map
5. Map<Integer, String> map = **new** LinkedHashMap<Integer, String>();
6. map.put(101,"Let us C");
7. map.put(102, "Operating System");
8. map.put(103, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

Output:

Values before remove: {101=Let us C, 102=Operating System, 103=Data Communication and Networking}

Values after remove: {101=Let us C, 103=Data Communication and Networking}

### **Java LinkedHashMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** LinkedHashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(2,b2);
24. map.put(1,b1);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

# **Java TreeMap class**

Java TreeMap class hierarchy

Java TreeMap class implements the Map interface by using a tree. It provides an efficient means of storing key/value pairs in sorted order.

The important points about Java TreeMap class are:

* A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead maintains ascending order.

### **TreeMap class declaration**

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

1. **public** **class** TreeMap<K,V> **extends** AbstractMap<K,V> **implements** NavigableMap<K,V>, Cloneable, Serializable

### **TreeMap class Parameters**

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

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

### **Constructors of Java TreeMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeMap() | It is used to construct an empty tree map that will be sorted using the natural order of its key. |
| TreeMap(Comparator comp) | It is used to construct an empty tree-based map that will be sorted using the comparator comp. |
| TreeMap(Map m) | It is used to initialize a tree map with the entries from **m**, which will be sorted using the natural order of the keys. |
| TreeMap(SortedMap sm) | It is used to initialize a tree map with the entries from the SortedMap **sm**, which will be sorted in the same order as **sm.** |

### **Methods of Java TreeMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| Object firstKey() | It is used to return the first (lowest) key currently in this sorted map. |
| Object get(Object key) | It is used to return the value to which this map maps the specified key. |
| Object lastKey() | It is used to return the last (highest) key currently in this sorted map. |
| Object remove(Object key) | It is used to remove the mapping for this key from this TreeMap if present. |
| void putAll(Map map) | It is used to copy all of the mappings from the specified map to this map. |
| Set entrySet() | It is used to return a set view of the mappings contained in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |
| Collection values() | It is used to return a collection view of the values contained in this map. |

### **Java TreeMap Example:**

1. **import** java.util.\*;
2. **class** TestCollection15{
3. **public** **static** **void** main(String args[]){
4. TreeMap<Integer,String> hm=**new** TreeMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(102,"Ravi");
7. hm.put(101,"Vijay");
8. hm.put(103,"Rahul");
9. **for**(Map.Entry m:hm.entrySet()){
10. System.out.println(m.getKey()+" "+m.getValue());
11. }
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection15)

Output:100 Amit

101 Vijay

102 Ravi

103 Rahul

### **Java TreeMap Example: remove()**

1. **import** java.util.\*;
2. **public** **class** TreeMapExample {
3. **public** **static** **void** main(String args[]) {
4. // Create and populate tree map
5. Map<Integer, String> map = **new** TreeMap<Integer, String>();
6. map.put(102,"Let us C");
7. map.put(103, "Operating System");
8. map.put(101, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

Output:

Values before remove: {101=Data Communication and Networking, 102=Let us C, 103=Operating System}

Values after remove: {101=Data Communication and Networking, 103=Operating System}

### **What is difference between HashMap and TreeMap?**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap can contain one null key. | TreeMap can not contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

### **Java TreeMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** TreeMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(2,b2);
24. map.put(1,b1);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

# **Java Hashtable class**

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

The important points about Java Hashtable class are:

* A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* It contains only unique elements.
* It may have not have any null key or value.
* It is synchronized.

### **Hashtable class declaration**

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

1. **public** **class** Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map<K,V>, Cloneable, Serializable

### **Hashtable class Parameters**

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

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

### **Constructors of Java Hashtable class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| Hashtable() | It is the default constructor of hash table it instantiates the Hashtable class. |
| Hashtable(int size) | It is used to accept an integer parameter and creates a hash table that has an initial size specified by integer value size. |
| Hashtable(int size, float fillRatio) | It is used to create a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. |

### **Methods of Java Hashtable class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to reset the hash table. |
| boolean contains(Object value) | This method return true if some value equal to the value exist within the hash table, else return false. |
| boolean containsValue(Object value) | This method return true if some value equal to the value exists within the hash table, else return false. |
| boolean containsKey(Object key) | This method return true if some key equal to the key exists within the hash table, else return false. |
| boolean isEmpty() | This method return true if the hash table is empty; returns false if it contains at least one key. |
| void rehash() | It is used to increase the size of the hash table and rehashes all of its keys. |
| Object get(Object key) | This method return the object that contains the value associated with the key. |
| Object remove(Object key) | It is used to remove the key and its value. This method return the value associated with the key. |
| int size() | This method return the number of entries in the hash table. |

### **Java Hashtable Example**

1. **import** java.util.\*;
2. **class** TestCollection16{
3. **public** **static** **void** main(String args[]){
4. Hashtable<Integer,String> hm=**new** Hashtable<Integer,String>();
6. hm.put(100,"Amit");
7. hm.put(102,"Ravi");
8. hm.put(101,"Vijay");
9. hm.put(103,"Rahul");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection16)

Output:

103 Rahul

102 Ravi

101 Vijay

100 Amit

### **Java Hashtable Example: remove()**

1. **import** java.util.\*;
2. **public** **class** HashtableExample {
3. **public** **static** **void** main(String args[]) {
4. // create and populate hash table
5. Hashtable<Integer, String> map = **new** Hashtable<Integer, String>();
6. map.put(102,"Let us C");
7. map.put(103, "Operating System");
8. map.put(101, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

Output:

Values before remove: {103=Operating System, 102=Let us C, 101=Data Communication and Networking}

Values after remove: {103=Operating System, 101=Data Communication and Networking}

### **Java Hashtable Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** HashtableExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** Hashtable<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(1,b1);
24. map.put(2,b2);
25. map.put(3,b3);
26. //Traversing map
27. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
28. **int** key=entry.getKey();
29. Book b=entry.getValue();
30. System.out.println(key+" Details:");
31. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
32. }
33. }
34. }

# **Difference between HashMap and Hashtable**

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

# **Java EnumSet class**

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

### **EnumSet class hierarchy**

The hierarchy of EnumSet class is given in the figure given below.

EnumSet class hierarchy

## EnumSet class declaration

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

1. **public** **abstract** **class** EnumSet<E **extends** Enum<E>> **extends** AbstractSet<E> **implements** Cloneable, Serializable

### **Methods of Java EnumSet class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType) | It is used to create an enum set containing all of the elements in the specified element type. |
| static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c) | It is used to create an enum set initialized from the specified collection. |
| static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) | It is used to create an empty enum set with the specified element type. |
| static <E extends Enum<E>> EnumSet<E> of(E e) | It is used to create an enum set initially containing the specified element. |
| static <E extends Enum<E>> EnumSet<E> range(E from, E to) | It is used to create an enum set initially containing the specified elements. |
| EnumSet<E> clone() | It is used to return a copy of this set. |

## Java EnumSet Example

1. **import** java.util.\*;
2. **enum** days {
3. SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
4. }
5. **public** **class** EnumSetExample {
6. **public** **static** **void** main(String[] args) {
7. Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
8. // Traversing elements
9. Iterator<days> iter = set.iterator();
10. **while** (iter.hasNext())
11. System.out.println(iter.next());
12. }
13. }

Output:

TUESDAY

WEDNESDAY

## Java EnumSet Example: allOf() and noneOf()

1. **import** java.util.\*;
2. **enum** days {
3. SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
4. }
5. **public** **class** EnumSetExample {
6. **public** **static** **void** main(String[] args) {
7. Set<days> set1 = EnumSet.allOf(days.**class**);
8. System.out.println("Week Days:"+set1);
9. Set<days> set2 = EnumSet.noneOf(days.**class**);
10. System.out.println("Week Days:"+set2);
11. }
12. }

# **Java EnumMap class**

Java EnumMap class is the specialized Map implementation for enum keys. It inherits Enum and AbstractMap classes.

### **EnumMap class hierarchy**

The hierarchy of EnumMap class is given in the figure given below.

EnumMap class hierarchy

## EnumMap class declaration

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

1. **public** **class** EnumMap<K **extends** Enum<K>,V> **extends** AbstractMap<K,V> **implements** Serializable, Cloneable

## EnumMap class Parameters

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

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

### **Constructors of Java EnumMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| EnumMap(Class<K> keyType) | It is used to create an empty enum map with the specified key type. |
| EnumMap(EnumMap<K,? extends V> m) | It is used to create an enum map with the same key type as the specified enum map. |
| EnumMap(Map<K,? extends V> m) | It is used to create an enum map initialized from the specified map. |

### **Methods of Java EnumMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all mappings from this map. |
| boolean containsKey(Object key) | This method return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | This method return true if this map maps one or more keys to the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with this map for equality. |
| V get(Object key) | This method returns the value to which the specified key is mapped. |
| V put(K key, V value) | It is used to associate the specified value with the specified key in this map. |
| V remove(Object key) | It is used to remove the mapping for this key. |
| Collection<V> values() | It is used to return a Collection view of the values contained in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |

## Java EnumMap Example

1. **import** java.util.\*;
2. **public** **class** EnumMapExample {
3. // create an enum
4. **public** **enum** Days {
5. Monday, Tuesday, Wednesday, Thursday
6. };
7. **public** **static** **void** main(String[] args) {
8. //create and populate enum map
9. EnumMap<Days, String> map = **new** EnumMap<Days, String>(Days.**class**);
10. map.put(Days.Monday, "1");
11. map.put(Days.Tuesday, "2");
12. map.put(Days.Wednesday, "3");
13. map.put(Days.Thursday, "4");
14. // print the map
15. **for**(Map.Entry m:map.entrySet()){
16. System.out.println(m.getKey()+" "+m.getValue());
17. }
18. }
19. }

Output:

Monday 1

Tuesday 2

Wednesday 3

Thursday 4

## Java EnumMap Example: Book

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** EnumMapExample {
15. // Creating enum
16. **public** **enum** Key{
17. One, Two, Three
18. };
19. **public** **static** **void** main(String[] args) {
20. EnumMap<Key, Book> map = **new** EnumMap<Key, Book>(Key.**class**);
21. // Creating Books
22. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
23. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
24. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
25. // Adding Books to Map
26. map.put(Key.One, b1);
27. map.put(Key.Two, b2);
28. map.put(Key.Three, b3);
29. // Traversing EnumMap
30. **for**(Map.Entry<Key, Book> entry:map.entrySet()){
31. Book b=entry.getValue();
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

# **Java Collections class**

Java collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

The important points about Java Collections class are:

* Java Collection class supports the **polymorphic algorithms** that operate on collections.
* Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

## Collections class declaration

Let's see the declaration for Java.util.Collections class.

1. **public** **class** Collections **extends** Object

### **Methods of Java Collections class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| static <T> boolean addAll(Collection<? super T> c, T... elements) | It is used to add all of the specified elements to the specified collection. |
| static <T> Queue<T> asLifoQueue(Deque<T> deque) | It is used to return a view of a Deque as a Last-In-First-Out (LIFO) Queue. |
| static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T< c) | It is used to search the specified list for the specified object using the binary search algorithm. |
| static <E> List<E> checkedList(List<E> list, Class<E> type) | It is used to return a dynamically typesafe view of the specified list. |
| static <E> Set<E> checkedSet(Set<E> s, Class<E> type) | It is used to return a dynamically typesafe view of the specified set. |
| static <E> SortedSet<E>checkedSortedSet(SortedSet<E> s, Class<E> type) | It is used to return a dynamically typesafe view of the specified sorted set |
| static void reverse(List<?> list) | It is used to reverse the order of the elements in the specified list. |
| static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp) | It is used to return the maximum element of the given collection, according to the order induced by the specified comparator. |
| static <T extends Object & Comparable<? super T>>T min(Collection<? extends T> coll) | It is used to return the minimum element of the given collection, according to the natural ordering of its elements. |
| static boolean replaceAll(List list, T oldVal, T newVal) | It is used to replace all occurrences of one specified value in a list with another. |

## Java Collections Example

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<String> list = **new** ArrayList<String>();
5. list.add("C");
6. list.add("Core Java");
7. list.add("Advance Java");
8. System.out.println("Initial collection value:"+list);
9. Collections.addAll(list, "Servlet","JSP");
10. System.out.println("After adding elements collection value:"+list);
11. String[] strArr = {"C#", ".Net"};
12. Collections.addAll(list, strArr);
13. System.out.println("After adding array collection value:"+list);
14. }
15. }

Output:

Initial collection value:[C, Core Java, Advance Java]

After adding elements collection value:[C, Core Java, Advance Java, Servlet, JSP]

After adding array collection value:[C, Core Java, Advance Java, Servlet, JSP, C#, .Net]

## Java Collections Example: max()

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<Integer> list = **new** ArrayList<Integer>();
5. list.add(46);
6. list.add(67);
7. list.add(24);
8. list.add(16);
9. list.add(8);
10. list.add(12);
11. System.out.println("Value of maximum element from the collection: "+Collections.max(list));
12. }
13. }

Output:

Value of maximum element from the collection: 67

## Java Collections Example: min()

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<Integer> list = **new** ArrayList<Integer>();
5. list.add(46);
6. list.add(67);
7. list.add(24);
8. list.add(16);
9. list.add(8);
10. list.add(12);
11. System.out.println("Value of minimum element from the collection: "+Collections.min(list));
12. }
13. }

# **Sorting in Collection**

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

|  |
| --- |
| **Collections** class provides static methods for sorting the elements of collection.If collection elements are of Set type, we can use TreeSet.But We cannot sort the elements of List.Collections class provides methods for sorting the elements of List type elements. |

### **Method of Collections class for sorting List elements**

|  |
| --- |
| **public void sort(List list):** is used to sort the elements of List.List elements must be of Comparable type. |

#### Note: String class and Wrapper classes implements the Comparable interface.So if you store the objects of string or wrapper classes, it will be Comparable.

### **Example of Sorting the elements of List that contains string objects**

1. **import** java.util.\*;
2. **class** TestSort1{
3. **public** **static** **void** main(String args[]){
5. ArrayList<String> al=**new** ArrayList<String>();
6. al.add("Viru");
7. al.add("Saurav");
8. al.add("Mukesh");
9. al.add("Tahir");
11. Collections.sort(al);
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSort1)

Output:Mukesh

Saurav

Tahir

Viru

### **Example of Sorting the elements of List that contains Wrapper class objects**

1. **import** java.util.\*;
2. **class** TestSort2{
3. **public** **static** **void** main(String args[]){
5. ArrayList al=**new** ArrayList();
6. al.add(Integer.valueOf(201));
7. al.add(Integer.valueOf(101));
8. al.add(230);//internally will be converted into objects as Integer.valueOf(230)
10. Collections.sort(al);
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

# **Java Comparable interface**

Java Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object). It provide single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be rollno, name, age or anything else.

### **compareTo(Object obj) method**

**public int compareTo(Object obj):** is used to compare the current object with the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

### **Collections class**

**Collections** class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But We cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

### **Method of Collections class for sorting List elements**

**public void sort(List list):** is used to sort the elements of List. List elements must be of Comparable type.

#### Note: String class and Wrapper classes implements Comparable interface by default. So if you store the objects of string or wrapper classes in list, set or map, it will be Comparable by default.

## Java Comparable Example

Let's see the example of Comparable interface that sorts the list elements on the basis of age.

*File: Student.java*

1. **class** Student **implements** Comparable<Student>{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
11. **public** **int** compareTo(Student st){
12. **if**(age==st.age)
13. **return** 0;
14. **else** **if**(age>st.age)
15. **return** 1;
16. **else**
17. **return** -1;
18. }
19. }

*File: TestSort3.java*

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** TestSort3{
4. **public** **static** **void** main(String args[]){
5. ArrayList<Student> al=**new** ArrayList<Student>();
6. al.add(**new** Student(101,"Vijay",23));
7. al.add(**new** Student(106,"Ajay",27));
8. al.add(**new** Student(105,"Jai",21));
10. Collections.sort(al);
11. **for**(Student st:al){
12. System.out.println(st.rollno+" "+st.name+" "+st.age);
13. }
14. }
15. }

# **Java Comparator interface**

**Java Comparator interface** is used to order the objects of user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

#### **compare() method**

**public int compare(Object obj1,Object obj2):** compares the first object with second object.

## Collections class

**Collections** class provides static methods for sorting the elements of collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

#### **Method of Collections class for sorting List elements**

**public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

## Java Comparator Example (Non-generic Old Style)

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

1. Student.java
2. AgeComparator.java
3. NameComparator.java
4. Simple.java

**Student.java**

This class contains three fields rollno, name and age and a parameterized constructor.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

**AgeComparator.java**

This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2 , 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator{
3. **public** **int** compare(Object o1,Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
7. **if**(s1.age==s2.age)
8. **return** 0;
9. **else** **if**(s1.age>s2.age)
10. **return** 1;
11. **else**
12. **return** -1;
13. }
14. }

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator{
3. **public** **int** compare(Object o1,Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
7. **return** s1.name.compareTo(s2.name);
8. }
9. }

**Simple.java**

In this class, we are printing the objects values by sorting on the basis of name and age.

1. **import** java.util.\*;
2. **import** java.io.\*;
4. **class** Simple{
5. **public** **static** **void** main(String args[]){
7. ArrayList al=**new** ArrayList();
8. al.add(**new** Student(101,"Vijay",23));
9. al.add(**new** Student(106,"Ajay",27));
10. al.add(**new** Student(105,"Jai",21));
12. System.out.println("Sorting by Name...");
14. Collections.sort(al,**new** NameComparator());
15. Iterator itr=al.iterator();
16. **while**(itr.hasNext()){
17. Student st=(Student)itr.next();
18. System.out.println(st.rollno+" "+st.name+" "+st.age);
19. }
21. System.out.println("sorting by age...");
23. Collections.sort(al,**new** AgeComparator());
24. Iterator itr2=al.iterator();
25. **while**(itr2.hasNext()){
26. Student st=(Student)itr2.next();
27. System.out.println(st.rollno+" "+st.name+" "+st.age);
28. }

31. }
32. }

Sorting by Name...

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by age...

105 Jai 21

101 Vijay 23

106 Ajay 27

## Java Comparator Example (Generic)

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

**AgeComparator.java**

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **if**(s1.age==s2.age)
5. **return** 0;
6. **else** **if**(s1.age>s2.age)
7. **return** 1;
8. **else**
9. **return** -1;
10. }
11. }

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **return** s1.name.compareTo(s2.name);
5. }
6. }

**Simple.java**

In this class, we are printing the objects values by sorting on the basis of name and age.

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **class** Simple{
4. **public** **static** **void** main(String args[]){
6. ArrayList<Student> al=**new** ArrayList<Student>();
7. al.add(**new** Student(101,"Vijay",23));
8. al.add(**new** Student(106,"Ajay",27));
9. al.add(**new** Student(105,"Jai",21));
11. System.out.println("Sorting by Name...");
13. Collections.sort(al,**new** NameComparator());
14. **for**(Student st: al){
15. System.out.println(st.rollno+" "+st.name+" "+st.age);
16. }
18. System.out.println("sorting by age...");
20. Collections.sort(al,**new** AgeComparator());
21. **for**(Student st: al){
22. System.out.println(st.rollno+" "+st.name+" "+st.age);
23. }
25. }
26. }

# **Difference between Comparable and Comparator**

Comparable and Comparator both are interfaces and can be used to sort collection elements.

But there are many differences between Comparable and Comparator interfaces that are given below.

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc. |
| 2) Comparable **affects the original class** i.e. actual class is modified. | Comparator **doesn't affect the original class** i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method. |

# **Properties class in Java**

The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from properties file and store data into properties file. Moreover, it can be used to get properties of system.

### **Advantage of properties file**

**Recompilation is not required, if information is changed from properties file:** If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

#### **Methods of Properties class**

The commonly used methods of Properties class are given below.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void load(Reader r) | loads data from the Reader object. |
| public void load(InputStream is) | loads data from the InputStream object |
| public String getProperty(String key) | returns value based on the key. |
| public void setProperty(String key,String value) | sets the property in the properties object. |
| public void store(Writer w, String comment) | writers the properties in the writer object. |
| public void store(OutputStream os, String comment) | writes the properties in the OutputStream object. |
| storeToXML(OutputStream os, String comment) | writers the properties in the writer object for generating xml document. |
| public void storeToXML(Writer w, String comment, String encoding) | writers the properties in the writer object for generating xml document with specified encoding. |

### **Example of Properties class to get information from properties file**

To get information from the properties file, create the properties file first.

**db.properties**

1. user=system
2. password=oracle

Now, lets create the java class to read the data from the properties file.

**Test.java**

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
5. FileReader reader=**new** FileReader("db.properties");
7. Properties p=**new** Properties();
8. p.load(reader);
10. System.out.println(p.getProperty("user"));
11. System.out.println(p.getProperty("password"));
12. }
13. }

Output:system

oracle

Now if you change the value of the properties file, you don't need to compile the java class again. That means no maintenance problem.

### **Example of Properties class to get all the system properties**

By System.getProperties() method we can get all the properties of system. Let's create the class that gets information from the system properties.

**Test.java**

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
6. Properties p=System.getProperties();
7. Set set=p.entrySet();
9. Iterator itr=set.iterator();
10. **while**(itr.hasNext()){
11. Map.Entry entry=(Map.Entry)itr.next();
12. System.out.println(entry.getKey()+" = "+entry.getValue());
13. }
15. }
16. }

Output:

java.runtime.name = Java(TM) SE Runtime Environment

sun.boot.library.path = C:\Program Files\Java\jdk1.7.0\_01\jre\bin

java.vm.version = 21.1-b02

java.vm.vendor = Oracle Corporation

java.vendor.url = http://java.oracle.com/

path.separator = ;

java.vm.name = Java HotSpot(TM) Client VM

file.encoding.pkg = sun.io

user.country = US

user.script =

sun.java.launcher = SUN\_STANDARD

...........

### **Example of Properties class to create properties file**

Now lets write the code to create the properties file.

**Test.java**

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
6. Properties p=**new** Properties();
7. p.setProperty("name","Sonoo Jaiswal");
8. p.setProperty("email","sonoojaiswal@javatpoint.com");
10. p.store(**new** FileWriter("info.properties"),"Javatpoint Properties Example");
12. }
13. }

# **Difference between ArrayList and Vector**

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. |

### **Example of Java ArrayList**

Let's see a simple example where we are using ArrayList to store and traverse the elements.

1. **import** java.util.\*;
2. **class** TestArrayList21{
3. **public** **static** **void** main(String args[]){
5. List<String> al=**new** ArrayList<String>();//creating arraylist
6. al.add("Sonoo");//adding object in arraylist
7. al.add("Michael");
8. al.add("James");
9. al.add("Andy");
10. //traversing elements using Iterator
11. Iterator itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestArrayList21)

Output:

Sonoo

Michael

James

Andy

### **Example of Java Vector**

Let's see a simple example of java Vector class that uses Enumeration interface.

1. **import** java.util.\*;
2. **class** TestVector1{
3. **public** **static** **void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();//creating vector
5. v.add("umesh");//method of Collection
6. v.addElement("irfan");//method of Vector
7. v.addElement("kumar");
8. //traversing elements using Enumeration
9. Enumeration e=v.elements();
10. **while**(e.hasMoreElements()){
11. System.out.println(e.nextElement());
12. }
13. }
14. }

# **Java JDBC Tutorial**

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.

JDBC (Java Database Connectivity) 

### **Why use JDBC**

Before JDBC, ODBC API was the database API to connect and execute query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

Do You Know

* How to connect Java application with Oracle and Mysql database using JDBC?
* What is the difference between Statement and PreparedStatement interface?
* How to print total numbers of tables and views of a database using JDBC ?
* How to store and retrieve images from Oracle database using JDBC?
* How to store and retrieve files from Oracle database using JDBC?

## What is API

API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

## Topics in Java JDBC Tutorial

[2) JDBC Drivers](https://www.javatpoint.com/jdbc-driver)

In this JDBC tutorial, we will learn 4 types of JDBC drivers, their advantages and disadvantages.

[3) 5 Steps to connect to the database](https://www.javatpoint.com/steps-to-connect-to-the-database-in-java)

In this JDBC tutorial, we will see the 5 steps to connect to the database in java using JDBC.

[4) Connectivity with Oracle using JDBC](https://www.javatpoint.com/example-to-connect-to-the-oracle-database)

In this JDBC tutorial, we will connect a simple java program with the oracle database.

[5) Connectivity with MySQL using JDBC](https://www.javatpoint.com/example-to-connect-to-the-mysql-database)

In this JDBC tutorial, we will connect a simple java program with the mysql database.

[6) Connectivity with Access without DSN](https://www.javatpoint.com/connectivity-with-access-without-dsn)

Let's connect java application with access database with and without DSN.

[7) DriverManager class](https://www.javatpoint.com/DriverManager-class)

In this JDBC tutorial, we will learn what does the DriverManager class and what are its methods.

[8) Connection interface](https://www.javatpoint.com/Connection-interface)

In this JDBC tutorial, we will learn what is Connection interface and what are its methods.

[9) Statement interface](https://www.javatpoint.com/Statement-interface)

In this JDBC tutorial, we will learn what is Statement interface and what are its methods.

[10) ResultSet interface](https://www.javatpoint.com/ResultSet-interface)

In this JDBC tutorial, we will learn what is ResultSet interface and what are its methods. Moreover, we will learn how we can make the ResultSet scrollable.

[11) PreparedStatement Interface](https://www.javatpoint.com/PreparedStatement-interface)

In this JDBC tutorial, we will learn what is benefit of PreparedStatement over Statement interface. We will see examples to insert, update or delete records using the PreparedStatement interface.

[12) ResultSetMetaData interface](https://www.javatpoint.com/ResultSetMetaData-interface)

In this JDBC tutorial, we will learn how we can get the metadata of a table.

[13) DatabaseMetaData interface](https://www.javatpoint.com/DatabaseMetaData-interface)

In this JDBC tutorial, we will learn how we can get the metadata of a database.

[14) Storing image in Oracle](https://www.javatpoint.com/storing-image-in-oracle-database)

Let's learn how to store image in the oracle database using JDBC.

[15) Retrieving image from Oracle](https://www.javatpoint.com/retrieving-image-from-oracle-database)

Let's see the simple example to retrieve image from the oracle database using JDBC.

[16) Storing file in Oracle](https://www.javatpoint.com/storing-file-in-oracle-database)

Let's see the simple example to store file in the oracle database using JDBC.

[17) Retrieving file from Oracle](https://www.javatpoint.com/retrieving-file-from-oracle-database)

Let's see the simple example to retrieve file from the oracle database using JDBC.

[18) CallableStatement](https://www.javatpoint.com/CallableStatement-interface)

Let's see the code to call stored procedures and functions using CallableStatement.

[19) Transaction Management using JDBC](https://www.javatpoint.com/transaction-management-in-jdbc)

Let's see the simple example to use transaction management using JDBC.

[20) Batch Statement using JDBC](https://www.javatpoint.com/batch-processing-in-jdbc)

Let's see the code to execute batch of queries.

[21) JDBC RowSet](https://www.javatpoint.com/jdbc-rowset)

Let's see the working of new JDBC RowSet interface.

# **JDBC Driver**

1. [JDBC Drivers](https://www.javatpoint.com/jdbc-driver)
   1. [JDBC-ODBC bridge driver](https://www.javatpoint.com/jdbc-driver#driver1)
   2. [Native-API driver](https://www.javatpoint.com/jdbc-driver#driver2)
   3. [Network Protocol driver](https://www.javatpoint.com/jdbc-driver#driver3)
   4. [Thin driver](https://www.javatpoint.com/jdbc-driver#driver4)

|  |
| --- |
| JDBC Driver is a software component that enables java application to interact with the database.There are 4 types of JDBC drivers:   1. JDBC-ODBC bridge driver 2. Native-API driver (partially java driver) 3. Network Protocol driver (fully java driver) 4. Thin driver (fully java driver) |

### **1) JDBC-ODBC bridge driver**

|  |
| --- |
| The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver. |



### **Advantages:**

* easy to use.
* can be easily connected to any database.

### **Disadvantages:**

* Performance degraded because JDBC method call is converted into the ODBC function calls.
* The ODBC driver needs to be installed on the client machine.

### **2) Native-API driver**

|  |
| --- |
| The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java. |



### **Advantage:**

* performance upgraded than JDBC-ODBC bridge driver.

### **Disadvantage:**

* The Native driver needs to be installed on the each client machine.
* The Vendor client library needs to be installed on client machine.

### **3) Network Protocol driver**

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.



### **Advantage:**

* No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

### **Disadvantages:**

* Network support is required on client machine.
* Requires database-specific coding to be done in the middle tier.
* Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

### **4) Thin driver**

|  |
| --- |
| The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language. |



### **Advantage:**

* Better performance than all other drivers.
* No software is required at client side or server side.

### **Disadvantage:**

* Drivers depends on the Database.

# **5 Steps to connect to the database in java**

1. [5 Steps to connect to the database in java](https://www.javatpoint.com/steps-to-connect-to-the-database-in-java)
   1. [Register the driver class](https://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step1)
   2. [Create the connection object](https://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step2)
   3. [Create the Statement object](https://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step3)
   4. [Execute the query](https://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step4)
   5. [Close the connection object](https://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step5)

|  |
| --- |
| There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:   * Register the driver class * Creating connection * Creating statement * Executing queries * Closing connection |

### **1) Register the driver class**

|  |
| --- |
| The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class. |

### **Syntax of forName() method**

1. **public** **static** **void** forName(String className)**throws** ClassNotFoundException

### **Example to register the OracleDriver class**

1. Class.forName("oracle.jdbc.driver.OracleDriver");

### **2) Create the connection object**

|  |
| --- |
| The getConnection() method of DriverManager class is used to establish connection with the database. |

### **Syntax of getConnection() method**

1. 1) **public** **static** Connection getConnection(String url)**throws** SQLException
2. 2) **public** **static** Connection getConnection(String url,String name,String password)
3. **throws** SQLException

### **Example to establish connection with the Oracle database**

1. Connection con=DriverManager.getConnection(
2. "jdbc:oracle:thin:@localhost:1521:xe","system","password");

### **3) Create the Statement object**

|  |
| --- |
| The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database. |

### **Syntax of createStatement() method**

1. **public** Statement createStatement()**throws** SQLException

### **Example to create the statement object**

1. Statement stmt=con.createStatement();

### **4) Execute the query**

|  |
| --- |
| The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table. |

### **Syntax of executeQuery() method**

1. **public** ResultSet executeQuery(String sql)**throws** SQLException

### **Example to execute query**

1. ResultSet rs=stmt.executeQuery("select \* from emp");
3. **while**(rs.next()){
4. System.out.println(rs.getInt(1)+" "+rs.getString(2));
5. }

### **5) Close the connection object**

|  |
| --- |
| By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection. |

### **Syntax of close() method**

1. **public** **void** close()**throws** SQLException

### **Example to close connection**

1. con.close();

# **Example to connect to the Oracle database in java**

|  |
| --- |
| For connecting java application with the oracle database, you need to follow 5 steps to perform database connectivity. In this example we are using Oracle10g as the database. So we need to know following information for the oracle database:   1. **Driver class:**The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**. 2. **Connection URL:**The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file. 3. **Username:**The default username for the oracle database is **system**. 4. **Password:**Password is given by the user at the time of installing the oracle database. |

|  |
| --- |
| Let's first create a table in oracle database. |

1. create table emp(id number(10),name varchar2(40),age number(3));

### **Example to Connect Java Application with Oracle database**

In this example, system is the username and oracle is the password of the Oracle database.

1. **import** java.sql.\*;
2. **class** OracleCon{
3. **public** **static** **void** main(String args[]){
4. **try**{
5. //step1 load the driver class
6. Class.forName("oracle.jdbc.driver.OracleDriver");
8. //step2 create  the connection object
9. Connection con=DriverManager.getConnection(
10. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
12. //step3 create the statement object
13. Statement stmt=con.createStatement();
15. //step4 execute query
16. ResultSet rs=stmt.executeQuery("select \* from emp");
17. **while**(rs.next())
18. System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
20. //step5 close the connection object
21. con.close();
23. }**catch**(Exception e){ System.out.println(e);}
25. }
26. }

[download this example](https://www.javatpoint.com/src/jdbc/OracleCon.zip)

The above example will fetch all the records of emp table.

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.

[download the jar file ojdbc14.jar](https://www.javatpoint.com/src/jdbc/ojdbc14.jar)

### **Two ways to load the jar file:**

1. paste the ojdbc14.jar file in jre/lib/ext folder
2. set classpath

### **1) paste the ojdbc14.jar file in JRE/lib/ext folder:**

|  |
| --- |
| Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here. |

### **2) set classpath:**

|  |
| --- |
| There are two ways to set the classpath:   * temporary * permanent |

### **How to set the temporary classpath:**

|  |
| --- |
| Firstly, search the ojdbc14.jar file then open command prompt and write: |

1. C:>set classpath=c:\folder\ojdbc14.jar;.;

### **How to set the permanent classpath:**

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.; as C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.;

To see the slides of seting parmanent path [click here](https://www.javatpoint.com/how-to-set-path-in-java)

# **Example to connect to the Oracle database in java**

|  |
| --- |
| For connecting java application with the oracle database, you need to follow 5 steps to perform database connectivity. In this example we are using Oracle10g as the database. So we need to know following information for the oracle database:   1. **Driver class:**The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**. 2. **Connection URL:**The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file. 3. **Username:**The default username for the oracle database is **system**. 4. **Password:**Password is given by the user at the time of installing the oracle database. |

|  |
| --- |
| Let's first create a table in oracle database. |

1. create table emp(id number(10),name varchar2(40),age number(3));

### **Example to Connect Java Application with Oracle database**

In this example, system is the username and oracle is the password of the Oracle database.

1. **import** java.sql.\*;
2. **class** OracleCon{
3. **public** **static** **void** main(String args[]){
4. **try**{
5. //step1 load the driver class
6. Class.forName("oracle.jdbc.driver.OracleDriver");
8. //step2 create  the connection object
9. Connection con=DriverManager.getConnection(
10. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
12. //step3 create the statement object
13. Statement stmt=con.createStatement();
15. //step4 execute query
16. ResultSet rs=stmt.executeQuery("select \* from emp");
17. **while**(rs.next())
18. System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
20. //step5 close the connection object
21. con.close();
23. }**catch**(Exception e){ System.out.println(e);}
25. }
26. }

[download this example](https://www.javatpoint.com/src/jdbc/OracleCon.zip)

The above example will fetch all the records of emp table.

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.

[download the jar file ojdbc14.jar](https://www.javatpoint.com/src/jdbc/ojdbc14.jar)

### **Two ways to load the jar file:**

1. paste the ojdbc14.jar file in jre/lib/ext folder
2. set classpath

### **1) paste the ojdbc14.jar file in JRE/lib/ext folder:**

|  |
| --- |
| Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here. |

### **2) set classpath:**

|  |
| --- |
| There are two ways to set the classpath:   * temporary * permanent |

### **How to set the temporary classpath:**

|  |
| --- |
| Firstly, search the ojdbc14.jar file then open command prompt and write: |

1. C:>set classpath=c:\folder\ojdbc14.jar;.;

### **How to set the permanent classpath:**

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.; as C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.;

To see the slides of seting parmanent path [click here](https://www.javatpoint.com/how-to-set-path-in-java)

# **Connectivity with Access without DSN**

There are two ways to connect java application with the access database.

1. Without DSN (Data Source Name)
2. With DSN

Java is mostly used with Oracle, mysql, or DB2 database. So you can learn this topic only for knowledge.

### **Example to Connect Java Application with access without DSN**

In this example, we are going to connect the java program with the access database. In such case, we have created the login table in the access database. There is only one column in the table named name. Let's get all the name of the login table.

1. **import** java.sql.\*;
2. **class** Test{
3. **public** **static** **void** main(String ar[]){
4. **try**{
5. String database="student.mdb";//Here database exists in the current directory
7. String url="jdbc:odbc:Driver={Microsoft Access Driver (\*.mdb)};
8. DBQ=" + database + ";DriverID=22;READONLY=**true**";
10. Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
11. Connection c=DriverManager.getConnection(url);
12. Statement st=c.createStatement();
13. ResultSet rs=st.executeQuery("select \* from login");
15. **while**(rs.next()){
16. System.out.println(rs.getString(1));
17. }
19. }**catch**(Exception ee){System.out.println(ee);}
21. }}

[download this example](https://www.javatpoint.com/src/jdbc/accesswithoutdsn.zip)

### **Example to Connect Java Application with access with DSN**

Connectivity with type1 driver is not considered good. To connect java application with type1 driver, create DSN first, here we are assuming your dsn name is mydsn.

1. **import** java.sql.\*;
2. **class** Test{
3. **public** **static** **void** main(String ar[]){
4. **try**{
5. String url="jdbc:odbc:mydsn";
6. Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
7. Connection c=DriverManager.getConnection(url);
8. Statement st=c.createStatement();
9. ResultSet rs=st.executeQuery("select \* from login");
11. **while**(rs.next()){
12. System.out.println(rs.getString(1));
13. }
15. }**catch**(Exception ee){System.out.println(ee);}
17. }}

# **DriverManager class**

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

### **Commonly used methods of DriverManager class:**

|  |  |
| --- | --- |
| 1) public static void registerDriver(Driver driver): | is used to register the given driver with DriverManager. |
| 2) public static void deregisterDriver(Driver driver): | is used to deregister the given driver (drop the driver from the list) with DriverManager. |
| 3) public static Connection getConnection(String url): | is used to establish the connection with the specified url. |
| 4) public static Connection getConnection(String url,String userName,String password): | is used to establish the connection with the specified url, username and password. |

# **Connection interface**

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.

#### By default, connection commits the changes after executing queries.

### **Commonly used methods of Connection interface:**

|  |
| --- |
| **1) public Statement createStatement():** creates a statement object that can be used to execute SQL queries. |
| **2) public Statement createStatement(int resultSetType,int resultSetConcurrency):** Creates a Statement object that will generate ResultSet objects with the given type and concurrency. |
| **3) public void setAutoCommit(boolean status):** is used to set the commit status.By default it is true. |
| **4) public void commit():** saves the changes made since the previous commit/rollback permanent. |
| **5) public void rollback():** Drops all changes made since the previous commit/rollback. |
| **6) public void close():** closes the connection and Releases a JDBC resources immediately. |

# **Statement interface**

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

### **Commonly used methods of Statement interface:**

The important methods of Statement interface are as follows:

|  |
| --- |
| **1) public ResultSet executeQuery(String sql):** is used to execute SELECT query. It returns the object of ResultSet. |
| **2) public int executeUpdate(String sql):** is used to execute specified query, it may be create, drop, insert, update, delete etc. |
| **3) public boolean execute(String sql):** is used to execute queries that may return multiple results. |
| **4) public int[] executeBatch():** is used to execute batch of commands. |

### **Example of Statement interface**

Let’s see the simple example of Statement interface to insert, update and delete the record.

1. **import** java.sql.\*;
2. **class** FetchRecord{
3. **public** **static** **void** main(String args[])**throws** Exception{
4. Class.forName("oracle.jdbc.driver.OracleDriver");
5. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
6. Statement stmt=con.createStatement();
8. //stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");
9. //int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where id=33");
10. **int** result=stmt.executeUpdate("delete from emp765 where id=33");
11. System.out.println(result+" records affected");
12. con.close();
13. }}

# **ResultSet interface**

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

#### By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE\_SCROLL\_INSENSITIVE or TYPE\_SCROLL\_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

1. Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE,
2. ResultSet.CONCUR\_UPDATABLE);

### **Commonly used methods of ResultSet interface**

|  |  |
| --- | --- |
| **1) public boolean next():** | is used to move the cursor to the one row next from the current position. |
| **2) public boolean previous():** | is used to move the cursor to the one row previous from the current position. |
| **3) public boolean first():** | is used to move the cursor to the first row in result set object. |
| **4) public boolean last():** | is used to move the cursor to the last row in result set object. |
| **5) public boolean absolute(int row):** | is used to move the cursor to the specified row number in the ResultSet object. |
| **6) public boolean relative(int row):** | is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative. |
| **7) public int getInt(int columnIndex):** | is used to return the data of specified column index of the current row as int. |
| **8) public int getInt(String columnName):** | is used to return the data of specified column name of the current row as int. |
| **9) public String getString(int columnIndex):** | is used to return the data of specified column index of the current row as String. |
| **10) public String getString(String columnName):** | is used to return the data of specified column name of the current row as String. |

### **Example of Scrollable ResultSet**

Let’s see the simple example of ResultSet interface to retrieve the data of 3rd row.

1. **import** java.sql.\*;
2. **class** FetchRecord{
3. **public** **static** **void** main(String args[])**throws** Exception{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
7. Statement stmt=con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,ResultSet.CONCUR\_UPDATABLE);
8. ResultSet rs=stmt.executeQuery("select \* from emp765");
10. //getting the record of 3rd row
11. rs.absolute(3);
12. System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));
14. con.close();
15. }}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next>>**](https://www.javatpoint.com/ResultSetMetaData-interface)[**<<prev**](https://www.javatpoint.com/ResultSet-interface) **PreparedStatement interface** The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.  Let's see the example of parameterized query:   1. String sql="insert into emp values(?,?,?)";   As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement. **Why use PreparedStatement?** **Improves performance**: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once. **How to get the instance of PreparedStatement?** The prepareStatement() method of Connection interface is used to return the object of PreparedStatement. Syntax:   1. **public** PreparedStatement prepareStatement(String query)**throws** SQLException{}  **Methods of PreparedStatement interface** The important methods of PreparedStatement interface are given below:   |  |  | | --- | --- | | **Method** | **Description** | | public void setInt(int paramIndex, int value) | sets the integer value to the given parameter index. | | public void setString(int paramIndex, String value) | sets the String value to the given parameter index. | | public void setFloat(int paramIndex, float value) | sets the float value to the given parameter index. | | public void setDouble(int paramIndex, double value) | sets the double value to the given parameter index. | | public int executeUpdate() | executes the query. It is used for create, drop, insert, update, delete etc. | | public ResultSet executeQuery() | executes the select query. It returns an instance of ResultSet. |  **Example of PreparedStatement interface that inserts the record** First of all create table as given below:   1. create table emp(id number(10),name varchar2(50));   Now insert records in this table by the code given below:   1. **import** java.sql.\*; 2. **class** InsertPrepared{ 3. **public** **static** **void** main(String args[]){ 4. **try**{ 5. Class.forName("oracle.jdbc.driver.OracleDriver"); 7. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle"); 9. PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)"); 10. stmt.setInt(1,101);//1 specifies the first parameter in the query 11. stmt.setString(2,"Ratan"); 13. **int** i=stmt.executeUpdate(); 14. System.out.println(i+" records inserted"); 16. con.close(); 18. }**catch**(Exception e){ System.out.println(e);} 20. } 21. }   [download this example](https://www.javatpoint.com/src/jdbc/InsertPrepared.zip) **Example of PreparedStatement interface that updates the record**  1. PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?"); 2. stmt.setString(1,"Sonoo");//1 specifies the first parameter in the query i.e. name 3. stmt.setInt(2,101); 5. **int** i=stmt.executeUpdate(); 6. System.out.println(i+" records updated");   [download this example](https://www.javatpoint.com/src/jdbc/UpdatePrepared.zip) **Example of PreparedStatement interface that deletes the record**  1. PreparedStatement stmt=con.prepareStatement("delete from emp where id=?"); 2. stmt.setInt(1,101); 4. **int** i=stmt.executeUpdate(); 5. System.out.println(i+" records deleted");   [download this example](https://www.javatpoint.com/src/jdbc/DeletePrepared.zip) **Example of PreparedStatement interface that retrieve the records of a table**  1. PreparedStatement stmt=con.prepareStatement("select \* from emp"); 2. ResultSet rs=stmt.executeQuery(); 3. **while**(rs.next()){ 4. System.out.println(rs.getInt(1)+" "+rs.getString(2)); 5. }   [download this example](https://www.javatpoint.com/src/jdbc/RetrievePrepared.zip) **Example of PreparedStatement to insert records until user press n**  1. **import** java.sql.\*; 2. **import** java.io.\*; 3. **class** RS{ 4. **public** **static** **void** main(String args[])**throws** Exception{ 5. Class.forName("oracle.jdbc.driver.OracleDriver"); 6. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle"); 8. PreparedStatement ps=con.prepareStatement("insert into emp130 values(?,?,?)"); 10. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in)); 12. **do**{ 13. System.out.println("enter id:"); 14. **int** id=Integer.parseInt(br.readLine()); 15. System.out.println("enter name:"); 16. String name=br.readLine(); 17. System.out.println("enter salary:"); 18. **float** salary=Float.parseFloat(br.readLine()); 20. ps.setInt(1,id); 21. ps.setString(2,name); 22. ps.setFloat(3,salary); 23. **int** i=ps.executeUpdate(); 24. System.out.println(i+" records affected"); 26. System.out.println("Do you want to continue: y/n"); 27. String s=br.readLine(); 28. **if**(s.startsWith("n")){ 29. **break**; 30. } 31. }**while**(**true**); 33. con.close(); 34. }} |

# **Java ResultSetMetaData Interface**

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

## Commonly used methods of ResultSetMetaData interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| public int getColumnCount()throws SQLException | it returns the total number of columns in the ResultSet object. |
| public String getColumnName(int index)throws SQLException | it returns the column name of the specified column index. |
| public String getColumnTypeName(int index)throws SQLException | it returns the column type name for the specified index. |
| public String getTableName(int index)throws SQLException | it returns the table name for the specified column index. |

### **How to get the object of ResultSetMetaData:**

|  |
| --- |
| The getMetaData() method of ResultSet interface returns the object of ResultSetMetaData. Syntax: |

1. **public** ResultSetMetaData getMetaData()**throws** SQLException

### **Example of ResultSetMetaData interface :**

1. **import** java.sql.\*;
2. **class** Rsmd{
3. **public** **static** **void** main(String args[]){
4. **try**{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection(
7. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9. PreparedStatement ps=con.prepareStatement("select \* from emp");
10. ResultSet rs=ps.executeQuery();
11. ResultSetMetaData rsmd=rs.getMetaData();
13. System.out.println("Total columns: "+rsmd.getColumnCount());
14. System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
15. System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));
17. con.close();
18. }**catch**(Exception e){ System.out.println(e);}
19. }
20. }

# **Java DatabaseMetaData interface**

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

## Commonly used methods of DatabaseMetaData interface

* **public String getDriverName()throws SQLException:**it returns the name of the JDBC driver.
* **public String getDriverVersion()throws SQLException:**it returns the version number of the JDBC driver.
* **public String getUserName()throws SQLException:**it returns the username of the database.
* **public String getDatabaseProductName()throws SQLException:**it returns the product name of the database.
* **public String getDatabaseProductVersion()throws SQLException:**it returns the product version of the database.
* **public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException:**it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

### **How to get the object of DatabaseMetaData:**

The getMetaData() method of Connection interface returns the object of DatabaseMetaData. Syntax:

1. **public** DatabaseMetaData getMetaData()**throws** SQLException

### **Simple Example of DatabaseMetaData interface :**

1. **import** java.sql.\*;
2. **class** Dbmd{
3. **public** **static** **void** main(String args[]){
4. **try**{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9. DatabaseMetaData dbmd=con.getMetaData();
11. System.out.println("Driver Name: "+dbmd.getDriverName());
12. System.out.println("Driver Version: "+dbmd.getDriverVersion());
13. System.out.println("UserName: "+dbmd.getUserName());
14. System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());
15. System.out.println("Database Product Version: "+dbmd.getDatabaseProductVersion());
17. con.close();
18. }**catch**(Exception e){ System.out.println(e);}
19. }
20. }

Output:Driver Name: Oracle JDBC Driver

Driver Version: 10.2.0.1.0XE

Database Product Name: Oracle

Database Product Version: Oracle Database 10g Express Edition

Release 10.2.0.1.0 -Production

[download this example](https://www.javatpoint.com/src/jdbc/Dbmd.java)

### **Example of DatabaseMetaData interface that prints total number of tables :**

1. **import** java.sql.\*;
2. **class** Dbmd2{
3. **public** **static** **void** main(String args[]){
4. **try**{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10. DatabaseMetaData dbmd=con.getMetaData();
11. String table[]={"TABLE"};
12. ResultSet rs=dbmd.getTables(**null**,**null**,**null**,table);
14. **while**(rs.next()){
15. System.out.println(rs.getString(3));
16. }
18. con.close();
20. }**catch**(Exception e){ System.out.println(e);}
22. }
23. }

[download this example](https://www.javatpoint.com/src/jdbc/Dbmd2.java)

### **Example of DatabaseMetaData interface that prints total number of views :**

1. **import** java.sql.\*;
2. **class** Dbmd3{
3. **public** **static** **void** main(String args[]){
4. **try**{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10. DatabaseMetaData dbmd=con.getMetaData();
11. String table[]={"VIEW"};
12. ResultSet rs=dbmd.getTables(**null**,**null**,**null**,table);
14. **while**(rs.next()){
15. System.out.println(rs.getString(3));
16. }
18. con.close();
20. }**catch**(Exception e){ System.out.println(e);}
22. }
23. }

[**next →**](https://www.javatpoint.com/retrieving-image-from-oracle-database)[**← prev**](https://www.javatpoint.com/DatabaseMetaData-interface)

# **Example to store image in Oracle database**

You can store images in the database in java by the help of **PreparedStatement** interface.

The **setBinaryStream()** method of PreparedStatement is used to set Binary information into the parameterIndex.

### **Signature of setBinaryStream method**

The syntax of setBinaryStream() method is given below:

1. 1) **public** **void** setBinaryStream(**int** paramIndex,InputStream stream)
2. **throws** SQLException
3. 2) **public** **void** setBinaryStream(**int** paramIndex,InputStream stream,**long** length)
4. **throws** SQLException

For storing image into the database, BLOB (Binary Large Object) datatype is used in the table. For example:

1. CREATE TABLE  "IMGTABLE"
2. (    "NAME" VARCHAR2(4000),
3. "PHOTO" BLOB
4. )
5. /

Let's write the jdbc code to store the image in the database. Here we are using d:\\d.jpg for the location of image. You can change it according to the image location.

## Java Example to store image in the database

1. **import** java.sql.\*;
2. **import** java.io.\*;
3. **public** **class** InsertImage {
4. **public** **static** **void** main(String[] args) {
5. **try**{
6. Class.forName("oracle.jdbc.driver.OracleDriver");
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10. PreparedStatement ps=con.prepareStatement("insert into imgtable values(?,?)");
11. ps.setString(1,"sonoo");
13. FileInputStream fin=**new** FileInputStream("d:\\g.jpg");
14. ps.setBinaryStream(2,fin,fin.available());
15. **int** i=ps.executeUpdate();
16. System.out.println(i+" records affected");
18. con.close();
19. }**catch** (Exception e) {e.printStackTrace();}
20. }
21. }

If you see the table, record is stored in the database but image will not be shown. To do so, you need to retrieve the image from the database which we are covering in the next page.

# **Example to retrieve image from Oracle database**

By the help of **PreparedStatement** we can retrieve and store the image in the database.

The **getBlob()** method of PreparedStatement is used to get Binary information, it returns the instance of Blob. After calling the **getBytes()** method on the blob object, we can get the array of binary information that can be written into the image file.

### **Signature of getBlob() method of PreparedStatement**

1. **public** Blob getBlob()**throws** SQLException

### **Signature of getBytes() method of Blob interface**

1. **public**  **byte**[] getBytes(**long** pos, **int** length)**throws** SQLException

We are assuming that image is stored in the imgtable.

1. CREATE TABLE  "IMGTABLE"
2. (    "NAME" VARCHAR2(4000),
3. "PHOTO" BLOB
4. )
5. /

Now let's write the code to retrieve the image from the database and write it into the directory so that it can be displayed.

In AWT, it can be displayed by the Toolkit class. In servlet, jsp, or html it can be displayed by the img tag.

1. **import** java.sql.\*;
2. **import** java.io.\*;
3. **public** **class** RetrieveImage {
4. **public** **static** **void** main(String[] args) {
5. **try**{
6. Class.forName("oracle.jdbc.driver.OracleDriver");
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10. PreparedStatement ps=con.prepareStatement("select \* from imgtable");
11. ResultSet rs=ps.executeQuery();
12. **if**(rs.next()){//now on 1st row
14. Blob b=rs.getBlob(2);//2 means 2nd column data
15. **byte** barr[]=b.getBytes(1,(**int**)b.length());//1 means first image
17. FileOutputStream fout=**new** FileOutputStream("d:\\sonoo.jpg");
18. fout.write(barr);
20. fout.close();
21. }//end of if
22. System.out.println("ok");
24. con.close();
25. }**catch** (Exception e) {e.printStackTrace();  }
26. }
27. }

Now if you see the d drive, sonoo.jpg image is created.

# **Example to store file in Oracle database:**

The setCharacterStream() method of PreparedStatement is used to set character information into the parameterIndex.

### **Syntax:**

|  |
| --- |
| 1) public void setBinaryStream(int paramIndex,InputStream stream)throws SQLException |
| 2) public void setBinaryStream(int paramIndex,InputStream stream,long length)throws SQLException |

For storing file into the database, CLOB (Character Large Object) datatype is used in the table. For example:

1. CREATE TABLE  "FILETABLE"
2. (    "ID" NUMBER,
3. "NAME" CLOB
4. )
5. /

## Java Example to store file in database

1. **import** java.io.\*;
2. **import** java.sql.\*;
4. **public** **class** StoreFile {
5. **public** **static** **void** main(String[] args) {
6. **try**{
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection(
9. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
11. PreparedStatement ps=con.prepareStatement(
12. "insert into filetable values(?,?)");
14. File f=**new** File("d:\\myfile.txt");
15. FileReader fr=**new** FileReader(f);
17. ps.setInt(1,101);
18. ps.setCharacterStream(2,fr,(**int**)f.length());
19. **int** i=ps.executeUpdate();
20. System.out.println(i+" records affected");
22. con.close();
24. }**catch** (Exception e) {e.printStackTrace();}
25. }
26. }

# **Example to retrieve file from Oracle database:**

The getClob() method of PreparedStatement is used to get file information from the database.

### **Syntax of getClob method**

1. **public** Clob getClob(**int** columnIndex){}

Let's see the table structure of this example to retrieve the file.

1. CREATE TABLE  "FILETABLE"
2. (    "ID" NUMBER,
3. "NAME" CLOB
4. )
5. /

The example to retrieve the file from the Oracle database is given below.

1. **import** java.io.\*;
2. **import** java.sql.\*;
4. **public** **class** RetrieveFile {
5. **public** **static** **void** main(String[] args) {
6. **try**{
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection(
9. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
11. PreparedStatement ps=con.prepareStatement("select \* from filetable");
12. ResultSet rs=ps.executeQuery();
13. rs.next();//now on 1st row
15. Clob c=rs.getClob(2);
16. Reader r=c.getCharacterStream();
18. FileWriter fw=**new** FileWriter("d:\\retrivefile.txt");
20. **int** i;
21. **while**((i=r.read())!=-1)
22. fw.write((**char**)i);
24. fw.close();
25. con.close();
27. System.out.println("success");
28. }**catch** (Exception e) {e.printStackTrace();  }
29. }
30. }

# **Java CallableStatement Interface**

CallableStatement interface is used to call the **stored procedures and functions**.

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

### **What is the difference between stored procedures and functions.**

The differences between stored procedures and functions are given below:

|  |  |
| --- | --- |
| **Stored Procedure** | **Function** |
| is used to perform business logic. | is used to perform calculation. |
| must not have the return type. | must have the return type. |
| may return 0 or more values. | may return only one values. |
| We can call functions from the procedure. | Procedure cannot be called from function. |
| Procedure supports input and output parameters. | Function supports only input parameter. |
| Exception handling using try/catch block can be used in stored procedures. | Exception handling using try/catch can't be used in user defined functions. |

### **How to get the instance of CallableStatement?**

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

1. **public** CallableStatement prepareCall("{ call procedurename(?,?...?)}");

The example to get the instance of CallableStatement is given below:

1. CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");

It calls the procedure myprocedure that receives 2 arguments.

### **Full example to call the stored procedure using JDBC**

To call the stored procedure, you need to create it in the database. Here, we are assuming that stored procedure looks like this.

1. create or replace procedure "INSERTR"
2. (id IN NUMBER,
3. name IN VARCHAR2)
4. is
5. begin
6. insert into user420 values(id,name);
7. end;
8. /

The table structure is given below:

1. create table user420(id number(10), name varchar2(200));

In this example, we are going to call the stored procedure INSERTR that receives id and name as the parameter and inserts it into the table user420. Note that you need to create the user420 table as well to run this application.

1. **import** java.sql.\*;
2. **public** **class** Proc {
3. **public** **static** **void** main(String[] args) **throws** Exception{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection(
7. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9. CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");
10. stmt.setInt(1,1011);
11. stmt.setString(2,"Amit");
12. stmt.execute();
14. System.out.println("success");
15. }
16. }

Now check the table in the database, value is inserted in the user420 table.

### **Example to call the function using JDBC**

In this example, we are calling the sum4 function that receives two input and returns the sum of the given number. Here, we have used the **registerOutParameter** method of CallableStatement interface, that registers the output parameter with its corresponding type. It provides information to the CallableStatement about the type of result being displayed.

The **Types** class defines many constants such as INTEGER, VARCHAR, FLOAT, DOUBLE, BLOB, CLOB etc.

Let's create the simple function in the database first.

1. create or replace function sum4
2. (n1 in number,n2 in number)
3. **return** number
4. is
5. temp number(8);
6. begin
7. temp :=n1+n2;
8. **return** temp;
9. end;
10. /

Now, let's write the simple program to call the function.

1. **import** java.sql.\*;
3. **public** **class** FuncSum {
4. **public** **static** **void** main(String[] args) **throws** Exception{
6. Class.forName("oracle.jdbc.driver.OracleDriver");
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10. CallableStatement stmt=con.prepareCall("{?= call sum4(?,?)}");
11. stmt.setInt(2,10);
12. stmt.setInt(3,43);
13. stmt.registerOutParameter(1,Types.INTEGER);
14. stmt.execute();
16. System.out.println(stmt.getInt(1));
18. }
19. }

# **Transaction Management in JDBC**

Transaction represents **a single unit of work**.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

**Atomicity** means either all successful or none.

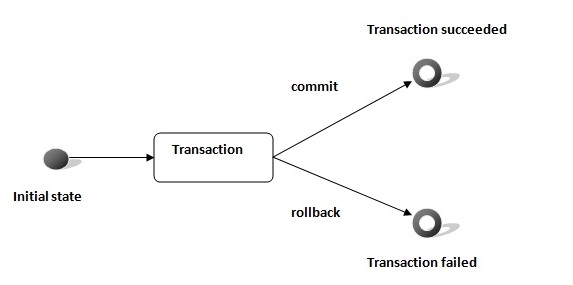
**Consistency** ensures bringing the database from one consistent state to another consistent state.

**Isolation** ensures that transaction is isolated from other transaction.

**Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

#### **Advantage of Transaction Mangaement**

**fast performance** It makes the performance fast because database is hit at the time of commit.



In JDBC, **Connection interface** provides methods to manage transaction.

|  |  |
| --- | --- |
| **Method** | **Description** |
| void setAutoCommit(boolean status) | It is true bydefault means each transaction is committed bydefault. |
| void commit() | commits the transaction. |
| void rollback() | cancels the transaction. |

### **Simple example of transaction management in jdbc using Statement**

Let's see the simple example of transaction management using Statement.

1. **import** java.sql.\*;
2. **class** FetchRecords{
3. **public** **static** **void** main(String args[])**throws** Exception{
4. Class.forName("oracle.jdbc.driver.OracleDriver");
5. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
6. con.setAutoCommit(**false**);
8. Statement stmt=con.createStatement();
9. stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
10. stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
12. con.commit();
13. con.close();
14. }}

If you see the table emp400, you will see that 2 records has been added.

### **Example of transaction management in jdbc using PreparedStatement**

Let's see the simple example of transaction management using PreparedStatement.

1. **import** java.sql.\*;
2. **import** java.io.\*;
3. **class** TM{
4. **public** **static** **void** main(String args[]){
5. **try**{
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9. con.setAutoCommit(**false**);
11. PreparedStatement ps=con.prepareStatement("insert into user420 values(?,?,?)");
13. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
14. **while**(**true**){
16. System.out.println("enter id");
17. String s1=br.readLine();
18. **int** id=Integer.parseInt(s1);
20. System.out.println("enter name");
21. String name=br.readLine();
23. System.out.println("enter salary");
24. String s3=br.readLine();
25. **int** salary=Integer.parseInt(s3);
27. ps.setInt(1,id);
28. ps.setString(2,name);
29. ps.setInt(3,salary);
30. ps.executeUpdate();
32. System.out.println("commit/rollback");
33. String answer=br.readLine();
34. **if**(answer.equals("commit")){
35. con.commit();
36. }
37. **if**(answer.equals("rollback")){
38. con.rollback();
39. }

42. System.out.println("Want to add more records y/n");
43. String ans=br.readLine();
44. **if**(ans.equals("n")){
45. **break**;
46. }
48. }
49. con.commit();
50. System.out.println("record successfully saved");
52. con.close();//before closing connection commit() is called
53. }**catch**(Exception e){System.out.println(e);}
55. }}

It will ask to add more records until you press n. If you press n, transaction is committed.

# **Batch Processing in JDBC**

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.

The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

#### **Advantage of Batch Processing**

Fast Performance

#### **Methods of Statement interface**

The required methods for batch processing are given below:

|  |  |
| --- | --- |
| **Method** | **Description** |
| void addBatch(String query) | It adds query into batch. |
| int[] executeBatch() | It executes the batch of queries. |

### **Example of batch processing in jdbc**

Let's see the simple example of batch processing in jdbc. It follows following steps:

* Load the driver class
* Create Connection
* Create Statement
* Add query in the batch
* Execute Batch
* Close Connection

1. **import** java.sql.\*;
2. **class** FetchRecords{
3. **public** **static** **void** main(String args[])**throws** Exception{
4. Class.forName("oracle.jdbc.driver.OracleDriver");
5. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
6. con.setAutoCommit(**false**);
8. Statement stmt=con.createStatement();
9. stmt.addBatch("insert into user420 values(190,'abhi',40000)");
10. stmt.addBatch("insert into user420 values(191,'umesh',50000)");
12. stmt.executeBatch();//executing the batch
14. con.commit();
15. con.close();
16. }}

If you see the table user420, two records has been added.

### **Example of batch processing using PreparedStatement**

1. **import** java.sql.\*;
2. **import** java.io.\*;
3. **class** BP{
4. **public** **static** **void** main(String args[]){
5. **try**{
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10. PreparedStatement ps=con.prepareStatement("insert into user420 values(?,?,?)");
12. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
13. **while**(**true**){
15. System.out.println("enter id");
16. String s1=br.readLine();
17. **int** id=Integer.parseInt(s1);
19. System.out.println("enter name");
20. String name=br.readLine();
22. System.out.println("enter salary");
23. String s3=br.readLine();
24. **int** salary=Integer.parseInt(s3);
26. ps.setInt(1,id);
27. ps.setString(2,name);
28. ps.setInt(3,salary);
30. ps.addBatch();
31. System.out.println("Want to add more records y/n");
32. String ans=br.readLine();
33. **if**(ans.equals("n")){
34. **break**;
35. }
37. }
38. ps.executeBatch();
40. System.out.println("record successfully saved");
42. con.close();
43. }**catch**(Exception e){System.out.println(e);}
45. }}

It will add the queries into the batch until user press n. Finally it executes the batch. Thus all the added queries will be fired.

# **JDBC RowSet**

The instance of **RowSet** is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.

It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use.

The implementation classes of RowSet interface are as follows:

* JdbcRowSet
* CachedRowSet
* WebRowSet
* JoinRowSet
* FilteredRowSet

Let's see how to create and execute RowSet.

1. JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
2. rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
3. rowSet.setUsername("system");
4. rowSet.setPassword("oracle");
6. rowSet.setCommand("select \* from emp400");
7. rowSet.execute();

#### It is the new way to get the instance of JdbcRowSet since JDK 7.

#### **Advantage of RowSet**

The advantages of using RowSet are given below:

1. It is easy and flexible to use
2. It is Scrollable and Updatable bydefault

### **Simple example of JdbcRowSet**

Let's see the simple example of JdbcRowSet without event handling code.

1. **import** java.sql.Connection;
2. **import** java.sql.DriverManager;
3. **import** java.sql.ResultSet;
4. **import** java.sql.Statement;
5. **import** javax.sql.RowSetEvent;
6. **import** javax.sql.RowSetListener;
7. **import** javax.sql.rowset.JdbcRowSet;
8. **import** javax.sql.rowset.RowSetProvider;
10. **public** **class** RowSetExample {
11. **public** **static** **void** main(String[] args) **throws** Exception {
12. Class.forName("oracle.jdbc.driver.OracleDriver");
14. //Creating and Executing RowSet
15. JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
16. rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
17. rowSet.setUsername("system");
18. rowSet.setPassword("oracle");
20. rowSet.setCommand("select \* from emp400");
21. rowSet.execute();
23. **while** (rowSet.next()) {
24. // Generating cursor Moved event
25. System.out.println("Id: " + rowSet.getString(1));
26. System.out.println("Name: " + rowSet.getString(2));
27. System.out.println("Salary: " + rowSet.getString(3));
28. }
30. }
31. }

The output is given below:

Id: 55

Name: Om Bhim

Salary: 70000

Id: 190

Name: abhi

Salary: 40000

Id: 191

Name: umesh

Salary: 50000

### **Full example of Jdbc RowSet with event handling**

To perform event handling with JdbcRowSet, you need to add the instance of **RowSetListener** in the addRowSetListener method of JdbcRowSet.

The RowSetListener interface provides 3 method that must be implemented. They are as follows:

1) public void cursorMoved(RowSetEvent event);

2) public void rowChanged(RowSetEvent event);

3) public void rowSetChanged(RowSetEvent event);

Let's write the code to retrieve the data and perform some additional tasks while cursor is moved, cursor is changed or rowset is changed. The event handling operation can't be performed using ResultSet so it is preferred now.

1. **import** java.sql.Connection;
2. **import** java.sql.DriverManager;
3. **import** java.sql.ResultSet;
4. **import** java.sql.Statement;
5. **import** javax.sql.RowSetEvent;
6. **import** javax.sql.RowSetListener;
7. **import** javax.sql.rowset.JdbcRowSet;
8. **import** javax.sql.rowset.RowSetProvider;
10. **public** **class** RowSetExample {
11. **public** **static** **void** main(String[] args) **throws** Exception {
12. Class.forName("oracle.jdbc.driver.OracleDriver");
14. //Creating and Executing RowSet
15. JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
16. rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
17. rowSet.setUsername("system");
18. rowSet.setPassword("oracle");
20. rowSet.setCommand("select \* from emp400");
21. rowSet.execute();
23. //Adding Listener and moving RowSet
24. rowSet.addRowSetListener(**new** MyListener());
26. **while** (rowSet.next()) {
27. // Generating cursor Moved event
28. System.out.println("Id: " + rowSet.getString(1));
29. System.out.println("Name: " + rowSet.getString(2));
30. System.out.println("Salary: " + rowSet.getString(3));
31. }
33. }
34. }
36. **class** MyListener **implements** RowSetListener {
37. **public** **void** cursorMoved(RowSetEvent event) {
38. System.out.println("Cursor Moved...");
39. }
40. **public** **void** rowChanged(RowSetEvent event) {
41. System.out.println("Cursor Changed...");
42. }
43. **public** **void** rowSetChanged(RowSetEvent event) {
44. System.out.println("RowSet changed...");
45. }
46. }

The output is as follows:

Cursor Moved...

Id: 55

Name: Om Bhim

Salary: 70000

Cursor Moved...

Id: 190

Name: abhi

Salary: 40000

Cursor Moved...

Id: 191

Name: umesh

Salary: 50000

Cursor Moved...

# **Jdbc New Features**

The latest version of JDBC is 4.0 currently. Java has updated jdbc api to ease and simplify the coding to database interactivity.

Here, we are going to see the features included in Jdbc 3.0 and Jdbc 4.0.

## Jdbc 3.0 Features

The important features of JDBC API 3.0 are as follows:

* **Jdbc RowSet** We have done the great discussion on JdbcRowSet in the previous page.
* **Savepoint in transaction management** Now you are able to create, rollback and release the savepoint by Connection.setSavepoint(), Connection.rollback(Savepoint svpt) and Connection.releaseSavepoint(Savepoint svpt) methods.
* **Statement and ResultSet Caching for Connection Pooling** Now you are able to reuse the statement and result set because jdbc 3 provides you the facility of statement caching and result set caching.
* **Switching between Global and Local Transactions**
* **Retrieval of auto generated keys** Now you are able to get the auto generated keys by the method getGeneratedKeys().

## Jdbc 4.0 Features

The important features of JDBC API 4.0 are given below:

* **Automatic Loading of Driver class** You don't need to write Class.forName() now because it is loaded bydefault since jdbc4.
* **Subclasses of SQLException** Jdbc 4 provides new subclasses of SQLException class for better readability and handling.
* **New methods** There are many new methods introduced in Connection, PreparedStatement, CallableStatement, ResultSet etc.
* **Improved DataSource** Now data source implementation is improved.
* **Event Handling support in Statement for Connection Pooling** Now Connection Pooling can listen statement error and statement closing events.

# **Java 8 Features**

Oracle released a new version of Java as Java 8 in March 18, 2014. It was a revolutionary release of the Java for software development platform. It includes various upgrades to the Java programming, JVM, Tools and libraries.

## Java 8 Programming Language Enhancements

Java 8 provides following features for Java Programming:

* Lambda expressions,
* Method references,
* Functional interfaces,
* Stream API,
* Default methods,
* Base64 Encode Decode,
* Static methods in interface,
* Optional class,
* Collectors class,
* ForEach() method,
* Parallel array sorting,
* Nashorn JavaScript Engine,
* Parallel Array Sorting,
* Type and Repating Annotations,
* IO Enhancements,
* Concurrency Enhancements,
* JDBC Enhancements etc.

## Lambda Expressions

Lambda expression helps us to write our code in functional style. It provides a clear and concise way to implement SAM interface(Single Abstract Method) by using an expression. It is very useful in collection library in which it helps to iterate, filter and extract data.

For more information and examples: [click here](http://www.javatpoint.com/java-lambda-expressions)

## Method References

Java 8 Method reference is used to refer method of functional interface . It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

For more information and examples: [click here](https://www.javatpoint.com/java-8-method-reference)

## Functional Interface

An Interface that contains only one abstract method is known as functional interface. It can have any number of default and static methods. It can also declare methods of object class.

Functional interfaces are also known as Single Abstract Method Interfaces (SAM Interfaces).

For more information and examples: [click here](http://www.javatpoint.com/java-8-functional-interfaces)

## Optional

Java introduced a new class Optional in Java 8. It is a public final class which is used to deal with NullPointerException in Java application. We must import java.util package to use this class. It provides methods to check the presence of value for particular variable.

For more information and examples: [click here](http://www.javatpoint.com/java-8-optional)

## forEach

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interfaces.

It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach() method to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

For more information and examples: [click here](https://www.javatpoint.com/java-8-features)

## Date/Time API

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

For more information and examples: [click here](http://www.javatpoint.com/java-date)

## Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default keyword are known as default methods. These methods are non-abstract methods and can have method body.

For more information and examples: [click here](http://www.javatpoint.com/java-default-methods)

## Nashorn JavaScript Engine

Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool **jjs** which is used to execute JavaScript code.

You can execute JavaScript code by two ways:

1. Using jjs command-line tool, and
2. By embedding into Java source code.

For more information and examples: [click here](https://www.javatpoint.com/java-nashorn)

## StringJoiner

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc.

For more information and examples: [click here](https://www.javatpoint.com/java-stringjoiner)

## Collectors

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria etc.

For more information and examples: [click here](https://www.javatpoint.com/java-8-collectors)

## Stream API

Java 8 java.util.stream package consists of classes, interfaces and an enum to allow functional-style operations on the elements. It performs lazy computation. So, it executes only when it requires.

For more information and examples: [click here](http://www.javatpoint.com/java-8-stream)

## Stream Filter

Java stream provides a method filter() to filter stream elements on the basis of given predicate. Suppose, you want to get only even elements of your list, you can do this easily with the help of filter() method.

This method takes predicate as an argument and returns a stream of resulted elements.

For more information and examples: [click here](https://www.javatpoint.com/java-8-stream-filter)

## Java Base64 Encoding and Decoding

Java provides a class Base64 to deal with encryption and decryption. You need to import java.util.Base64 class in your source file to use its methods.

This class provides three different encoders and decoders to encrypt information at each level.

For more information and examples: [click here](https://www.javatpoint.com/java-base64-encode-decode)

## Java Parallel Array Sorting

Java provides a new additional feature in Arrays class which is used to sort array elements parallelly. The parallelSort() method has added to java.util.Arrays class that uses the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays. It is an overloaded method.

For more information and examples: [click here](https://www.javatpoint.com/java-8-parallel-array-sorting)

## Java 8 Security Enhancements

1) The Java Secure Socket Extension(JSSE) provider enables the protocols Transport Layer Security (TLS) 1.1 and TLS 1.2 by default on the client side.

2) A improved method AccessController.doPrivileged has been added which enables code to assert a subset of its privileges, without preventing the full traversal of the stack to check for other permissions.

3) Advanced Encryption Standard (AES) and Password-Based Encryption (PBE) algorithms, such as PBEWithSHA256AndAES\_128 and PBEWithSHA512AndAES\_256 has been added to the SunJCE provider.

4) Java Secure Socket Extension (SunJSSE) has enabled Server Name Indication (SNI) extension for client applications by default in JDK 7 and JDK 8 supports the SNI extension for server applications. The SNI extension is a feature that extends the SSL/TLS protocols to indicate what server name the client is attempting to connect to during handshaking.

5) The SunJSSE is enhanced to support Authenticated Encryption with Associated Data (AEAD) algorithms. The Java Cryptography Extension (SunJCE) provider is enhanced to support AES/GCM/NoPadding cipher implementation as well as Galois/Counter Mode (GCM) algorithm parameters.

6) A new command flag -importpassword is added to the keytool utility. It is used to accept a password and store it securely as a secret key. Classes such as java.security.DomainLoadStoreParameter andjava.security.PKCS12Attribute is added to support DKS keystore type.

7) In JDK 8, the cryptographic algorithms have been enhanced with the SHA-224 variant of the SHA-2 family of message-digest implementations.

8) Enhanced support for NSA Suite B Cryptography which includes:

* OID registration for NSA Suite B cryptography algorithms
* Support for 2048-bit DSA key pair generation and additional signature algorithms for 2048-bit DSA keys such as SHA224withDSA and SHA256withDSA.
* Lifting of the keysize restriction from 1024 to 2048 for Diffie-Hellman (DH) algorithm.

9) SecureRandom class provides the generation of cryptographically strong random numbers which is used for private or public keys, ciphers and signed messages. The getInstanceStrong() method was introduced in JDK 8, which returns an instance of the strongest SecureRandom. It should be used when you need to create RSA private and public key. SecureRandom includes following other changes:

* Two new implementations has introduced for UNIX platforms, which provide blocking and non-blocking behavior.

10) A new PKIXRevocationChecker class is included which checks the revocation status of certificates with the PKIX algorithm. It supports best effort checking, end-entity certificate checking, and mechanism-specific options.

11) The Public Key Cryptography Standards 11 (PKCS) has been expanded to include 64-bit supports for Windows.

12) Two new rcache types are added to Kerberos 5. Type none means no rcache at all, and type dfl means the DFL style file-based rcache. Also, the acceptor requested subkey is now supported. They are configured using the sun.security.krb5.rcache and sun.security.krb5.acceptor.subkey system properties.

13) In JDK 8, Kerberos 5 protocol transition and constrained delegation are supported within the same realm.

14) Java 8 has disabled weak encryption by default. The DES-related Kerberos 5 encryption types are not supported by default. These encryption types can be enabled by adding allow\_weak\_crypto=true in the krb5.conf file.

15) You can set server name to null to denote an unbound server. It means a client can request for the service using any server name. After a context is established, the server can retrieve the name as a negotiated property with the key name SASL.BOUND\_SERVER\_NAME.

16) Java Native Interface (JNI) bridge to native Java Generic Security Service (JGSS) is now supported on Mac OS X. You can set system property sun.security.jgss.native to true to enable it.

17) A new system property, jdk.tls.ephemeralDHKeySize is defined to customize the ephemeral DH key sizes. The minimum acceptable DH key size is 1024 bits, except for exportable cipher suites or legacy mode (jdk.tls.ephemeralDHKeySize=legacy).

18) Java Secure Socket Extension (JSSE) provider honors the client's cipher suite preference by default. However, the behavior can be changed to respect the server's cipher suite preference by calling SSLParameters.setUseCipherSuitesOrder(true) over the server.

## Java 8 Tools Enhancements

1) A jjs command is introduced, which invokes the Nashorn engine either in interactive shell mode, or to interpret script files.

2) The java command is capable of launching JavaFX applications, provided that the JavaFX application is packaged correctly.

3) The java command man page (both nroff and HTML) has been completely reworked. The advanced options are now divided into Runtime, Compiler, Garbage Collection, and Serviceability, according to the area that they affect. Several previously missing options are now described. There is also a section for options that were deprecated or removed since the previous release.

4) New jdeps command-line tool allows the developer to analyze class files to determine package-level or class-level dependencies.

5) You can access diagnostic commands remotely, which were previously accessible only locally via the jcmd tool. Remote access is provided using the Java Management Extensions (JMX), so diagnostic commands are exposed to a platform MBean registered to the platform MBean server. The MBean is the com.sun.management.DiagnosticCommandMBean interface.

6) A new option -tsapolicyid is included in the jarsigner tool which enables you to request a signed time stamp from a Time Stamping Authority and attach it to a signed JAR file.

7) A new method java.lang.reflect.Executable.getParameters is included which allows you to access the names of the formal parameters of any method or constructor. However, .class files do not store formal parameter names by default. To store formal parameter names in a particular .class file, and thus enable the Reflection API to retrieve formal parameter names, compile the source file with the -parameters option of the javac compiler.

8) The type rules for binary comparisons in the Java Language Specification (JLS) Section 15.21 will now be correctly enforced by javac.

9) In this release, the apt tool and its associated API contained in the package com.sun.mirror have been removed.

## Javadoc Enhancements

In Java SE 8, the following new APIs were added to the Javadoc tool.

* A new DocTree API introduce a scanner which enables you to traverse source code that is represented by an abstract syntax tree. This extends the Compiler Tree API to provide structured access to the content of javadoc comments.
* The javax.tools package contains classes and interfaces that enable you to invoke the Javadoc tool directly from a Java application, without executing a new process.
* The "Method Summary" section of the generated documentation of a class or interface has been restructured. Method descriptions in this section are grouped by type. By default, all methods are listed. You can click a tab to view methods of a particular type (static, instance, abstract, concrete, or deprecated, if they exist in the class or interface).
* The javadoc tool now has support for checking the content of javadoc comments for issues that could lead to various problems, such as invalid HTML or accessibility issues, in the files that are generated by javadoc. The feature is enabled by default, and can also be controlled by the new -Xdoclint option.

## Pack200 Enhancements

The Java class file format has been updated because of JSR 292 which Supports Dynamically Typed Languages on the Java Platform.

The Pack200 engine has been updated to ensure that Java SE 8 class files are compressed effectively. Now, it can recognize constant pool entries and new bytecodes introduced by JSR 292. As a result, compressed files created with this version of the pack200 tool will not be compatible with older versions of the unpack200 tool.

## Java 8 I/O Enhancements

In Java 8, there are several improvements to the java.nio.charset.Charset and extended charset implementations. It includes the following:

* A New SelectorProvider which may improve performance or scalability for server. The /dev/poll SelectorProvider continues to be the default. To use the Solaris event port mechanism, run with the system property java.nio.channels.spi.Selector set to the value sun.nio.ch.EventPortSelectorProvider.
* The size of <JDK\_HOME>/jre/lib/charsets.jar file is decreased.
* Performance has been improvement for the java.lang.String(byte[], ∗) constructor and the java.lang.String.getBytes() method.

## Java 8 Networking Enhancements

1) A new class java.net.URLPermission has been added. It represents a permission for accessing a resource defined by a given URL.

2) A package jdk.net has been added which contains platform specific socket options and a mechanism for setting these options on all of the standard socket types. The socket options are defined in jdk.net.ExtendedSocketOptions.

3) In class HttpURLConnection, if a security manager is installed, and if a method is called which results in an attempt to open a connection, the caller must possess either a "connect"SocketPermission to the host/port combination of the destination URL or a URLPermission that permits this request.

If automatic redirection is enabled, and this request is redirected to another destination, the caller must also have permission to connect to the redirected host/URL.

## Java 8 Concurrency Enhancements

The java.util.concurrent package added two new interfaces and four new classes.

### **Java.util.concurrent Interfaces**

|  |  |
| --- | --- |
| **Interface** | **Description** |
| public static interface CompletableFuture.AsynchronousCompletionTask | It is a marker interface which is used to identify asynchronous tasks produced by async methods. It may be useful for monitoring, debugging, and tracking asynchronous activities. |
| public interface CompletionStage<T> | It creates a stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes. |

### **Java.util.concurrent Classes**

|  |  |
| --- | --- |
| **Class** | **Description** |
| public class CompletableFuture<T> extends Object implements Future<T>, CompletionStage<T> | It is aFuture that may be explicitly completed, and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion. |
| public static class ConcurrentHashMap.KeySetView<K,V> extends Object implements Set<K>, Serializable | It is a view of a ConcurrentHashMap as a Set of keys, in which additions may optionally be enabled by mapping to a common value. |
| public abstract class CountedCompleter<T> extends ForkJoinTask<T> | A ForkJoinTask with a completion action performed when triggered and there are no remaining pending actions. |
| public class CompletionException extends RuntimeException | It throws an exception when an error or other exception is encountered in the course of completing a result or task. |

#### **New Methods in java.util.concurrent.ConcurrentHashMap class**

ConcurrentHashMap class introduces several new methods in its latest release. It includes various forEach methods (forEach, forEachKey, forEachValue, and forEachEntry), search methods (search, searchKeys, searchValues, and searchEntries) and a large number of reduction methods (reduce, reduceToDouble, reduceToLong etc.). Other miscellaneous methods (mappingCount and newKeySet) have been added as well.

#### **New classes in java.util.concurrent.atomic**

Latest release introduces scalable, updatable, variable support through a small set of new classes DoubleAccumulator, DoubleAdder, LongAccumulator andLongAdder. It internally employ contention-reduction techniques that provide huge throughput improvements as compared to Atomic variables.

|  |  |
| --- | --- |
| **Class** | **Description** |
| public class DoubleAccumulator extends Number implements Serializable | It is used for one or more variables that together maintain a running double value updated using a supplied function. |
| public class DoubleAdder extends Number implements Serializable | It is used for one or more variables that together maintain an initially zero double sum. |
| public class LongAccumulator extends Number implements Serializable | It is used for one or more variables that together maintain a running long value updated using a supplied function. |
| public class LongAdder extends Number implements Serializable | It is used for one or more variables that together maintain an initially zero long sum. |

### **New methods in java.util.concurrent.ForkJoinPool Class**

This class has added two new methods getCommonPoolParallelism() and commonPool(), which return the targeted parallelism level of the common pool, or the common pool instance, respectively.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static ForkJoinPool commonPool() | It returns the common pool instance. |
| Public static int getCommonPoolParallelism() | It returns the targeted parallelism level of the common pool. |

### **New class java.util.concurrent.locks.StampedLock**

A new class StampedLock is added which is used to add capability-based lock with three modes for controlling read/write access (writing, reading, and optimistic reading). This class also supports methods that conditionally provide conversions across the three modes.

|  |  |
| --- | --- |
| **Class** | **Description** |
| public class StampedLock extends Object implements Serializable | This class represents a capability-based lock with three modes for controlling read/write access. |

## Java API for XML Processing (JAXP) 1.6 Enhancements

In Java 8, Java API is added for XML Processing (JAXP) 1.6. It requires the use of the service provider loader facility which is defined by java.util.ServiceLoader to load services from service configuration files.

The rationale for this is to allow for future modularization of the Java SE platform where service providers may be deployed by means other than JAR files and perhaps without the service configuration files.

## Java Virtual Machine Enhancements

The verification of invokespecial instructions has been tightened so that only an instance initialization method in the current class or its direct super class may be invoked.

## Java Mission Control 5.3 is included in Java 8

Java Mission Control (JMC) is an advanced set of tools that enables efficient and detailed data analysis and delivers advanced, unobtrusive Java monitoring and management. JMC provides sections for common analysis areas such as code performance, memory and latency.

Babel Language Packs in Japanese and Simplified Chinese are now included by default in the Java Mission Control that is included in the JDK 8.

## Java 8 Internationalization Enhancements

### **1) Unicode Enhancements**

The JDK 8 includes support for Unicode 6.2.0. It contains the following features.

* 733 new characters including Turkish Lira sign.
* 7 new scripts:
  + Meroitic Hieroglyphs
  + Meroitic Cursive
  + Sora Sompeng
  + Chakma
  + Sharada
  + Takri
  + Miao
* 11 new blocks: including 7 blocks for the new scripts listed above and 4 blocks for the following existing scripts:
* Arabic Extended-A
* Sundanese Supplement
* Meetei Mayek Extensions
* Arabic Mathematical Alphabetical Symbols

### **Adoption of Unicode CLDR Data and the java.locale.providers System Property**

The Unicode Consortium has released the Common Locale Data Repository (CLDR) project to "support the world's languages, with the largest and most extensive standard repository of locale data available." The CLDR is becoming the de-facto standard for locale data. The CLDR's XML-based locale data has been incorporated into the JDK 8 release, however it is disabled by default.

There are four distinct sources for locale data:

* CLDR represents the locale data provided by the Unicode CLDR project.
* HOST represents the current user's customization of the underlying operating system's settings. It works only with the user's default locale, and the customizable settings may vary depending on the OS, but primarily Date, Time, Number, and Currency formats are supported.
* SPI represents the locale sensitive services implemented in the installed SPI providers.
* JRE represents the locale data that is compatible with the prior JRE releases.

To select the desired locale data source, use the java.locale.providers system property. listing the data sources in the preferred order. For example: java.locale.providers=HOST,SPI,CLDR,JRE The default behavior is equivalent to the following setting: java.locale.providers=JRE,SPI

## Java 8 New Calendar and Locale APIs

The JDK 8 includes two new classes, several new methods, and a new return value for an existing static method.

Two new abstract classes for service providers are added to the java.util.spi package.

|  |  |
| --- | --- |
| **Class** | **Description** |
| public abstract class CalendarDataProvider extends LocaleServiceProvider | It is an abstract class for service providers that provide locale-dependent Calendar parameters. |
| public abstract class CalendarNameProvider extends LocaleServiceProvider | It is an abstract class for service providers that provide localized string representations (display names) of Calendar field values. |

A static method is now able to recognize Locale.UNICODE\_LOCALE\_EXTENSION for the numbering system.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static final DecimalFormatSymbols getInstance(Locale locale) | It is used to get the DecimalFormatSymbols instance for the specified locale. This method provides access to DecimalFormatSymbols instances for locales supported by the Java runtime itself as well as for those supported by installed DecimalFormatSymbolsProvider implementations. It throws NullPointerException if locale is null. |

Added New methods in calender API:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public boolean isSupportedLocale(Locale locale) | It returns true if the given locale is supported by this locale service provider. The given locale may contain extensions that should be taken into account for the support determination. It is define in java.util.spi.LocaleServiceProvider class |
| public String getCalendarType() | It returns the calendar type of this Calendar. Calendar types are defined by the Unicode Locale Data Markup Language (LDML) specification. It is defined in java.util.Calendar class. |

New style specifiers are added for the Calendar.getDisplayName and Calendar.getDisplayNames methods to determine the format of the Calendar name.

|  |  |
| --- | --- |
| **Specifier** | **Description** |
| public static final int SHORT\_FORMAT | It is a style specifier for getDisplayName and getDisplayNames indicating a short name used for format. |
| public static final int LONG\_FORMAT | It is a style specifier for getDisplayName and getDisplayNames indicating a long name used for format. |
| public static final int SHORT\_STANDALONE | It is a style specifier for getDisplayName and getDisplayNames indicating a short name used independently, such as a month abbreviation as calendar headers. |
| public static final int LONG\_STANDALONE | It is a style specifier for getDisplayName and getDisplayNames indicating a long name used independently, such as a month name as calendar headers. |

Two new Locale methods for dealing with a locale's (optional) extensions.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public boolean hasExtensions() | It returns true if this Locale has any extensions. |
| public Locale stripExtensions() | It returns a copy of this Locale with no extensions. If this Locale has no extensions, this Locale is returned itself. |

Two new Locale.filter methods return a list of Locale instances that match the specified criteria, as defined in RFC 4647:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static List<Locale> filter(List<Locale.LanguageRange> priorityList,Collection<Locale> locales) | It returns a list of matching Locale instances using the filtering mechanism defined in RFC 4647. This is equivalent to filter(List, Collection, FilteringMode) when mode is Locale.FilteringMode.AUTOSELECT\_FILTERING. |
| public static List<Locale> filter(List<Locale.LanguageRange> priorityList,Collection<Locale> locales, Locale.FilteringMode mode) | It returns a list of matching Locale instances using the filtering mechanism defined in RFC 4647. |

Two new Locale.filterTags methods return a list of language tags that match the specified criteria, as defined in RFC 4647.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static List<String> filterTags(List<Locale.LanguageRange> priorityList, Collection<String> tags) | It returns a list of matching languages tags using the basic filtering mechanism defined in RFC 4647. This is equivalent to filterTags(List, Collection, FilteringMode) when mode is Locale.FilteringMode.AUTOSELECT\_FILTERING. |
| public static List<String> filterTags(List<Locale.LanguageRange> priorityList, Collection<String> tags, Locale.FilteringMode mode) | It returns a list of matching languages tags using the basic filtering mechanism defined in RFC 4647. |

Two new lookup methods return the best-matching locale or language tag using the lookup mechanism defined in RFC 4647.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static Locale lookup(List<Locale.LanguageRange> priorityList, Collection<Locale> locales) | It returns a Locale instance for the best-matching language tag using the lookup mechanism defined in RFC 4647. |
| Public static String lookupTag(List<Locale.LanguageRange> priorityList,Collection<String> tags) | It returns the best-matching language tag using the lookup mechanism defined in RFC 4647. |

## Other Java 8 Version Enhancements

## Enhancements in JDK 8u5

1) The frequency in which the security prompts are shown for an application has been reduced.

## Enhancements in JDK 8u11

1) An option to suppress offers from sponsors when the JRE is installed or updated is available in the Advanced tab of the Java Control Panel.

2) The Entry-Point attribute can be included in the JAR file manifest to identify one or more classes as a valid entry point for your RIA(Rich Internet application).

## Enhancements in JDK 8u20

1) The javafxpackager tool has been renamed to javapackager. This tool has been enhanced with new arguments for self-contained application bundlers.

Follwing enhancements are related to the java tool:

* An experimental JIT compiler option related to Restricted Transactional Memory (RTM) has been added.
* Several options related to string deduplication have been added.
* Several options related to Advanced Encryption Standard (AES) intrinsics have been added.
* Combinations of garbage collection options have been deprecated.

2) Garbage Collection Tuning Guide has been added to the Java HotSpot Virtual Machine. It describes the garbage collectors included with the Java HotSpot VM and helps you to decide which garbage collector can best optimize the performance of your application, especially if it handles large amounts of data (multiple gigabytes), has many threads, and has high transaction rates.

## Enhancements in JDK 8u31

1) In this release, the SSLv3 protocol is removed from the Java Control Panel Advanced options.

## Enhancements in JDK 8u40

### **Java tool**

1) The -XX:+CheckEndorsedAndExtDirs has been added because the endorsed-standards override mechanism (JDK-8065675) and the extension mechanism (JDK-8065702) have been deprecated. The option helps identify any existing uses of these mechanisms and is supported in JDK 7u80 and JDK 8u40.

2) Java Flight Recorder (JFR) offers a variety of ways to unlock commercial features and enable JFR during the runtime of an application.

It includes java command line options such as jcmd diagnostic commands and Graphical User Interface (GUI) controls within Java Mission Control. This flexibility enables you to provide the appropriate options at startup, or interact with JFR later.

3) The option -XX:StartFlightRecording=parameter=value has a new parameter, dumponexit={true|false}, which specifies whether a dump file of JFR data should be generated when the JVM terminates in a controlled manner.

4) The options related to Restricted Transactional Memory (RTM) are no longer experimental. These options include -XX:RTMAbortRatio=abort\_ratio, -XX:RTMRetryCount=number\_of\_retries, -XX:+UseRTMDeopt, and -XX:+UseRTMLocking.

5) In Java 8, Application Class Data Sharing (AppCDS) has been introduced. AppCDS extends CDS (Class Data Sharing) to enable classes from the standard extensions directories and the application class path to be placed in the shared archive. This is a commercial feature and is no longer considered experimental.

6) New options -XX:+ResourceManagement and -XX:ResourceManagementSampleInterval=value have been added.

7) Additional information about large pages has been added. Large Pages, also known as huge pages, are memory pages that are significantly larger than the standard memory page size. Large pages optimize processor Translation-Lookaside Buffers. The Linux options -XX:+UseHugeTLBFS, -XX:+UseSHM, and -XX:+UseTransparentHugePages have been documented.

8) The option -XX:ObjectAlignmentInBytes=alignment has been documented.

### **JJS tool**

1) The option --optimistic-types=[true|false] has been added. It enables or disables optimistic type assumptions with deoptimizing recompilation.

2) The option --language=[es5] has been added to the jjs tool. It specifies the ECMAScript language version.

### **Javapackager tool**

1) New arguments are available for OS X bundlers. The mac.CFBundleVersion argument identifies the internal version number to be used.

2) The mac.dmg.simple argument indicates if DMG customization steps that depend on executing AppleScript code are skipped.

### **Jcmd tool**

Jcmd tool is used to dynamically interact with Java Flight Recorder (JFR). You can use it to unlock commercial features, enable/start/stop flight recordings, and obtain various status messages from the system.

### **Jstat tool**

The jstat tool has been updated with information about compressed class space which is a special part of metaspace.

### **Virtual machine**

The Scalable Native Memory Tracking HotSpot VM feature helps diagnose VM memory leaks and clarify users when memory leaks are not in the VM. Native Memory Tracker can be run without self-shutdown on large systems and without causing a significant performance impact beyond what is considered acceptable for small programs.

# **Java Lambda Expressions**

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection. Before lambda expression, anonymous inner class was the only option to implement the method.

In other words, we can say it is a replacement of java inner anonymous class. Java lambda expression is treated as a function, so compiler does not create .class file.

## Functional Interface

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

## Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

## Java Lambda Expression Syntax

1. (argument-list) -> {body}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

Let's see a scenario. If we don't implement Java lambda expression. Here, we are implementing an interface method without using lambda expression.

## Java Example without Lambda Expression

1. **interface** Drawable{
2. **public** **void** draw();
3. }
4. **public** **class** LambdaExpressionExample {
5. **public** **static** **void** main(String[] args) {
6. **int** width=10;
8. //without lambda, Drawable implementation using anonymous class
9. Drawable d=**new** Drawable(){
10. **public** **void** draw(){System.out.println("Drawing "+width);}
11. };
12. d.draw();
13. }
14. }

Output:

Drawing 10

## Java Example with Lambda Expression

Now, we are implementing the above example with the help of lambda expression.

1. @FunctionalInterface  //It is optional
2. **interface** Drawable{
3. **public** **void** draw();
4. }
6. **public** **class** LambdaExpressionExample {
7. **public** **static** **void** main(String[] args) {
8. **int** width=10;
10. //with lambda
11. Drawable d2=()->{
12. System.out.println("Drawing "+width);
13. };
14. d2.draw();
15. }
16. }

Output:

Drawing 10

A lambda expression can have zero or any number of arguments. Let's see the examples:

## Java Lambda Expression Example: No Parameter

1. **interface** Sayable{
2. **public** String say();
3. }
4. **public** **class** LambdaExpressionExample{
5. **public** **static** **void** main(String[] args) {
6. Sayable s=()->{
7. **return** "I have nothing to say.";
8. };
9. System.out.println(s.say());
10. }
11. }

Output:

I have nothing to say.

## Java Lambda Expression Example: Single Parameter

1. **interface** Sayable{
2. **public** String say(String name);
3. }
5. **public** **class** LambdaExpressionExample{
6. **public** **static** **void** main(String[] args) {
8. // Lambda expression with single parameter.
9. Sayable s1=(name)->{
10. **return** "Hello, "+name;
11. };
12. System.out.println(s1.say("Sonoo"));
14. // You can omit function parentheses
15. Sayable s2= name ->{
16. **return** "Hello, "+name;
17. };
18. System.out.println(s2.say("Sonoo"));
19. }
20. }

Output:

Hello, Sonoo

Hello, Sonoo

## Java Lambda Expression Example: Multiple Parameters

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample{
6. **public** **static** **void** main(String[] args) {
8. // Multiple parameters in lambda expression
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Multiple parameters with data type in lambda expression
13. Addable ad2=(**int** a,**int** b)->(a+b);
14. System.out.println(ad2.add(100,200));
15. }
16. }

Output:

30

300

## Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

1. **package** lambdaExample;
3. **interface** Addable{
4. **int** add(**int** a,**int** b);
5. }
7. **public** **class** lambdaExpression {
8. **public** **static** **void** main(String[] args) {
10. // Lambda expression without return keyword.
11. Addable ad1=(a,b)->(a+b);
12. System.out.println(ad1.add(10,20));
14. // Lambda expression with return keyword.
15. Addable ad2=(**int** a,**int** b)->{
16. **return** (a+b);
17. };
18. System.out.println(ad2.add(100,200));
19. }
20. }

Output:

30

300

## Java Lambda Expression Example: Foreach Loop

1. **import** java.util.\*;
2. **public** **class** LambdaExpressionExample{
3. **public** **static** **void** main(String[] args) {
5. List<String> list=**new** ArrayList<String>();
6. list.add("ankit");
7. list.add("mayank");
8. list.add("irfan");
9. list.add("jai");
11. list.forEach(
12. (n)->System.out.println(n)
13. );
14. }
15. }

Output:

ankit

mayank

irfan

jai

## Java Lambda Expression Example: Multiple Statements

1. @FunctionalInterface
2. **interface** Sayable{
3. String say(String message);
4. }
6. **public** **class** LambdaExpressionExample{
7. **public** **static** **void** main(String[] args) {
9. // You can pass multiple statements in lambda expression
10. Sayable person = (message)-> {
11. String str1 = "I would like to say, ";
12. String str2 = str1 + message;
13. **return** str2;
14. };
15. System.out.println(person.say("time is precious."));
16. }
17. }

Output:

I would like to say, time is precious.

## Java Lambda Expression Example: Creating Thread

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

1. **public** **class** LambdaExpressionExample{
2. **public** **static** **void** main(String[] args) {
4. //Thread Example without lambda
5. Runnable r1=**new** Runnable(){
6. **public** **void** run(){
7. System.out.println("Thread1 is running...");
8. }
9. };
10. Thread t1=**new** Thread(r1);
11. t1.start();
12. //Thread Example with lambda
13. Runnable r2=()->{
14. System.out.println("Thread2 is running...");
15. };
16. Thread t2=**new** Thread(r2);
17. t2.start();
18. }
19. }

Output:

Thread1 is running...

Thread2 is running...

Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data. Following are some lambda and collection examples provided.

## Java Lambda Expression Example: Comparator

1. **import** java.util.ArrayList;
2. **import** java.util.Collections;
3. **import** java.util.List;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
8. **public** Product(**int** id, String name, **float** price) {
9. **super**();
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** LambdaExpressionExample{
16. **public** **static** **void** main(String[] args) {
17. List<Product> list=**new** ArrayList<Product>();
19. //Adding Products
20. list.add(**new** Product(1,"HP Laptop",25000f));
21. list.add(**new** Product(3,"Keyboard",300f));
22. list.add(**new** Product(2,"Dell Mouse",150f));
24. System.out.println("Sorting on the basis of name...");
26. // implementing lambda expression
27. Collections.sort(list,(p1,p2)->{
28. **return** p1.name.compareTo(p2.name);
29. });
30. **for**(Product p:list){
31. System.out.println(p.id+" "+p.name+" "+p.price);
32. }
34. }
35. }

Output:

Sorting on the basis of name...

2 Dell Mouse 150.0

1 HP Laptop 25000.0

3 Keyboard 300.0

## Java Lambda Expression Example: Filter Collection Data

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **import** java.util.stream.Stream;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
8. **public** Product(**int** id, String name, **float** price) {
9. **super**();
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** LambdaExpressionExample{
16. **public** **static** **void** main(String[] args) {
17. List<Product> list=**new** ArrayList<Product>();
18. list.add(**new** Product(1,"Samsung A5",17000f));
19. list.add(**new** Product(3,"Iphone 6S",65000f));
20. list.add(**new** Product(2,"Sony Xperia",25000f));
21. list.add(**new** Product(4,"Nokia Lumia",15000f));
22. list.add(**new** Product(5,"Redmi4 ",26000f));
23. list.add(**new** Product(6,"Lenevo Vibe",19000f));
25. // using lambda to filter data
26. Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);
28. // using lambda to iterate through collection
29. filtered\_data.forEach(
30. product -> System.out.println(product.name+": "+product.price)
31. );
32. }
33. }

Output:

Iphone 6S: 65000.0

Sony Xperia: 25000.0

Redmi4 : 26000.0

## Java Lambda Expression Example: Event Listener

1. **import** javax.swing.JButton;
2. **import** javax.swing.JFrame;
3. **import** javax.swing.JTextField;
4. **public** **class** LambdaEventListenerExample {
5. **public** **static** **void** main(String[] args) {
6. JTextField tf=**new** JTextField();
7. tf.setBounds(50, 50,150,20);
8. JButton b=**new** JButton("click");
9. b.setBounds(80,100,70,30);
11. // lambda expression implementing here.
12. b.addActionListener(e-> {tf.setText("hello swing");});
14. JFrame f=**new** JFrame();
15. f.add(tf);f.add(b);
16. f.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
17. f.setLayout(**null**);
18. f.setSize(300, 200);
19. f.setVisible(**true**);
21. }
23. }

# **Java Method References**

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

## Types of Method References

There are four types of method references:

1. Reference to a static method.
2. Reference to an instance method of a particular object.
3. Reference to an instance method of an arbitrary object of a particular type.
4. Reference to a constructor.

## 1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

1. ContainingClass::staticMethodName

### **Example 1**

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

1. **interface** Sayable{
2. **void** say();
3. }
4. **public** **class** MethodReference {
5. **public** **static** **void** saySomething(){
6. System.out.println("Hello, this is static method.");
7. }
8. **public** **static** **void** main(String[] args) {
9. // Referring static method
10. Sayable sayable = MethodReference::saySomething;
11. // Calling interface method
12. sayable.say();
13. }
14. }

Output:

Hello, this is static method.

### **Example 2**

In the following example, we are using predefined functional interface Runnable to refer static method.

1. **public** **class** MethodReference {
2. **public** **static** **void** ThreadStatus(){
3. System.out.println("Thread is running...");
4. }
5. **public** **static** **void** main(String[] args) {
6. Thread t2=**new** Thread(MethodReference::ThreadStatus);
7. t2.start();
8. }
9. }

Output:

Thread is running...

### **Example 3**

You can also use predefined functional interface to refer methods. In the following example, we are using BiFunction interface and using it's apply() method.

1. **import** java.util.function.BiFunction;
2. **class** Arithmetic{
3. **public** **static** **int** add(inta, intb){
4. **return** a+b;
5. }
6. }
7. **public** **class** StaticMethodReference {
8. **public** **static** **void** main(String[] args) {
9. BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;
10. **int** result = adder.apply(10, 20);
11. System.out.println(result);
12. }
13. }

Output:

30

### **Example 4**

You can also override static methods by referring methods. In the following example, we have defined and overloaded three add methods.

1. **import** java.util.function.BiFunction;
3. **class** Arithmetic{
4. **public** **static** **int** add(inta, intb){
5. **return** a+b;
6. }
7. **public** **static** **float** add(inta, floatb){
8. **return** a+b;
9. }
10. **public** **static** **float** add(floata, floatb){
11. **return** a+b;
12. }
13. }
15. **public** **class** StaticMethodReference {
16. **public** **static** **void** main(String[] args) {
17. BiFunction<Integer, Integer, Integer>adder1 = Arithmetic::add;
18. BiFunction<Integer, Float, Float>adder2 = Arithmetic::add;
19. BiFunction<Float, Float, Float>adder3 = Arithmetic::add;
20. intresult1 = adder1.apply(10, 20);
21. floatresult2 = adder2.apply(10, 20.0f);
22. floatresult3 = adder3.apply(10.0f, 20.0f);
23. System.out.println(result1);
24. System.out.println(result2);
25. System.out.println(result3);
26. }
27. }

Output:

30

30.0

30.0

## 2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

1. containingObject::instanceMethodName

### **Example 1**

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

1. **interface** Sayable{
2. **void** say();
3. }
4. publicclass MethodReference {
5. **public** **void** saySomething(){
6. System.out.println("Hello, this is non-static method.");
7. }
8. **public** **static** **void** main(String[] args) {
9. MethodReference methodReference = **new** MethodReference(); // Creating object
10. // Referring non-static method using reference
11. Sayable sayable = methodReference::saySomething;
12. // Calling interface method
13. sayable.say();
14. // Referring non-static method using anonymous object
15. Sayable sayable2 = **new** MethodReference()::saySomething; // You can use anonymous object also
16. // Calling interface method
17. sayable2.say();
18. }
19. }

Output:

Hello, this is non-static method.

Hello, this is non-static method.

### **Example 2**

In the following example, we are referring instance (non-static) method. Runnable interface contains only one abstract method. So, we can use it as functional interface.

1. **public** **class** InstanceMethodReference {
2. **public** **void** printnMsg(){
3. System.out.println("Hello, this is instance method");
4. }
5. **public** **static** **void** main(String[] args) {
6. Thread t2=**new** Thread(**new** InstanceMethodReference()::printnMsg);
7. t2.start();
8. }
9. }

Output:

Hello, this is instance method

### **Example 3**

In the following example, we are using BiFunction interface. It is a predefined interface and contains a functional method apply(). Here, we are referring add method to apply method.

1. **import** java.util.function.BiFunction;
2. **class** Arithmetic{
3. **public** **int** add(inta, intb){
4. **return** a+b;
5. }
6. }
7. **public** **class** InstanceMethodReference {
8. **public** **static** **void** main(String[] args) {
9. BiFunction<Integer, Integer, Integer>adder = **new** Arithmetic()::add;
10. **int** result = adder.apply(10, 20);
11. System.out.println(result);
12. }
13. }

Output:

30

## 4) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

1. ClassName::**new**

### **Example**

1. **interface** Messageable{
2. Message getMessage(String msg);
3. }
4. **class** Message{
5. **public** Message(String msg){
6. System.out.print(msg);
7. }
8. }
9. **public** **class** ConstructorReference {
10. **public** **static** **void** main(String[] args) {
11. Messageable hello = Message::**new**;
12. hello.getMessage("Hello");
13. }
14. }

Output:

Hello

# **Java Functional Interfaces**

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

### **Example 1**

1. @FunctionalInterface
2. **interface** sayable{
3. **void** say(String msg);
4. }
5. **public** **class** FunctionalInterfaceExample **implements** sayable{
6. **public** **void** say(String msg){
7. System.out.println(msg);
8. }
9. **public** **static** **void** main(String[] args) {
10. FunctionalInterfaceExample fie = **new** FunctionalInterfaceExample();
11. fie.say("Hello there");
12. }
13. }

Output:

Hello there

A functional interface can have methods of object class. See in the following example.

### **Example 2**

2. @FunctionalInterface
3. **interface** sayable{
4. **void** say(String msg);   // abstract method
6. // It can contain any number of methods of Object class.
7. **int** hashCode();
8. String toString();
9. **boolean** equals(Object obj);
10. }
11. **public** **class** FunctionalInterfaceExample **implements** sayable{
13. **public** **void** say(String msg){
14. System.out.println(msg);
15. }
16. **public** **static** **void** main(String[] args) {
17. FunctionalInterfaceExample fie = **new** FunctionalInterfaceExample();
18. fie.say("Hello there");
19. }
20. }

Output:

Hello there

A functional interface can extends to other interface only when that does not have any abstract method.

### **Example 3**

1. **interface** sayable{
2. **void** say(String msg);   // abstract method
3. }
4. @FunctionalInterface
5. **interface** doable **extends** sayable{
6. // Invalid '@FunctionalInterface' annotation; doable is not a functional interface
7. **void** doIt();
8. }

Output:

compile-time error

### **Example 4**

In the following example, a functional interface is extending to a non-functional interface.

1. **interface** doable{
2. **default** **void** doIt(){
3. System.out.println("Do it now");
4. }
5. }
7. @FunctionalInterface
8. **interface** sayable **extends** doable{
9. **void** say(String msg);   // abstract method
10. }
12. **public** **class** FunctionalInterfaceExample **implements** sayable{
14. **public** **void** say(String msg){
15. System.out.println(msg);
16. }
17. **public** **static** **void** main(String[] args) {
18. FunctionalInterfaceExample fie = **new** FunctionalInterfaceExample();
19. fie.say("Hello there");
20. fie.doIt();
21. }
22. }

Output:

Hello there

Do it now

### **Java Predefined-Functional Interfaces**

Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.

You can also define your own custom functional interface. Following is the list of functional interface which are placed in java.util.function package.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| [BiConsumer<T,U>](https://www.javatpoint.com/java-biconsumer-interface) | It represents an operation that accepts two input arguments and returns no result. |
| [Consumer<T>](https://www.javatpoint.com/java-consumer-interface) | It represents an operation that accepts a single argument and returns no result. |
| [Function<T,R>](https://www.javatpoint.com/java-function-interface) | It represents a function that accepts one argument and returns a result. |
| [Predicate<T>](https://www.javatpoint.com/java-predicate-interface) | It represents a predicate (boolean-valued function) of one argument. |
| BiFunction<T,U,R> | It represents a function that accepts two arguments and returns a a result. |
| BinaryOperator<T> | It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands. |
| BiPredicate<T,U> | It represents a predicate (boolean-valued function) of two arguments. |
| BooleanSupplier | It represents a supplier of boolean-valued results. |
| DoubleBinaryOperator | It represents an operation upon two double type operands and returns a double type value. |
| DoubleConsumer | It represents an operation that accepts a single double type argument and returns no result. |
| DoubleFunction<R> | It represents a function that accepts a double type argument and produces a result. |
| DoublePredicate | It represents a predicate (boolean-valued function) of one double type argument. |
| DoubleSupplier | It represents a supplier of double type results. |
| DoubleToIntFunction | It represents a function that accepts a double type argument and produces an int type result. |
| DoubleToLongFunction | It represents a function that accepts a double type argument and produces a long type result. |
| DoubleUnaryOperator | It represents an operation on a single double type operand that produces a double type result. |
| IntBinaryOperator | It represents an operation upon two int type operands and returns an int type result. |
| IntConsumer | It represents an operation that accepts a single integer argument and returns no result. |
| IntFunction<R> | It represents a function that accepts an integer argument and returns a result. |
| IntPredicate | It represents a predicate (boolean-valued function) of one integer argument. |
| IntSupplier | It represents a supplier of integer type. |
| IntToDoubleFunction | It represents a function that accepts an integer argument and returns a double. |
| IntToLongFunction | It represents a function that accepts an integer argument and returns a long. |
| IntUnaryOperator | It represents an operation on a single integer operand that produces an integer result. |
| LongBinaryOperator | It represents an operation upon two long type operands and returns a long type result. |
| LongConsumer | It represents an operation that accepts a single long type argument and returns no result. |
| LongFunction<R> | It represents a function that accepts a long type argument and returns a result. |
| LongPredicate | It represents a predicate (boolean-valued function) of one long type argument. |
| LongSupplier | It represents a supplier of long type results. |
| LongToDoubleFunction | It represents a function that accepts a long type argument and returns a result of double type. |
| LongToIntFunction | It represents a function that accepts a long type argument and returns an integer result. |
| LongUnaryOperator | It represents an operation on a single long type operand that returns a long type result. |
| ObjDoubleConsumer<T> | It represents an operation that accepts an object and a double argument, and returns no result. |
| ObjIntConsumer<T> | It represents an operation that accepts an object and an integer argument. It does not return result. |
| ObjLongConsumer<T> | It represents an operation that accepts an object and a long argument, it returns no result. |
| Supplier<T> | It represents a supplier of results. |
| ToDoubleBiFunction<T,U> | It represents a function that accepts two arguments and produces a double type result. |
| ToDoubleFunction<T> | It represents a function that returns a double type result. |
| ToIntBiFunction<T,U> | It represents a function that accepts two arguments and returns an integer. |
| ToIntFunction<T> | It represents a function that returns an integer. |
| ToLongBiFunction<T,U> | It represents a function that accepts two arguments and returns a result of long type. |
| ToLongFunction<T> | It represents a function that returns a result of long type. |
| UnaryOperator<T> | It represents an operation on a single operand that returnsa a result of the same type as its operand. |

# **Java 8 Stream**

Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing java.util.stream package.

Stream provides following features:

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

## Java Stream Interface Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| boolean allMatch(Predicate<? super T> predicate) | It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated. |
| boolean anyMatch(Predicate<? super T> predicate) | It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated. |
| static <T> Stream.Builder<T> builder() | It returns a builder for a Stream. |
| <R,A> R collect(Collector<? super T,A,R> collector) | It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning. |
| <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) | It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result. |
| static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b) | It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked. |
| long count() | It returns the count of elements in this stream. This is a special case of a reduction. |
| Stream<T> distinct() | It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream. |
| static <T> Stream<T> empty() | It returns an empty sequential Stream. |
| Stream<T> filter(Predicate<? super T> predicate) | It returns a stream consisting of the elements of this stream that match the given predicate. |
| Optional<T> findAny() | It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty. |
| Optional<T> findFirst() | It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned. |
| <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) | It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper) | It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper) | It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper) | It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| void forEach(Consumer<? super T> action) | It performs an action for each element of this stream. |
| void forEachOrdered(Consumer<? super T> action) | It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order. |
| static <T> Stream<T> generate(Supplier<T> s) | It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc. |
| static <T> Stream<T> iterate(T seed,UnaryOperator<T> f) | It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc. |
| Stream<T> limit(long maxSize) | It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length. |
| <R> Stream<R> map(Function<? super T,? extends R> mapper) | It returns a stream consisting of the results of applying the given function to the elements of this stream. |
| DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) | It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream. |
| IntStream mapToInt(ToIntFunction<? super T> mapper) | It returns an IntStream consisting of the results of applying the given function to the elements of this stream. |
| LongStream mapToLong(ToLongFunction<? super T> mapper) | It returns a LongStream consisting of the results of applying the given function to the elements of this stream. |
| Optional<T> max(Comparator<? super T> comparator) | It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction. |
| Optional<T> min(Comparator<? super T> comparator) | It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction. |
| boolean noneMatch(Predicate<? super T> predicate) | It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated. |
| @SafeVarargs static <T> Stream<T> of(T... values) | It returns a sequential ordered stream whose elements are the specified values. |
| static <T> Stream<T> of(T t) | It returns a sequential Stream containing a single element. |
| Stream<T> peek(Consumer<? super T> action) | It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. |
| Optional<T> reduce(BinaryOperator<T> accumulator) | It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any. |
| T reduce(T identity, BinaryOperator<T> accumulator) | It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. |
| <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) | It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions. |
| Stream<T> skip(long n) | It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned. |
| Stream<T> sorted() | It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed. |
| Stream<T> sorted(Comparator<? super T> comparator) | It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator. |
| Object[] toArray() | It returns an array containing the elements of this stream. |
| <A> A[] toArray(IntFunction<A[]> generator) | It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing. |

### **Java Example: Filtering Collection without using Stream**

In the following example, we are filtering data without using stream. This approach we are used before the stream package was released.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. List<Float> productPriceList = **new** ArrayList<Float>();
22. **for**(Product product: productsList){
24. // filtering data of list
25. **if**(product.price<30000){
26. productPriceList.add(product.price);    // adding price to a productPriceList
27. }
28. }
29. System.out.println(productPriceList);   // displaying data
30. }
31. }

Output:

[25000.0, 28000.0, 28000.0]

### **Java Stream Example: Filtering Collection by using Stream**

Here, we are filtering data by using stream. You can see that code is optimized and maintained. Stream provides fast execution.

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
13. **public** **class** JavaStreamExample {
14. **public** **static** **void** main(String[] args) {
15. List<Product> productsList = **new** ArrayList<Product>();
16. //Adding Products
17. productsList.add(**new** Product(1,"HP Laptop",25000f));
18. productsList.add(**new** Product(2,"Dell Laptop",30000f));
19. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
20. productsList.add(**new** Product(4,"Sony Laptop",28000f));
21. productsList.add(**new** Product(5,"Apple Laptop",90000f));
22. List<Float> productPriceList2 =productsList.stream()
23. .filter(p -> p.price > 30000)// filtering data
24. .map(p->p.price)        // fetching price
25. .collect(Collectors.toList()); // collecting as list
26. System.out.println(productPriceList2);
27. }
28. }

Output:

[90000.0]

### **Java Stream Iterating Example**

You can use stream to iterate any number of times. Stream provides predefined methods to deal with the logic you implement. In the following example, we are iterating, filtering and passed a limit to fix the iteration.

1. **import** java.util.stream.\*;
2. **public** **class** JavaStreamExample {
3. **public** **static** **void** main(String[] args){
4. Stream.iterate(1, element->element+1)
5. .filter(element->element%5==0)
6. .limit(5)
7. .forEach(System.out::println);
8. }
9. }

Output:

5

10

15

20

25

### **Java Stream Example: Filtering and Iterating Collection**

In the following example, we are using filter() method. Here, you can see code is optimized and very concise.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. // This is more compact approach for filtering data
22. productsList.stream()
23. .filter(product -> product.price == 30000)
24. .forEach(product -> System.out.println(product.name));
25. }
26. }

Output:

Dell Laptop

### **Java Stream Example : reduce() Method in Collection**

This method takes a sequence of input elements and combines them into a single summary result by repeated operation. For example, finding the sum of numbers, or accumulating elements into a list.

In the following example, we are using reduce() method, which is used to sum of all the product prices.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. // This is more compact approach for filtering data
22. Float totalPrice = productsList.stream()
23. .map(product->product.price)
24. .reduce(0.0f,(sum, price)->sum+price);   // accumulating price
25. System.out.println(totalPrice);
26. // More precise code
27. **float** totalPrice2 = productsList.stream()
28. .map(product->product.price)
29. .reduce(0.0f,Float::sum);   // accumulating price, by referring method of Float class
30. System.out.println(totalPrice2);
32. }
33. }

Output:

201000.0

201000.0

### **Java Stream Example: Sum by using Collectors Methods**

We can also use collectors to compute sum of numeric values. In the following example, we are using Collectors class and it?s specified methods to compute sum of all the product prices.

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
13. **public** **class** JavaStreamExample {
14. **public** **static** **void** main(String[] args) {
15. List<Product> productsList = **new** ArrayList<Product>();
16. //Adding Products
17. productsList.add(**new** Product(1,"HP Laptop",25000f));
18. productsList.add(**new** Product(2,"Dell Laptop",30000f));
19. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
20. productsList.add(**new** Product(4,"Sony Laptop",28000f));
21. productsList.add(**new** Product(5,"Apple Laptop",90000f));
22. // Using Collectors's method to sum the prices.
23. **double** totalPrice3 = productsList.stream()
24. .collect(Collectors.summingDouble(product->product.price));
25. System.out.println(totalPrice3);
27. }
28. }

Output:

201000.0

### **Java Stream Example: Find Max and Min Product Price**

Following example finds min and max product price by using stream. It provides convenient way to find values without using imperative approach.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. // max() method to get max Product price
22. Product productA = productsList.stream()
23. .max((product1, product2)->
24. product1.price > product2.price ? 1: -1).get();
26. System.out.println(productA.price);
27. // min() method to get min Product price
28. Product productB = productsList.stream()
29. .max((product1, product2)->
30. product1.price < product2.price ? 1: -1).get();
31. System.out.println(productB.price);
33. }
34. }

Output:

90000.0

25000.0

### **Java Stream Example: count() Method in Collection**

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. // count number of products based on the filter
22. **long** count = productsList.stream()
23. .filter(product->product.price<30000)
24. .count();
25. System.out.println(count);
26. }
27. }

Output:

3

stream allows you to collect your result in any various forms. You can get you result as set, list or map and can perform manipulation on the elements.

### **Java Stream Example : Convert List into Set**

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
14. **public** **class** JavaStreamExample {
15. **public** **static** **void** main(String[] args) {
16. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
25. // Converting product List into Set
26. Set<Float> productPriceList =
27. productsList.stream()
28. .filter(product->product.price < 30000)   // filter product on the base of price
29. .map(product->product.price)
30. .collect(Collectors.toSet());   // collect it as Set(remove duplicate elements)
31. System.out.println(productPriceList);
32. }
33. }

Output:

[25000.0, 28000.0]

### **Java Stream Example : Convert List into Map**

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
14. **public** **class** JavaStreamExample {
15. **public** **static** **void** main(String[] args) {
16. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
25. // Converting Product List into a Map
26. Map<Integer,String> productPriceMap =
27. productsList.stream()
28. .collect(Collectors.toMap(p->p.id, p->p.name));
30. System.out.println(productPriceMap);
31. }
32. }

Output:

{1=HP Laptop, 2=Dell Laptop, 3=Lenevo Laptop, 4=Sony Laptop, 5=Apple Laptop}

### **Method Reference in stream**

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
15. **public** **int** getId() {
16. **return** id;
17. }
18. **public** String getName() {
19. **return** name;
20. }
21. **public** **float** getPrice() {
22. **return** price;
23. }
24. }
26. **public** **class** JavaStreamExample {
28. **public** **static** **void** main(String[] args) {
30. List<Product> productsList = **new** ArrayList<Product>();
32. //Adding Products
33. productsList.add(**new** Product(1,"HP Laptop",25000f));
34. productsList.add(**new** Product(2,"Dell Laptop",30000f));
35. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
36. productsList.add(**new** Product(4,"Sony Laptop",28000f));
37. productsList.add(**new** Product(5,"Apple Laptop",90000f));
39. List<Float> productPriceList =
40. productsList.stream()
41. .filter(p -> p.price > 30000) // filtering data
42. .map(Product::getPrice)         // fetching price by referring getPrice method
43. .collect(Collectors.toList());  // collecting as list
44. System.out.println(productPriceList);
45. }
46. }

Output:

[90000.0]

# **Java Stream Filter**

Java stream provides a method filter() to filter stream elements on the basis of given predicate. Suppose you want to get only even elements of your list then you can do this easily with the help of filter method.

This method takes predicate as an argument and returns a stream of consisting of resulted elements.

## Signature

The signature of Stream filter() method is given below:

1. Stream<T> filter(Predicate<? **super** T> predicate)

### **Parameter**

**predicate:** It takes Predicate reference as an argument. Predicate is a functional interface. So, you can also pass lambda expression here.

### **Return**

It returns a new stream.

### **Java Stream filter() example**

In the following example, we are fetching and iterating filtered data.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. productsList.stream()
22. .filter(p ->p.price> 30000)   // filtering price
23. .map(pm ->pm.price)          // fetching price
24. .forEach(System.out::println);  // iterating price
25. }
26. }

Output:

90000.0

### **Java Stream filter() example 2**

In the following example, we are fetching filtered data as a list.

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
13. **public** **class** JavaStreamExample {
14. **public** **static** **void** main(String[] args) {
15. List<Product> productsList = **new** ArrayList<Product>();
16. //Adding Products
17. productsList.add(**new** Product(1,"HP Laptop",25000f));
18. productsList.add(**new** Product(2,"Dell Laptop",30000f));
19. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
20. productsList.add(**new** Product(4,"Sony Laptop",28000f));
21. productsList.add(**new** Product(5,"Apple Laptop",90000f));
22. List<Float> pricesList =  productsList.stream()
23. .filter(p ->p.price> 30000)   // filtering price
24. .map(pm ->pm.price)          // fetching price
25. .collect(Collectors.toList());
26. System.out.println(pricesList);
27. }
28. }

Output:

[90000.0]

# **Java Base64 Encode and Decode**

Java provides a class Base64 to deal with encryption. You can encrypt and decrypt your data by using provided methods. You need to import java.util.Base64 in your source file to use its methods.

This class provides three different encoders and decoders to encrypt information at each level. You can use these methods at the following levels.

## Basic Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 and RFC 2045 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

## URL and Filename Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

## MIME

It uses the Base64 alphabet as specified in RFC 2045 for encoding and decoding operations. The encoded output must be represented in lines of no more than 76 characters each and uses a carriage return '\r' followed immediately by a linefeed '\n' as the line separator. No line separator is added to the end of the encoded output. All line separators or other characters not found in the base64 alphabet table are ignored in decoding operation.

### **Nested Classes of Base64**

|  |  |
| --- | --- |
| **Class** | **Description** |
| Base64.Decoder | This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. |
| Base64.Encoder | This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. |

### **Base64 Methods**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static Base64.Decoder getDecoder() | It returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme. |
| public static Base64.Encoder getEncoder() | It returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme. |
| public static Base64.Decoder getUrlDecoder() | It returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme. |
| public static Base64.Decoder getMimeDecoder() | It returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme. |
| public static Base64.Encoder getMimeEncoder() | It Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme. |
| public static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator) | It returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators. |
| public static Base64.Encoder getUrlEncoder() | It returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme. |

### **Base64.Decoder Methods**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public byte[] decode(byte[] src) | It decodes all bytes from the input byte array using the Base64 encoding scheme, writing the results into a newly-allocated output byte array. The returned byte array is of the length of the resulting bytes. |
| public byte[] decode(String src) | It decodes a Base64 encoded String into a newly-allocated byte array using the Base64 encoding scheme. |
| public int decode(byte[] src, byte[] dst) | It decodes all bytes from the input byte array using the Base64 encoding scheme, writing the results into the given output byte array, starting at offset 0. |
| public ByteBuffer decode(ByteBuffer buffer) | It decodes all bytes from the input byte buffer using the Base64 encoding scheme, writing the results into a newly-allocated ByteBuffer. |
| public InputStream wrap(InputStream is) | It returns an input stream for decoding Base64 encoded byte stream. |

### **Base64.Encoder Methods**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public byte[] encode(byte[] src) | It encodes all bytes from the specified byte array into a newly-allocated byte array using the Base64 encoding scheme. The returned byte array is of the length of the resulting bytes. |
| public int encode(byte[] src, byte[] dst) | It encodes all bytes from the specified byte array using the Base64 encoding scheme, writing the resulting bytes to the given output byte array, starting at offset 0. |
| public String encodeToString(byte[] src) | It encodes the specified byte array into a String using the Base64 encoding scheme. |
| public ByteBuffer encode(ByteBuffer buffer) | It encodes all remaining bytes from the specified byte buffer into a newly-allocated ByteBuffer using the Base64 encoding scheme. Upon return, the source buffer's position will be updated to its limit; its limit will not have been changed. The returned output buffer's position will be zero and its limit will be the number of resulting encoded bytes. |
| public OutputStream wrap(OutputStream os) | It wraps an output stream for encoding byte data using the Base64 encoding scheme. |
| public Base64.Encoder withoutPadding() | It returns an encoder instance that encodes equivalently to this one, but without adding any padding character at the end of the encoded byte data. |

### **Java Base64 Example: Basic Encoding and Decoding**

1. **import** java.util.Base64;
2. publicclass Base64BasicEncryptionExample {
3. publicstaticvoid main(String[] args) {
4. // Getting encoder
5. Base64.Encoder encoder = Base64.getEncoder();
6. // Creating byte array
7. bytebyteArr[] = {1,2};
8. // encoding byte array
9. bytebyteArr2[] = encoder.encode(byteArr);
10. System.out.println("Encoded byte array: "+byteArr2);
11. bytebyteArr3[] = newbyte[5];                // Make sure it has enough size to store copied bytes
12. intx = encoder.encode(byteArr,byteArr3);    // Returns number of bytes written
13. System.out.println("Encoded byte array written to another array: "+byteArr3);
14. System.out.println("Number of bytes written: "+x);
16. // Encoding string
17. String str = encoder.encodeToString("JavaTpoint".getBytes());
18. System.out.println("Encoded string: "+str);
19. // Getting decoder
20. Base64.Decoder decoder = Base64.getDecoder();
21. // Decoding string
22. String dStr = **new** String(decoder.decode(str));
23. System.out.println("Decoded string: "+dStr);
24. }
25. }

Output:

Encoded byte array: [B@6bc7c054

Encoded byte array written to another array: [B@232204a1

Number of bytes written: 4

Encoded string: SmF2YVRwb2ludA==

Decoded string: JavaTpoint

### **Java Base64 Example: URL Encoding and Decoding**

1. **import** java.util.Base64;
2. publicclass Base64BasicEncryptionExample {
3. publicstaticvoid main(String[] args) {
4. // Getting encoder
5. Base64.Encoder encoder = Base64.getUrlEncoder();
6. // Encoding URL
7. String eStr = encoder.encodeToString("http://www.javatpoint.com/java-tutorial/".getBytes());
8. System.out.println("Encoded URL: "+eStr);
9. // Getting decoder
10. Base64.Decoder decoder = Base64.getUrlDecoder();
11. // Decoding URl
12. String dStr = **new** String(decoder.decode(eStr));
13. System.out.println("Decoded URL: "+dStr);
14. }
15. }

Output:

Encoded URL: aHR0cDovL3d3dy5qYXZhdHBvaW50LmNvbS9qYXZhLXR1dG9yaWFsLw==

Decoded URL: http://www.javatpoint.com/java-tutorial/

### **Java Base64 Example: MIME Encoding and Decoding**

1. **package** Base64Encryption;
2. **import** java.util.Base64;
3. publicclass Base64BasicEncryptionExample {
4. publicstaticvoid main(String[] args) {
5. // Getting MIME encoder
6. Base64.Encoder encoder = Base64.getMimeEncoder();
7. String message = "Hello, \nYou are informed regarding your inconsistency of work";
8. String eStr = encoder.encodeToString(message.getBytes());
9. System.out.println("Encoded MIME message: "+eStr);
11. // Getting MIME decoder
12. Base64.Decoder decoder = Base64.getMimeDecoder();
13. // Decoding MIME encoded message
14. String dStr = **new** String(decoder.decode(eStr));
15. System.out.println("Decoded message: "+dStr);
16. }
17. }

Output:

Encoded MIME message: SGVsbG8sIApZb3UgYXJlIGluZm9ybWVkIHJlZ2FyZGluZyB5b3VyIGluY29uc2lzdGVuY3kgb2Yg

d29yaw==

Decoded message: Hello,

You are informed regarding your inconsistency of work

# **Java Default Methods**

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

### **Java Default Method Example**

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Let's see a simple

1. **interface** Sayable{
2. // Default method
3. **default** **void** say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. **void** sayMore(String msg);
8. }
9. **public** **class** DefaultMethods **implements** Sayable{
10. **public** **void** sayMore(String msg){        // implementing abstract method
11. System.out.println(msg);
12. }
13. **public** **static** **void** main(String[] args) {
14. DefaultMethods dm = **new** DefaultMethods();
15. dm.say();   // calling default method
16. dm.sayMore("Work is worship");  // calling abstract method
18. }
19. }

Output:

Hello, this is default method

Work is worship

## Static Methods inside Java 8 Interface

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

1. **interface** Sayable{
2. // default method
3. **default** **void** say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. **void** sayMore(String msg);
8. // static method
9. **static** **void** sayLouder(String msg){
10. System.out.println(msg);
11. }
12. }
13. **public** **class** DefaultMethods **implements** Sayable{
14. **public** **void** sayMore(String msg){     // implementing abstract method
15. System.out.println(msg);
16. }
17. **public** **static** **void** main(String[] args) {
18. DefaultMethods dm = **new** DefaultMethods();
19. dm.say();                       // calling default method
20. dm.sayMore("Work is worship");      // calling abstract method
21. Sayable.sayLouder("Helloooo...");   // calling static method
22. }
23. }

Output:

Hello there

Work is worship

Helloooo...

## Abstract Class vs Java 8 Interface

After having default and static methods inside the interface, we think about the need of abstract class in Java. An interface and an abstract class is almost similar except that you can create constructor in the abstract class whereas you can't do this in interface.

1. **abstract** **class** AbstractClass{
2. **public** AbstractClass() {        // constructor
3. System.out.println("You can create constructor in abstract class");
4. }
5. **abstract** **int** add(**int** a, **int** b); // abstract method
6. **int** sub(**int** a, **int** b){      // non-abstract method
7. **return** a-b;
8. }
9. **static** **int** multiply(**int** a, **int** b){  // static method
10. **return** a\*b;
11. }
12. }
13. **public** **class** AbstractTest **extends** AbstractClass{
14. **public** **int** add(**int** a, **int** b){        // implementing abstract method
15. **return** a+b;
16. }
17. **public** **static** **void** main(String[] args) {
18. AbstractTest a = **new** AbstractTest();
19. **int** result1 = a.add(20, 10);    // calling abstract method
20. **int** result2 = a.sub(20, 10);    // calling non-abstract method
21. **int** result3 = AbstractClass.multiply(20, 10); // calling static method
22. System.out.println("Addition: "+result1);
23. System.out.println("Substraction: "+result2);
24. System.out.println("Multiplication: "+result3);
25. }
26. }

Output:

You can create constructor in abstract class

Addition: 30

Substraction: 10

Multiplication: 200

# **Java forEach loop**

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

## forEach() Signature in Iterable Interface

1. **default** **void** forEach(Consumer<**super** T>action)

### **Java 8 forEach() example 1**

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing lambda expression--------------");
11. gamesList.forEach(games -> System.out.println(games));
13. }
14. }

Output:

------------Iterating by passing lambda expression--------------

Football

Cricket

Chess

Hocky

### **Java 8 forEach() example 2**

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing method reference---------------");
11. gamesList.forEach(System.out::println);
12. }
13. }

Output:

------------Iterating by passing method reference---------------

Football

Cricket

Chess

Hocky

## Java Stream forEachOrdered() Method

Along with forEach() method, Java provides one more method forEachOrdered(). It is used to iterate elements in the order specified by the stream.

### **Singnature:**

1. **void** forEachOrdered(Consumer<? **super** T> action)

### **Java Stream forEachOrdered() Method Example**

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachOrderedExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing lambda expression---------------");
11. gamesList.stream().forEachOrdered(games -> System.out.println(games));
12. System.out.println("------------Iterating by passing method reference---------------");
13. gamesList.stream().forEachOrdered(System.out::println);
14. }
16. }

Output:

------------Iterating by passing lambda expression---------------

Football

Cricket

Chess

Hocky

------------Iterating by passing method reference---------------

Football

Cricket

Chess

Hocky

# **Java Collectors**

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Java Collectors class provides various methods to deal with elements

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static <T> Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper) | It returns a Collector that produces the arithmetic mean of a double-valued function applied to the input elements. If no elements are present, the result is 0. |
| public static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> op) | It returns a Collector which performs a reduction of its input elements under a specified BinaryOperator using the provided identity. |
| public static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> op) | It returns a Collector which performs a reduction of its input elements under a specified BinaryOperator. The result is described as an Optional<T>. |
| public static <T,U> Collector<T,?,U> reducing(U identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op) | It returns a Collector which performs a reduction of its input elements under a specified mapping function and BinaryOperator. This is a generalization of reducing(Object, BinaryOperator) which allows a transformation of the elements before reduction. |
| public static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier) | It returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map. |
| public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? Super T,A,D> downstream) | It returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. |
| public static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream) | It returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. The Map produced by the Collector is created with the supplied factory function. |
| public static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier) | It returns a concurrent Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function. |
| public static <T,K,A,D> Collector<T,?,ConcurrentMap<K,D>> groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream) | It returns a concurrent Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. |
| public static <T,K,A,D,M extends ConcurrentMap<K,D>> Collector<T,?,M> groupingByConcurrent(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream) | It returns a concurrent Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. The ConcurrentMap produced by the Collector is created with the supplied factory function. |
| public static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate) | It returns a Collector which partitions the input elements according to a Predicate, and organizes them into a Map<Boolean, List<T>>. There are no guarantees on the type, mutability, serializability, or thread-safety of the Map returned. |
| public static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(Predicate<? super T> predicate, Collector<? Super T,A,D> downstream) | It returns a Collector which partitions the input elements according to a Predicate, reduces the values in each partition according to another Collector, and organizes them into a Map<Boolean, D> whose values are the result of the downstream reduction. |
| public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper) | It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction) | It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier) | It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper) | It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction) | It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U,M extends ConcurrentMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier) | It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper) | It returns a Collector which applies an int-producing mapping function to each input element, and returns summary statistics for the resulting values. |
| public static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper) | It returns a Collector which applies an long-producing mapping function to each input element, and returns summary statistics for the resulting values. |
| public static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper) | It returns a Collector which applies an double-producing mapping function to each input element, and returns summary statistics for the resulting values. |

### **Java Collectors Example: Fetching data as a List**

1. **import** java.util.stream.Collectors;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** CollectorsExample {
16. **public** **static** **void** main(String[] args) {
17. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
24. List<Float> productPriceList =
25. productsList.stream()
26. .map(x->x.price)         // fetching price
27. .collect(Collectors.toList());  // collecting as list
28. System.out.println(productPriceList);
29. }
30. }

Output:

[25000.0, 30000.0, 28000.0, 28000.0, 90000.0]

### **Java Collectors Example: Converting Data as a Set**

1. **import** java.util.stream.Collectors;
2. **import** java.util.Set;
3. **import** java.util.List;
4. **import** java.util.ArrayList;
5. classProduct{
6. intid;
7. String name;
8. floatprice;
10. **public** Product(intid, String name, floatprice) {
11. **this**.id = id;
12. **this**.name = name;
13. **this**.price = price;
14. }
15. }
16. publicclass CollectorsExample {
17. publicstaticvoid main(String[] args) {
18. List<Product>productsList = **new** ArrayList<Product>();
19. //Adding Products
20. productsList.add(newProduct(1,"HP Laptop",25000f));
21. productsList.add(newProduct(2,"Dell Laptop",30000f));
22. productsList.add(newProduct(3,"Lenevo Laptop",28000f));
23. productsList.add(newProduct(4,"Sony Laptop",28000f));
24. productsList.add(newProduct(5,"Apple Laptop",90000f));
25. Set<Float>productPriceList =
26. productsList.stream()
27. .map(x->x.price)         // fetching price
28. .collect(Collectors.toSet());   // collecting as list
29. System.out.println(productPriceList);
30. }
31. }

Output:

[25000.0, 30000.0, 28000.0, 90000.0]

### **Java Collectors Example: using sum method**

1. **import** java.util.stream.Collectors;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** CollectorsExample {
16. **public** **static** **void** main(String[] args) {
17. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
24. Double sumPrices =
25. productsList.stream()
26. .collect(Collectors.summingDouble(x->x.price));  // collecting as list
27. System.out.println("Sum of prices: "+sumPrices);
28. Integer sumId =
29. productsList.stream().collect(Collectors.summingInt(x->x.id));
30. System.out.println("Sum of id's: "+sumId);
31. }
32. }

Output:

Sum of prices: 201000.0

Sum of id's: 15

### **Java Collectors Example: Getting Product Average Price**

1. **import** java.util.stream.Collectors;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** CollectorsExample {
16. **public** **static** **void** main(String[] args) {
17. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
24. Double average = productsList.stream()
25. .collect(Collectors.averagingDouble(p->p.price));
26. System.out.println("Average price is: "+average);
27. }
28. }

Output:

Average price is: 40200.0

### **Java Collectors Example: Counting Elements**

1. **import** java.util.stream.Collectors;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **class** Product{
5. intid;
6. String name;
7. floatprice;
9. **public** Product(intid, String name, floatprice) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. publicint getId() {
15. returnid;
16. }
17. **public** String getName() {
18. returnname;
19. }
20. publicfloat getPrice() {
21. returnprice;
22. }
23. }
24. publicclass CollectorsExample {
25. publicstaticvoid main(String[] args) {
26. List<Product>productsList = **new** ArrayList<Product>();
27. //Adding Products
28. productsList.add(**new** Product(1,"HP Laptop",25000f));
29. productsList.add(**new** Product(2,"Dell Laptop",30000f));
30. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
31. productsList.add(**new** Product(4,"Sony Laptop",28000f));
32. productsList.add(**new** Product(5,"Apple Laptop",90000f));
33. Long noOfElements = productsList.stream()
34. .collect(Collectors.counting());
35. System.out.println("Total elements : "+noOfElements);
36. }
37. }

Output:

Total elements : 5

# **Java StringJoiner**

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

### **StringJoiner Constructors**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| Public StringJoiner(CharSequence delimiter) | It constructs a StringJoiner with no characters in it, with no prefix or suffix, and a copy of the supplied delimiter. It throws NullPointerException if delimiter is null. |
| Public StringJoiner(CharSequence delimiter,CharSequence prefix,CharSequence suffix) | It constructs a StringJoiner with no characters in it using copies of the supplied prefix, delimiter and suffix. It throws NullPointerException if prefix, delimiter, or suffix is null. |

### **StringJoiner Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Public StringJoiner add(CharSequence newElement) | It adds a copy of the given CharSequence value as the next element of the StringJoiner value. If newElement is null,"null" is added. |
| Public StringJoiner merge(StringJoiner other) | It adds the contents of the given StringJoiner without prefix and suffix as the next element if it is non-empty. If the given StringJoiner is empty, the call has no effect. |
| Public int length() | It returns the length of the String representation of this StringJoiner. |
| Public StringJoiner setEmptyValue(CharSequence emptyValue) | It sets the sequence of characters to be used when determining the string representation of this StringJoiner and no elements have been added yet, that is, when it is empty. |

### **Java StringJoiner Example**

1. // importing StringJoiner class
2. **import** java.util.StringJoiner;
3. **public** **class** StringJoinerExample {
4. **public** **static** **void** main(String[] args) {
5. StringJoiner joinNames = **new** StringJoiner(","); // passing comma(,) as delimiter
7. // Adding values to StringJoiner
8. joinNames.add("Rahul");
9. joinNames.add("Raju");
10. joinNames.add("Peter");
11. joinNames.add("Raheem");
13. System.out.println(joinNames);
14. }
15. }

Output:

Rahul,Raju,Peter,Raheem

### **Java StringJoiner Example: adding prefix and suffix**

1. // importing StringJoiner class
2. **import** java.util.StringJoiner;
3. **public** **class** StringJoinerExample {
4. **public** **static** **void** main(String[] args) {
5. StringJoiner joinNames = **new** StringJoiner(",", "[", "]");   // passing comma(,) and square-brackets as delimiter
7. // Adding values to StringJoiner
8. joinNames.add("Rahul");
9. joinNames.add("Raju");
10. joinNames.add("Peter");
11. joinNames.add("Raheem");
13. System.out.println(joinNames);
14. }
15. }

Output:

[Rahul,Raju,Peter,Raheem]

### **StringJoiner Example: Merge Two StringJoiner**

The merge() method merges two StringJoiner objects excluding of prefix and suffix of second StringJoiner object.

1. // importing StringJoiner class
2. **import** java.util.StringJoiner;
3. **public** **class** StringJoinerExample {
4. **public** **static** **void** main(String[] args) {
6. StringJoiner joinNames = **new** StringJoiner(",", "[", "]");   // passing comma(,) and square-brackets as delimiter
8. // Adding values to StringJoiner
9. joinNames.add("Rahul");
10. joinNames.add("Raju");
12. // Creating StringJoiner with :(colon) delimiter
13. StringJoiner joinNames2 = **new** StringJoiner(":", "[", "]");  // passing colon(:) and square-brackets as delimiter
15. // Adding values to StringJoiner
16. joinNames2.add("Peter");
17. joinNames2.add("Raheem");
19. // Merging two StringJoiner
20. StringJoiner merge = joinNames.merge(joinNames2);
21. System.out.println(merge);
22. }
23. }

Output:

[Rahul,Raju,Peter:Raheem]

### **StringJoiner Example: StringJoiner Methods**

1. // importing StringJoiner class
2. **import** java.util.StringJoiner;
3. **public** **class** StringJoinerExample {
4. **public** **static** **void** main(String[] args) {
5. StringJoiner joinNames = **new** StringJoiner(","); // passing comma(,) as delimiter
7. // Prints nothing because it is empty
8. System.out.println(joinNames);
10. // We can set default empty value.
11. joinNames.setEmptyValue("It is empty");
12. System.out.println(joinNames);

15. // Adding values to StringJoiner
16. joinNames.add("Rahul");
17. joinNames.add("Raju");
18. System.out.println(joinNames);
20. // Returns length of StringJoiner
21. **int** length = joinNames.length();
22. System.out.println("Length: "+length);
24. // Returns StringJoiner as String type
25. String str = joinNames.toString();
26. System.out.println(str);
28. // Now, we can apply String methods on it
29. **char** ch = str.charAt(3);
30. System.out.println("Character at index 3: "+ch);
32. // Adding one more element
33. joinNames.add("Sorabh");
34. System.out.println(joinNames);
36. // Returns length
37. **int** newLength = joinNames.length();
38. System.out.println("New Length: "+newLength);
39. }
40. }

Output:

It is empty

Rahul,Raju

Length: 10

Rahul,Raju

Character at index 3: u

Rahul,Raju,Sorabh

New Length: 17

# **Java Optional Class**

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

## Java Optional Class Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static <T> Optional<T> empty() | It returns an empty Optional object. No value is present for this Optional. |
| public static <T> Optional<T> of(T value) | It returns an Optional with the specified present non-null value. |
| public static <T> Optional<T> ofNullable(T value) | It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional. |
| public T get() | If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException. |
| public boolean isPresent() | It returns true if there is a value present, otherwise false. |
| public void ifPresent(Consumer<? super T> consumer) | If a value is present, invoke the specified consumer with the value, otherwise do nothing. |
| public Optional<T> filter(Predicate<? super T> predicate) | If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional. |
| public <U> Optional<U> map(Function<? super T,? extends U> mapper) | If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. Otherwise return an empty Optional. |
| public <U> Optional<U> flatMap(Function<? super T,Optional<U> mapper) | If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. |
| public T orElse(T other) | It returns the value if present, otherwise returns other. |
| public T orElseGet(Supplier<? extends T> other) | It returns the value if present, otherwise invoke other and return the result of that invocation. |
| public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X extends Throwable | It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier. |
| public boolean equals(Object obj) | Indicates whether some other object is "equal to" this Optional or not. The other object is considered equal if:   * It is also an Optional and; * Both instances have no value present or; * the present values are "equal to" each other via equals(). |
| public int hashCode() | It returns the hash code value of the present value, if any, or returns 0 (zero) if no value is present. |
| public String toString() | It returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions. |

### **Example: Java Program without using Optional**

In the following example, we are not using Optional class. This program terminates abnormally and throws a nullPointerException.

1. **public** **class** OptionalExample {
2. **public** **static** **void** main(String[] args) {
3. String[] str = **new** String[10];
4. String lowercaseString = str[5].toLowerCase();
5. System.out.print(lowercaseString);
6. }
7. }

Output:

Exception in thread "main" java.lang.NullPointerException

at lambdaExample.OptionalExample.main(OptionalExample.java:6)

To avoid the abnormal termination, we use Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

### **Java Optional Example: If Value is not Present**

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. Optional<String> checkNull = Optional.ofNullable(str[5]);
6. **if**(checkNull.isPresent()){  // check for value is present or not
7. String lowercaseString = str[5].toLowerCase();
8. System.out.print(lowercaseString);
9. }**else**
10. System.out.println("string value is not present");
11. }
12. }

Output:

string value is not present

### **Java Optional Example: If Value is Present**

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";// Setting value for 5th index
6. Optional<String> checkNull = Optional.ofNullable(str[5]);
7. **if**(checkNull.isPresent()){  // It Checks, value is present or not
8. String lowercaseString = str[5].toLowerCase();
9. System.out.print(lowercaseString);
10. }**else**
11. System.out.println("String value is not present");
12. }
13. }

Output:

java optional class example

### **Another Java Optional Example**

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
6. Optional<String> checkNull = Optional.ofNullable(str[5]);
7. checkNull.ifPresent(System.out::println);   // printing value by using method reference
8. System.out.println(checkNull.get());    // printing value by using get method
9. System.out.println(str[5].toLowerCase());
10. }
11. }

Output:

JAVA OPTIONAL CLASS EXAMPLE

JAVA OPTIONAL CLASS EXAMPLE

java optional class example

### **Java Optional Methods Example**

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
6. // It returns an empty instance of Optional class
7. Optional<String> empty = Optional.empty();
8. System.out.println(empty);
9. // It returns a non-empty Optional
10. Optional<String> value = Optional.of(str[5]);
11. // If value is present, it returns an Optional otherwise returns an empty Optional
12. System.out.println("Filtered value: "+value.filter((s)->s.equals("Abc")));
13. System.out.println("Filtered value: "+value.filter((s)->s.equals("JAVA OPTIONAL CLASS EXAMPLE")));
14. // It returns value of an Optional. if value is not present, it throws an NoSuchElementException
15. System.out.println("Getting value: "+value.get());
16. // It returns hashCode of the value
17. System.out.println("Getting hashCode: "+value.hashCode());
18. // It returns true if value is present, otherwise false
19. System.out.println("Is value present: "+value.isPresent());
20. // It returns non-empty Optional if value is present, otherwise returns an empty Optional
21. System.out.println("Nullable Optional: "+Optional.ofNullable(str[5]));
22. // It returns value if available, otherwise returns specified value,
23. System.out.println("orElse: "+value.orElse("Value is not present"));
24. System.out.println("orElse: "+empty.orElse("Value is not present"));
25. value.ifPresent(System.out::println);   // printing value by using method reference
26. }
27. }

Output:

Optional.empty

Filtered value: Optional.empty

Filtered value: Optional[JAVA OPTIONAL CLASS EXAMPLE]

Getting value: JAVA OPTIONAL CLASS EXAMPLE

Getting hashCode: -619947648

Is value present: true

Nullable Optional: Optional[JAVA OPTIONAL CLASS EXAMPLE]

orElse: JAVA OPTIONAL CLASS EXAMPLE

orElse: Value is not present

JAVA OPTIONAL CLASS EXAMPLE

# **Java Nashorn**

Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool jjs which is used to execute JavaScript code.

You can execute JavaScript code by using jjs command-line tool and by embedding into Java source code.

### **Example: Executing by Using Terminal**

Following is the step by step process to execute JavaScript code at the JVM.

1) Create a file hello.js.

2) Write and save the following code into the file.

1. var hello = function(){
2. print("Hello Nashorn");
3. };
4. hello();

3) Open terminal

4) Write command **jjs hello.js** and press enter.

After executing command, you will see the below output.

Output:

Hello Nashorn

### **Example: Executing JavaScript file in Java Code**

You can execute JavaScript file directly from your Java file. In the following code, we are reading a file hello.js with the help of FileReader class.

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. // Reading Nashorn file
8. ee.eval(**new** FileReader("js/hello.js"));
9. }
10. }

Output:

Hello Nashorn

### **Example: Embedding JavaScript Code in Java Source File**

You can embed your JavaScript code in Java source file. Java compiler will not complaint but it is not good practice when you have large source code. In the following example, we are evaluating JavaScript code.

1. **import** javax.script.\*;
2. **public** **class** NashornExample {
3. **public** **static** **void** main(String[] args) **throws** Exception{
4. // Creating script engine
5. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
6. // Evaluating Nashorn code
7. ee.eval("print('Hello Nashorn');");
8. }
9. }

Output:

Hello Nashorn

### **Example: Embedding JavaScript Expression**

You can embed JavaScript expressions and variables in JavaScript code. In the following code we are embedding a variable to string. To execute this program you need to pass a flag -scripting in command-line.

File: hello.js

1. var hello = function(msg){
2. print("Hello ${msg}");
3. };
4. hello("Nashron");

**Command:** jjs -scripting hello.js

Output:

Hello Nashorn

## Heredocs

In Nashorn, heredocs are simply multi-line strings. You can create it with << followed by a special termination marker, which is EOF. You can also embed JavaScript expressions in ${...} expressions.

### **Example : Heredocs in JavaScript File**

file: hello.js

1. var message = <<EOF
2. This is a java script file
3. it contains multiple lines
4. of code.
5. let's execute.
6. EOF
7. print(message)

**Command:** jjs -scripting hello.js

Output:

This is a java script file

it contains multiple lines

of code.

let's execute.

### **Example: Setting JavaScript variable in Java File**

You can pass value to JavaScript variable in the Java file. In the followed example, we are binding and passing variable to JavaScript file.

File: hello.js

1. print("Hello "+name);

File: NashornExample.java

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. //Binding script and Define scope of script
8. Bindings bind = ee.getBindings(ScriptContext.ENGINE\_SCOPE);
9. bind.put("name", "Nashorn");
10. // Reading Nashorn file
11. ee.eval(**new** FileReader("js/hello.js"));
12. }
13. }

Output:

Hello Nashorn

## Import Java Package in JavaScript File

Java provides a facility to import Java package inside the JavaScript code. Here, we are using two approaches to import Java packages.

### **Example1: Import Java Package in JavaScript File**

File: hello.js

1. print(java.lang.Math.sqrt(4));

Output:

2

### **Example2: Import Java Package in JavaScript File**

File: hello.js

1. var importFile = **new** JavaImporter(java.util);
2. var a = **new** importFile.ArrayList();
3. a.add(12);
4. a.add(20);
5. print(a);
6. print(a.getClass());

Output:

[12, 20]

class java.util.ArrayList

### **Example3: Import Java Package in JavaScript File**

you can import multiple packages at the same time.

File: hello.js

1. var importIt = **new** JavaImporter(java.lang.String,java.util,java.io);
2. with (importIt) {
3. var linkedHS = **new** LinkedHashSet();
4. linkedHS.add(**new** File("abc"));
5. linkedHS.add(**new** File("hello.js"));
6. linkedHS.add("india".toUpperCase());
7. }
8. print(linkedHS);

Output:

[abc, hello.js, INDIA]

## Calling JavaScript function inside Java code

You can call JavaScript function inside the Java file. In the followed example, we are calling JavaScript functions.

### **Example: Calling function inside Java code**

File: hello.js

1. var functionDemo1 = function(){
2. print("This is JavaScript function");
3. }
4. var functionDemo2 = function(message){
5. print("Hello "+message);
6. }

File: NashornExample.java

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. // Reading Nashorn file
8. ee.eval(**new** FileReader("js/hello.js"));
9. Invocable invocable = (Invocable)ee;
10. // calling a function
11. invocable.invokeFunction("functionDemo1");
12. // calling a function and passing variable as well.
13. invocable.invokeFunction("functionDemo2","Nashorn");
14. }
15. }

Output:

This is JavaScript function

Hello Nashorn

# **Java Parallel Array Sorting**

Java provides a new additional feature in Array class which is used to sort array elements parallel.New methods has added to java.util.Arrays package that use the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays in parallel.The methods are called parallelSort() and are overloaded for all the primitive data types and Comparable objects.

The following table contains Arrays overloaded sorting methods.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static void parallelSort(byte[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(byte[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(char[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(char[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(double[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(double[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(float[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(float[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(int[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(int[] a,int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(long[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(long[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(short[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(short[] a,int fromIndex,int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static <T extends Comparable<? super T>> void parallelSort(T[] a) | Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array). |
| public static <T7gt; void parallelSort(T[] a,Comparator<? super T> cmp) | It sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array). |
| public static <T extends Comparable<? super T>> void parallelSort(T[] a,int fromIndex, int toIndex) | It sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be sorted is empty.) All elements in this range must implement the Comparable interface. Furthermore, all elements in this range must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array). |
| public static <T> void parallelSort(T[] a, int fromIndex, int toIndex, Comparator<? super T> cmp) | It sorts the specified range of the specified array of objects according to the order induced by the specified comparator. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be sorted is empty.) All elements in the range must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the range). |

### **Java Parallel Array Sorting Example**

1. **import** java.util.Arrays;
2. **public** **class** ParallelArraySorting {
3. **public** **static** **void** main(String[] args) {
4. // Creating an integer array
5. **int**[] arr = {5,8,1,0,6,9};
6. // Iterating array elements
7. **for** (**int** i : arr) {
8. System.out.print(i+" ");
9. }
10. // Sorting array elements parallel
11. Arrays.parallelSort(arr);
12. System.out.println("\nArray elements after sorting");
13. // Iterating array elements
14. **for** (**int** i : arr) {
15. System.out.print(i+" ");
16. }
17. }
18. }

Output:

5 8 1 0 6 9

Array elements after sorting

0 1 5 6 8 9

## Java Parallel Array Sorting Example: Passing Start and End Index

In the following example, we are passing starting and end index of the array. The first index is inclusive and end index is exclusive i.e. if we pass 0 as start index and 4 as end index, only 0 to 3 index elements will be sorted.

It throws IllegalArgumentException if start index > end index.

It throws ArrayIndexOutOfBoundsException if start index < 0 or end index > a.length.

1. **import** java.util.Arrays;
2. **public** **class** ParallelArraySorting {
3. **public** **static** **void** main(String[] args) {
4. // Creating an integer array
5. **int**[] arr = {5,8,1,0,6,9,50,-3};
6. // Iterating array elements
7. **for** (**int** i : arr) {
8. System.out.print(i+" ");
9. }
10. // Sorting array elements parallel and passing start, end index
11. Arrays.parallelSort(arr,0,4);
12. System.out.println("\nArray elements after sorting");
13. // Iterating array elements
14. **for** (**int** i : arr) {
15. System.out.print(i+" ");
16. }
17. }
18. }

Output:

5 8 1 0 6 9 50 -3

Array elements after sorting

0 1 5 8 6 9 50 -3

# **Java Type Inference**

Type inference is a feature of Java which provides ability to compiler to look at each method invocation and corresponding declaration to determine the type of arguments.

Java provides improved version of type inference in Java 8. the following example explains, how we can use type inference in our code:

Here, we are creating arraylist by mentioning integer type explicitly at both side. The following approach is used earlier versions of Java.

1. List<Integer> list = **new** ArrayList<Integer>();

In the following declaration, we are mentioning type of arraylist at one side. This approach was introduce in Java 7. Here, you can left second side as blank diamond and compiler will infer type of it by type of reference variable.

1. List<Integer> list2 = **new** ArrayList<>();

### **Improved Type Inference**

In Java 8, you can call specialized method without explicitly mentioning of type of arguments.

1. showList(**new** ArrayList<>());

## Java Type Inference Example

You can use type inference with generic classes and methods.

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** TypeInferenceExample {
4. **public** **static** **void** showList(List<Integer>list){
5. **if**(!list.isEmpty()){
6. list.forEach(System.out::println);
7. }**else** System.out.println("list is empty");
8. }
9. **public** **static** **void** main(String[] args) {
10. // An old approach(prior to Java 7) to create a list
11. List<Integer> list1 = **new** ArrayList<Integer>();
12. list1.add(11);
13. showList(list1);
14. // Java 7
15. List<Integer> list2 = **new** ArrayList<>(); // You can left it blank, compiler can infer type
16. list2.add(12);
17. showList(list2);
18. // Compiler infers type of ArrayList, in Java 8
19. showList(**new** ArrayList<>());
20. }
21. }

Output:

11

12

list is empty

You can also create your own custom generic class and methods. In the following example, we are creating our own generic class and method.

## Java Type Inference Example 2

1. **class** GenericClass<X> {
2. X name;
3. **public** **void** setName(X name){
4. **this**.name = name;
5. }
6. **public** X getName(){
7. returnname;
8. }
9. **public** String genericMethod(GenericClass<String> x){
10. x.setName("John");
11. returnx.name;
12. }
13. }
15. **public** **class** TypeInferenceExample {
16. **public** **static** **void** main(String[] args) {
17. GenericClass<String> genericClass = **new** GenericClass<String>();
18. genericClass.setName("Peter");
19. System.out.println(genericClass.getName());
21. GenericClass<String> genericClass2 = **new** GenericClass<>();
22. genericClass2.setName("peter");
23. System.out.println(genericClass2.getName());
25. // New improved type inference
26. System.out.println(genericClass2.genericMethod(**new** GenericClass<>()));
27. }
28. }

Output:

Peter

peter

John

# **Method Parameter Reflection**

Java provides a new feature in which you can get the names of formal parameters of any method or constructor. The java.lang.reflect package contains all the required classes like Method and Parameter to work with parameter reflection.

### **Method class**

It provides information about single method on a class or interface. The reflected method may be a class method or an instance method.

### **Method Class methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| public boolean equals(Object obj) | It compares this Method against the specified object. It returns true if the objects are the same. Two Methods are the same if they were declared by the same class and have the same name and formal parameter types and return type. |
| public AnnotatedType getAnnotatedReturnType() | It returns an AnnotatedType object that represents the use of a type to specify the return type of the method/constructor. |
| public <T extends Annotation> T getAnnotation(Class<T> annotationClass) | It returns this element's annotation for the specified type if such an annotation is present otherwise returns null. NullPointerException - if the given annotation class is null |
| public Annotation[] getDeclaredAnnotations() | It returns annotations that are directly present on this element. This method ignores inherited annotations. If there are no annotations directly present on this element, the return value is an array of length 0. The caller of this method is free to modify the returned array. it will have no effect on the arrays returned to other callers. |
| public Class<?> getDeclaringClass() | It returns the Class object representing the class or interface that declares the executable represented by this object. |
| public Object getDefaultValue() | It returns the default value for the annotation member represented by this Method instance. |
| public Class<?>[] getExceptionTypes() | It returns an array of Class objects that represent the types of exceptions declared to be thrown by the underlying executable represented by this object. |
| public Type[] getGenericExceptionTypes() | It returns an array of Type objects that represent the exceptions declared to be thrown by this executable object. It returns an array of length 0 if the underlying executable declares no exceptions in its throws clause. It throws following exceptions: **GenericSignatureFormatError** - if the generic method signature does not conform to the format specified in The Java Virtual Machine Specification. **TypeNotPresentException** - if the underlying executable's throws clause refers to a non-existent type declaration. **MalformedParameterizedTypeException** - if the underlying executable's throws clause refers to a parameterized type that cannot be instantiated for any reason. |
| public Type[] getGenericParameterTypes() | It returns an array of Type objects that represent the formal parameter types. It throws following exceptions: **GenericSignatureFormatError** - if the generic method signature does not conform to the format specified in The Java Virtual Machine Specification. **TypeNotPresentException** - if any of the parameter types of the underlying executable refers to a non-existent type declaration.**MalformedParameterizedTypeException** - if any of the underlying executable's parameter types refer to a parameterized type that cannot be instantiated for any reason. |
| public int getModifiers() | It returns the Java language modifiers for the executable represented by this object. |
| public String getName() | It returns the name of the method represented by this Method object as a String. |
| public Annotation[][] getParameterAnnotations() | It returns an array of arrays that represent the annotations on the formal and implicit parameters, in declaration order, of the executable represented by this object. |
| public int getParameterCount() | It returns the number of formal parameters for the executable represented by this object. |
| public Class<?>[] getParameterTypes() | It returns an array of Class objects that represent the formal parameter types. in declaration order, of the executable represented by this object. It returns an array of length 0 if the underlying executable takes no parameters. |
| public Class<?> getReturnType() | It returns a Class object that represents the formal return type of the method represented by this Method object. |
| public TypeVariable<Method>[] getTypeParameters() | It returns an array of TypeVariable objects that represent the type variables declared by the generic declaration represented by this GenericDeclaration object, in declaration order. It throws GenericSignatureFormatError, if the generic signature of this generic declaration does not conform to the format specified in The Java Virtual Machine Specification |
| public int hashCode() | It returns a hashcode for this Method. The hashcode is computed as the exclusive-or of the hashcodes for the underlying method's declaring class name and the method's name. |
| public Object invoke(Object obj, Object... args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException | It invokes the underlying method represented by this Method object, on the specified object with the specified parameters. If the underlying method is static, the specified obj argument is ignored. It may be null. If the number of formal parameters required by the underlying method is 0, the supplied args array may be of length 0 or null. If the underlying method is an instance method, it is invoked using dynamic method lookup as documented in The Java Language Specification. If the underlying method is static, the class that declared the method is initialized if it has not already been initialized. If the method completes normally, the value it returns is returned to the caller of invoke. |
| public boolean isBridge() | It returns true if this method is a bridge method. otherwise returns false. |
| public boolean isDefault() | It returns true if this method is a default method otherwise returns false. A default method is a public non-abstract instance method, that is, a non-static method with a body, declared in an interface type. |
| public boolean isSynthetic() | It returns true if this executable is a synthetic construct; returns false otherwise. |
| public boolean isVarArgs() | It returns true if this executable was declared to take a variable number of arguments; returns false otherwise. |
| public String toGenericString() | It returns a string describing this Method, including type parameters. |
| public String toString() | It returns a string. |

## Parameter class

Parameter class provides information about method parameters, including its name and modifiers. It also provides an alternate means of obtaining attributes for the parameter.

### **Parameter Methods**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public boolean equals(Object obj) | It compares based on the executable and the index. |
| public AnnotatedType getAnnotatedType() | It returns an AnnotatedType object that represents the use of a type to specify the type of the formal parameter represented by this Parameter. |
| public <T extends Annotation> T getAnnotation(Class<T> annotationClass) | It returns this element's annotation for the specified type if such an annotation is present, else null. It throws NullPointerException, if the given annotation class is null. |
| public Annotation[] getAnnotations() | It returns annotations that are present on this element. If there are no annotations present on this element, the return value is an array of length 0. |
| public <T extends Annotation> T[] getAnnotationsByType(Class<T> annotationClass) | It returns annotations that are associated with this element. If there are no annotations associated with this element, the return value is an array of length 0. The difference between this method and AnnotatedElement.getAnnotation(Class) is that this method detects if its argument is a repeatable annotation type (JLS 9.6), and if so, attempts to find one or more annotations of that type by "looking through" a container annotation. It throws NullPointerException, if the given annotation class is null. |
| public <T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass) | It returns this element's annotation for the specified type if such an annotation is directly present, else null. This method ignores inherited annotations. It throws NullPointerException, if the given annotation class is null. |
| public Annotation[] getDeclaredAnnotations() | It returns annotations that are directly present on this element. This method ignores inherited annotations. If there are no annotations directly present on this element, the return value is an array of length 0. |
| public <T extends Annotation> T[] getDeclaredAnnotationsByType(Class<T> annotationClass) | It returns this element's annotations for the specified type if such annotations are either directly present or indirectly present. This method ignores inherited annotations. If there are no specified annotations directly or indirectly present on this element, the return value is an array of length 0. The difference between this method and AnnotatedElement.getDeclaredAnnotation(Class) is that this method detects if its argument is a repeatable annotation type (JLS 9.6), and if so, attempts to find one or more annotations of that type by "looking through" a container annotation if one is present. The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers. It throws NullPointerException, if the given annotation class is null |
| public Executable getDeclaringExecutable() | It returns the Executable which declares this parameter. |
| public int getModifiers() | It returns the modifier flags for the parameter represented by this Parameter object. |
| public String getName() | It returns the name of the parameter. If the parameter's name is present, this method returns the name provided by the class file. Otherwise, this method synthesizes a name of the form argN, where N is the index of the parameter in the descriptor of the method which declares the parameter. |
| public Type getParameterizedType() | It returns a Type object that identifies the parameterized type for the parameter represented by this Parameter object. |
| public Class<?> getType() | It returns a Class object that identifies the declared type for the parameter represented by this Parameter object. |
| public int hashCode()mul int arg0 int arg1 add int arg0 int arg1 | It returns a hash code based on the executable's hash code and the index. |
| public boolean isImplicit() | It returns true if this parameter is implicitly declared in source code. Otherwise, returns false. |
| public boolean isNamePresent() | It returns true if the parameter has a name according to the class file, otherwise, returns false. |
| public boolean isSynthetic() | It returns true if this parameter is neither implicitly nor explicitly declared in source code. Otherwise returns false. |
| public boolean isVarArgs() | It returns true if this parameter represents a variable argument list; returns false otherwise. |
| public String toString() | It returns a string describing this parameter. The format is the modifiers for the parameter, if any, in canonical order as recommended by The Java? Language Specification. |

## Java Method Parameter Reflection Example

*File: Calculate.java*

1. **public** **class** Calculate {
2. **int** add(**int** a, **int** b){
3. **return** (a+b);
4. }
5. **int** mul(**int** a, **int** b){
6. **return** (b\*a);
7. }
8. }

#### Note - before compiling and executing below code, first compile Calculate class by following command:

1. javac -parameters Calculate.java

**-parameter** flag in the above command is used to store parameters in the Calculate class file. By default .class does not store parameters and returns argsN as parameter name, where N is a number of parameters in the method.

*File: ParameterReflection.java*

1. **import** java.lang.reflect.Method;
2. **import** java.lang.reflect.Parameter;
3. **public** **class** ParameterReflection {
4. **public** **static** **void** main(String[] args) {
5. // Creating object of a class
6. Calculate calculate = **new** Calculate();
7. // instantiating Class class
8. Classcls = calculate.getClass();
9. // Getting declared methods inside the Calculate class
10. Method[] method = cls.getDeclaredMethods(); // It returns array of methods
11. // Iterating method array
12. **for** (Method method2 : method) {
13. System.out.print(method2.getName());    // getting name of method
14. // Getting parameters of each method
15. Parameter parameter[] = method2.getParameters(); // It returns array of parameters
16. // Iterating parameter array
17. **for** (Parameter parameter2 : parameter) {
18. System.out.print(""+parameter2.getParameterizedType()); // returns type of parameter
19. System.out.print(""+parameter2.getName()); // returns parameter name
20. }
21. System.out.println();
22. }
23. }
24. }

Output:

mul int a int b

add int a int b

Above code will produce the below output if you don't use **-parameters** flag to compile the Calculate.java file.

Output:

mul int arg0 int arg1

add int arg0 int arg1

# **Java Type and Repeating Annotations**

## Java Type Annotations

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be applied to any type use. It means that annotations can be used anywhere you use a type. For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this:

1. @NonNull String str;

Following are the examples of type annotations:

1. @NonNull List<String>
2. List<@NonNull String> str
3. Arrays<@NonNegative Integer> sort
4. @Encrypted File file
5. @Open Connection connection
6. **void** divideInteger(**int** a, **int** b) **throws** @ZeroDivisor ArithmeticException

#### Note - Java created type annotations to support improved analysis of Java programs. It supports way of ensuring stronger type checking.

## Java Repeating Annotations

In Java 8 release, Java allows you to repeating annotations in your source code. It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

1. Declare a repeatable annotation type
2. Declare the containing annotation type

### **1) Declare a repeatable annotation type**

Declaring of repeatable annotation type must be marked with the @Repeatable meta-annotation. In the following example, we have defined a custom @Game repeatable annotation type.

1. @Repeatable(Games.**class**)
2. @interfaceGame{
3. String name();
4. String day();
5. }

The value of the @Repeatable meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In the following example, the containing annotation type is Games. So, repeating @Game annotations is stored in an @Games annotation.

### **2) Declare the containing annotation type**

Containing annotation type must have a value element with an array type. The component type of the array type must be the repeatable annotation type. In the following example, we are declaring Games containing annotation type:

1. @interfaceGames{
2. Game[] value();
3. }

#### Note - Compiler will throw a compile-time error, if you apply the same annotation to a declaration without first declaring it as repeatable.

## Java Repeating Annotations Example

1. // Importing required packages for repeating annotation
2. **import** java.lang.annotation.Repeatable;
3. **import** java.lang.annotation.Retention;
4. **import** java.lang.annotation.RetentionPolicy;
5. // Declaring repeatable annotation type
6. @Repeatable(Games.**class**)
7. @interfaceGame{
8. String name();
9. String day();
10. }
11. // Declaring container for repeatable annotation type
12. @Retention(RetentionPolicy.RUNTIME)
13. @interfaceGames{
14. Game[] value();
15. }
16. // Repeating annotation
17. @Game(name = "Cricket",  day = "Sunday")
18. @Game(name = "Hockey",   day = "Friday")
19. @Game(name = "Football", day = "Saturday")
20. **public** **class** RepeatingAnnotationsExample {
21. **public** **static** **void** main(String[] args) {
22. // Getting annotation by type into an array
23. Game[] game = RepeatingAnnotationsExample.**class**.getAnnotationsByType(Game.**class**);
24. **for** (Gamegame2 : game) {    // Iterating values
25. System.out.println(game2.name()+" on "+game2.day());
26. }
27. }
28. }

OUTPUT:

Cricket on Sunday

Hockey on Friday

Football on Saturday

# **Java 8 JDBC Improvements**

In Java 8, Java made two major changes in JDBC API.

#### **1) The JDBC-ODBC Bridge has been removed.**

Oracle does not support the JDBC-ODBC Bridge. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

#### **2) Added some new features in JDBC 4.2.**

Java JDBC 4.2 introduces the following features:

* Addition of REF\_CURSOR support.
* Addition of java.sql.DriverAction Interface
* Addition of security check on deregisterDriver Method in DriverManager Class
* Addition of the java.sql.SQLType Interface
* Addition of the java.sql.JDBCType Enum
* Add Support for large update counts
* Changes to the existing interfaces
* Rowset 1.2: Lists the enhancements for JDBC RowSet.

## Java JDBC DriverAction

It is an interface that must be implemented when a Driver wants to be notified by DriverManager. It is added in java.sql package and contains only one abstract method.

### **DriverAction Method**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void deregister() | This method called by DriverManager.deregisterDriver(Driver) to notify the JDBC driver that it was de-registered. |

The deregister method is intended only to be used by JDBC Drivers and not by applications.

JDBC drivers are recommended not to implement the DriverAction in a public class.

If there are active connections to the database at the time that the deregister method is called, it is implementation specific as to whether the connections are closed or allowed to continue. Once this method is called, it is implementation specific as to whether the driver may limit the ability to create new connections to the database, invoke other Driver methods or throw a SQLException.

## Java JDBC4.2 DriverAction Example

1. **import** java.sql.\*;
2. // implementing DriverAction interface
3. **class** JdbcExample **implements** DriverAction{
4. // implementing deregister method of DriverAction interface
5. @Override
6. **public** **void** deregister() {
7. System.out.println("Driver deregistered");
8. }
9. **public** **static** **void** main(String args[]){
10. **try**{
11. // Creating driver instance
12. Driver driver = **new** com.mysql.jdbc.Driver();
13. // Creating Action Driver
14. DriverAction da = **new** JdbcExample();
15. // Registering driver by passing driver and driverAction
16. DriverManager.registerDriver(driver, da);
17. // Creating connection
18. Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","mysql");
19. //Here student is database name, root is username and password is mysql
20. Statement stmt=con.createStatement();
21. // Executing SQL query
22. ResultSet rs=stmt.executeQuery("select \* from user");
23. **while**(rs.next()){
24. System.out.println(rs.getInt(1)+""+rs.getString(2)+""+rs.getString(3));
25. }
26. // Closing connection
27. con.close();
28. // Calling deregisterDriver method
29. DriverManager.deregisterDriver(driver);
30. }**catch**(Exception e){ System.out.println(e);}
31. }
33. }

Output:

1 Arun 25

2 irfan 22

3 Neraj kumar 25

Driver deregistered

## Java JDBC SQLType

This interface is used to identify a generic SQL type, JDBC type or a vendor specific data type.

It provides following methods.

|  |  |
| --- | --- |
| **Method** | **Description** |
| String getName() | It returns the SQLType name that represents a SQL data type. |
| String getVendor() | It returns the name of the vendor that supports this data type. The value returned typically is the package name for this vendor. |
| Integer getVendorTypeNumber() | It returns the vendor specific type number for the data type. |

## Java JDBCType

It is an Enumeration which defines the constants that are used to identify generic SQL types, called JDBC types. It extends java.lang.Enum and implements java.sql.SQLType.

## JDBCType Fields

The following table contains constants defined in the JDBCType.

|  |  |
| --- | --- |
| **Enum constant** | **Description** |
| public static final JDBCType ARRAY | It identifies the generic SQL type ARRAY. |
| public static final JDBCType BIGINT | It identifies the generic SQL type BIGINT. |
| public static final JDBCType BIT | It identifies the generic SQL type BIT. |
| public static final JDBCType BLOB | It identifies the generic SQL type BLOB. |
| public static final JDBCType BOOLEAN | It identifies the generic SQL type BOOLEAN. |
| public static final JDBCType CHAR | It identifies the generic SQL type CHAR. |
| public static final JDBCType CLOB | It identifies the generic SQL type CLOB. |
| public static final JDBCType DATALINK | It identifies the generic SQL type DATALINK. |
| public static final JDBCType DATE | It identifies the generic SQL type DATE. |
| public static final JDBCType DECIMAL | It identifies the generic SQL type DECIMAL. |
| public static final JDBCType DISTINCT | It identifies the generic SQL type DISTINCT. |
| public static final JDBCType DOUBLE | It identifies the generic SQL type DOUBLE. |
| public static final JDBCType FLOAT | It identifies the generic SQL type FLOAT. |
| public static final JDBCType INTEGER | It identifies the generic SQL type INTEGER. |
| public static final JDBCType JAVA\_OBJECT | It indicates that the SQL type is database-specific and gets mapped to a Java object that can be accessed via the methods getObject and setObject. |
| Public static final JDBCType LONGNVARCHAR | It identifies the generic SQL type LONGNVARCHAR. |
| public static final JDBCType NCHAR | It identifies the generic SQL type NCHAR. |
| public static final JDBCType NCLOB | It identifies the generic SQL type NCLOB. |
| public static final JDBCType NULL | It identifies the generic SQL value NULL. |
| public static final JDBCType NUMERIC | It identifies the generic SQL type NUMERIC. |
| public static final JDBCType NVARCHAR | It identifies the generic SQL type NVARCHAR. |
| public static final JDBCType OTHER | It indicates that the SQL type is database-specific and gets mapped to a Java object that can be accessed via the methods getObject and setObject. |
| public static final JDBCType REAL | It identifies the generic SQL type REAL.Identifies the generic SQL type VARCHAR. |
| public static final JDBCType REF | It identifies the generic SQL type REF. |
| public static final JDBCType REF\_CURSOR | It identifies the generic SQL type REF\_CURSOR. |
| public static final JDBCType ROWID | It identifies the SQL type ROWID. |
| public static final JDBCType SMALLINT | It identifies the generic SQL type SMALLINT. |
| public static final JDBCType SQLXML | It identifies the generic SQL type SQLXML. |
| public static final JDBCType STRUCT | It identifies the generic SQL type STRUCT. |
| public static final JDBCType TIME | It identifies the generic SQL type TIME. |
| public static final JDBCType TIME\_WITH\_TIMEZONE | It identifies the generic SQL type TIME\_WITH\_TIMEZONE. |
| public static final JDBCType TIMESTAMP | It identifies the generic SQL type TIMESTAMP. |
| public static final JDBCType TIMESTAMP\_WITH\_TIMEZONE | It identifies the generic SQL type TIMESTAMP\_WITH\_TIMEZONE. |
| public static final JDBCType TINYINT | It identifies the generic SQL type TINYINT. |
| public static final JDBCType VARBINARY | It identifies the generic SQL type VARBINARY. |
| public static final JDBCType VARCHAR | It identifies the generic SQL type VARCHAR. |

## JDBCType Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| public String getName() | It returns the SQLType name that represents a SQL data type. |
| public String getVendor() | It returns the name of the vendor that supports this data type. |
| public Integer getVendorTypeNumber() | It returns the vendor specific type number for the data type. |
| public static JDBCType valueOf(int type) | It returns the JDBCType that corresponds to the specified Types value. It throws IllegalArgumentException, if this enum type has no constant with the specified Types value. |
| public static JDBCType valueOf(String name) | It returns the enum constant of this type with the specified name. The string must match exactly an identifier used to declare an enum constant in this type. It throws IllegalArgumentException, if this enum type has no constant with the specified name. It throws NullPointerException, if the argument is null. |
| public static JDBCType[] values() | It returns an array containing the constants of this enum type, in the order they are declared. This method may be used to iterate over the constants. |

# **Binary Literals**

Java added a new feature Binary Literal in Java 7. I allows you to express integral types (byte, short, int, and long) in binary number system. To specify a binary literal, add the prefix 0b or 0B to the integral value.

In the following example, we are creating binary literals from integral values.

## Binary Literal Example

1. **public** **class** BinaryLiteralsExample {
2. **public** **static** **void** main(String[] args) {
3. // Binary literal in byte type
4. **byte** b1 = 0b101;    // Using b0, The b can be lower or upper case
5. **byte** b2 = 0B101;    // Using B0
6. System.out.println("----------Binary Literal in Byte----------------");
7. System.out.println("b1 = "+b1);
8. System.out.println("b2 = "+b2);
10. // Binary literal in short type
11. **short** s1 = 0b101;   // Using b0, The b can be lower or upper case
12. **short** s2 = 0B101;   // Using B0
13. System.out.println("----------Binary Literal in Short----------------");
14. System.out.println("s1 = "+s1);
15. System.out.println("s2 = "+s2);
17. // Binary literal in int type
18. **int** i1 = 0b101;     // Using b0, The b can be lower or upper case
19. **int** i2 = 0B101;     // Using B0
20. System.out.println("----------Binary Literal in Integer----------------");
21. System.out.println("i1 = "+i1);
22. System.out.println("i2 = "+i2);
24. // Binary literal in long type
25. **long** l1 = 0b0000011111100001;   // Using b0, The b can be lower or upper case
26. **long** l2 = 0B0000011111100001;   // Using B0
27. System.out.println("----------Binary Literal in Long----------------");
28. System.out.println("l1 = "+l1);
29. System.out.println("l2 = "+l2);
30. }
31. }

Output:

----------Binary Literal in Byte----------------

b1 = 5

b2 = 5

----------Binary Literal in Short----------------

s1 = 5

s2 = 5

----------Binary Literal in Integer----------------

i1 = 5

i2 = 5

----------Binary Literal in Long----------------

l1 = 2017

l2 = 2017

## Binary Literal Example 2

In this example, we are creating negative binary, using underscore in binary literals and manipulating as well.

1. **public** **class** BinaryLiteralsExample {
2. **public** **static** **void** main(String[] args) {
3. **byte** b1 = 5; // a decimal value
4. // Using binary of 5
5. **byte** b2 = 0b101;    // using b0, The b can be lower or upper case
6. // Declaring negative binary
7. **byte** b3 = -0b101;
8. // Using underscore in binary literal
9. **byte** b4 = 0b101\_0;
10. System.out.println("b1 = "+b1);
11. System.out.println("b2 = "+b2);
12. System.out.println("b3 = "+b3);
13. System.out.println("b4 = "+b4);
14. // Check whether binary and decimal are equal
15. System.out.println("is b1 and b2 equal: "+(b1==b2));
16. // Perform operation on binary value
17. System.out.println("b2 + 1 = "+(b2+1));
18. // Perform operation on negative binary value
19. System.out.println("b3 + 1 = "+(b3+1));
20. System.out.println("b4 x 2 = "+(b4\*2));
21. }
22. }

Output:

b1 = 5

b2 = 5

b3 = -5

b4 = 10

is b1 and b2 equal: true

b2 + 1 = 6

b3 + 1 = -4

b4 x 2 = 20

# **String in Switch Statement**

In Java 7, Java allows you to use string objects in the expression of switch statement. In order to use string, you need to consider the following points:

* It must be only string object.
  1. Object game = "Hockey"; // It is not allowed
  2. String game = "Hockey"; // It is OK.
* String object is case sensitive.
  1. "Hickey" and "hocker" are not equal.
* No Null object

be careful while passing string object, passing a null object cause to NullPointerException.

## String in Switch Statement Example 1

1. **public** **class** StringInSwitchStatementExample {
2. **public** **static** **void** main(String[] args) {
3. String game = "Cricket";
4. **switch**(game){
5. **case** "Hockey":
6. System.out.println("Let's play Hockey");
7. **break**;
8. **case** "Cricket":
9. System.out.println("Let's play Cricket");
10. **break**;
11. **case** "Football":
12. System.out.println("Let's play Football");
13. }
14. }
15. }

Output:

Let's play Cricket

## String in Switch Statement Example 2

1. **public** **class** StringInSwitchStatementExample {
2. **public** **static** **void** main(String[] args) {
3. String game = "Card-Games";
4. **switch**(game){
5. **case** "Hockey": **case**"Cricket": **case**"Football":
6. System.out.println("This is a outdoor game");
7. **break**;
8. **case** "Chess": **case**"Card-Games": **case**"Puzzles": **case**"Indoor basketball":
9. System.out.println("This is a indoor game");
10. **break**;
11. **default**:
12. System.out.println("What game it is?");
13. }
14. }
15. }

Output:

This is a indoor game

# **Java 7 Catch Multiple Exceptions**

Java allows you to catch multiple type exceptions in a single catch block. It was introduced in Java 7 and helps to optimize code.

You can use vertical bar (|) to separate multiple exceptions in catch block.

An old, prior to Java 7 approach to handle multiple exceptions.

### **Catching Multiple Exception Types Example 1**

1. **public** **class** MultipleExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** array[] = newint[10];
5. array[10] = 30/0;
6. }
7. **catch**(ArithmeticException e){
8. System.out.println(e.getMessage());
9. }
10. **catch**(ArrayIndexOutOfBoundsException e){
11. System.out.println(e.getMessage());
12. }
13. **catch**(Exception e){
14. System.out.println(e.getMessage());
15. }
16. }
17. }

Output:

/ by zero

## Catching Multiple Exception Types Example 2

What Java 7 provides us:

1. **public** **class** MultipleExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** array[] = newint[10];
5. array[10] = 30/0;
6. }
7. **catch**(ArithmeticException | ArrayIndexOutOfBoundsException e){
8. System.out.println(e.getMessage());
9. }
10. }
11. }

Output:

/ by zero

## Catching Multiple Exception Types Example 3

1. **public** **class** MultipleExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** array[] = newint[10];
5. array[10] = 30/0;
6. }
7. **catch**(Exception | ArithmeticException | ArrayIndexOutOfBoundsException e){
8. System.out.println(e.getMessage());
9. }
10. }
11. }

Output:

Compile-time error: The exception ArithmeticException is already caught by the alternative Exception

So here, in case when your are catching multiple exceptions, follow the rule of generalized to more specialized. It means that, if you are using super (general) class, don't use child (specialized) class.

#### Note - Catch block which handles more than one exception type makes the catch parameter implicitly final. In the above example, the catch parameter "e" is final and therefore you cannot assign any value to it.

# **The try-with-resources statement**

In Java, the try-with-resources statement is a try statement that declares one or more resources. The resource is as an object that must be closed after finishing the program. The try-with-resources statement ensures that each resource is closed at the end of the statement execution.

You can pass any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable.

The following example writes a string into a file. It uses an instance of FileOutputStream to write data into the file. FileOutputStream is a resource that must be closed after the program is finished with it. So, in this example, closing of resource is done by itself try.

## Try-with-resources Example 1

1. **import** java.io.FileOutputStream;
2. **public** **class** TryWithResources {
3. **public** **static** **void** main(String args[]){
4. // Using try-with-resources
5. **try**(FileOutputStream fileOutputStream =newFileOutputStream("/java7-new-features/src/abc.txt")){
6. String msg = "Welcome to javaTpoint!";
7. **byte** byteArray[] = msg.getBytes(); //converting string into byte array
8. fileOutputStream.write(byteArray);
9. System.out.println("Message written to file successfuly!");
10. }**catch**(Exception exception){
11. System.out.println(exception);
12. }
13. }
14. }

Output:

Message written to file successfuly!

Output of file

Welcome to javaTpoint!

## Try-with-resources Example : Using Multiple Resources

1. **import** java.io.DataInputStream;
2. **import** java.io.FileInputStream;
3. **import** java.io.FileOutputStream;
4. **import** java.io.InputStream;
5. **public** **class** TryWithResources {
6. **public** **static** **void** main(String args[]){
7. // Using try-with-resources
8. **try**(    // Using multiple resources
9. FileOutputStream fileOutputStream =**new** FileOutputStream("/java7-new-features/src/abc.txt");
10. InputStream input = **new** FileInputStream("/java7-new-features/src/abc.txt")){
11. // -----------------------------Code to write data into file--------------------------------------------//
12. String msg = "Welcome to javaTpoint!";
13. **byte** byteArray[] = msg.getBytes();  // Converting string into byte array
14. fileOutputStream.write(byteArray);  // Writing  data into file
15. System.out.println("------------Data written into file--------------");
16. System.out.println(msg);
17. // -----------------------------Code to read data from file---------------------------------------------//
18. // Creating input stream instance
19. DataInputStream inst = **new** DataInputStream(input);
20. **int** data = input.available();
21. // Returns an estimate of the number of bytes that can be read from this input stream.
22. **byte**[] byteArray2 = **new** **byte**[data]; //
23. inst.read(byteArray2);
24. String str = **new** String(byteArray2); // passing byte array into String constructor
25. System.out.println("------------Data read from file--------------");
26. System.out.println(str); // display file data
27. }**catch**(Exception exception){
28. System.out.println(exception);
29. }
30. }
31. }

Output:

------------Data written into file--------------

Welcome to javaTpoint!

------------Data read from file--------------

Welcome to javaTpoint!

You can use catch and finally blocks with try-with-resources statement just like an ordinary try statement.

#### Note - In a try-with-resources statement, catch or finally block executes after closing of the declared resources.

## Try-with-resources Example: using finally block

1. **import** java.io.FileOutputStream;
2. **public** **class** TryWithResources {
3. **public** **static** **void** main(String args[]){
4. **try**(    FileOutputStream fileOutputStream=
5. **new** FileOutputStream("/home/irfan/scala-workspace/java7-new-features/src/abc.txt")){
6. // -----------------------------Code to write data into file--------------------------------------------//
7. String msg = "Welcome to javaTpoint!";
8. **byte** byteArray[] = msg.getBytes();  // Converting string into byte array
9. fileOutputStream.write(byteArray);  // Writing  data into file
10. System.out.println("Data written successfully!");
11. }**catch**(Exception exception){
12. System.out.println(exception);
13. }
14. **finally**{
15. System.out.println("Finally executes after closing of declared resources.");
16. }
17. }
18. }

Output:

Data written successfully!

Finally executes after closing of declared resources.

## Suppressed Exceptions

If a try block throws an exception and one or more exceptions are thrown by the try-with-resources, the exceptions thrown by try-with-resources are suppressed. In other words, we can say, exceptions which are thrown by try-with-resources are suppressed exceptions.

You can get these exceptions by using the getSuppress() method of Throwable class.

Java added a new constructor and two new methods in Throwable class to deal with suppressed exceptions.

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| protected Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) | It constructs a new throwable with the specified detail message, cause, suppression enabled or disabled, and writable stack trace enabled or disabled. |  |

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final void addSuppressed(Throwable exception)/td> | It appends the specified exception to the exceptions that were suppressed in order to deliver this exception. This method is thread-safe and typically called (automatically and implicitly) by the try-with-resources statement. It throws following exceptions: **IllegalArgumentException:** if exception is throwable, a throwable cannot suppress itself. **NullPointerException:** if exception is null. |  |
| public final Throwable[] getSuppressed() | It returns an array containing all of the exceptions that were suppressed by the try-with-resources statement. If no exceptions were suppressed or suppression is disabled, an empty array is returned. |  |

# **Type Inference for Generic Instance Creation**

In Java 7, Java provides improved compiler which is enough smart to infer the type of generic instance. Now, you can replace the type arguments with an empty set of type parameters (<>). This pair of angle brackets is informally called the diamond.

The following approach is used in Java 6 and prior version.

1. Ex. List<Integer> list  = **new** List<Integer>();

Now, you can use the following new approach introduced in Java 7.

1. Ex. List<Integer> list = **new** List<>(); // Here, we just used diamond

## Type Inference for Generic Instance Creation Example

1. **import** java.util.List;
2. **import** java.util.ArrayList;
3. **public** **class** TypeInference {
4. **public** **static** **void** main(String[] args) {
5. // In Java 6 and earlier
6. List<Integer> list = **new** ArrayList<Integer>();
7. list.add(12);
8. **for** (Integer element : list) {
9. System.out.println(element);
10. }
11. // In Java 7
12. List<Integer> list2 = **new** ArrayList<>(); // Here, diamond is used
13. list2.add(12);
14. **for** (Integer element : list2) {
15. System.out.println(element);
16. }
17. }
18. }

Output:

12

12

## Type Inference and Generic Constructors

You can create generic constructor in both generic and non-generic classes. In the following example, we have created a generic type constructor.

1. **class** GenericClass<X> {
2. <T> GenericClass(T t) {
3. System.out.println(t);
4. }
5. }
6. **public** **class** TypeInference {
7. **public** **static** **void** main(String[] args) {
8. GenericClass<String>gc2 = **new** GenericClass<>("Hello");
9. }
10. }

Output:

Hello

# **Java Numeric Literals with Underscore**

Java allows you to use underscore in numeric literals. This feature was introduced in Java 7. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your source code.

The following points are considerable:

* You cannot use underscore at the beginning or end of a number.
  1. Ex. **int** a = \_10; // Error, this is an identifier, not a numeric literal
  2. Ex. **int** a = 10\_; // Error, cannot put underscores at the end of a number
* You cannot use underscore adjacent to a decimal point in a floating point literal.
  1. Ex. **float** a = 10.\_0; // Error, cannot put underscores adjacent to a decimal point
  2. Ex. **float** a = 10\_.0; // Error, cannot put underscores adjacent to a decimal point
* You cannot use underscore prior to an F or L suffix
  1. Ex. **long** a = 10\_100\_00\_L; // Error, cannot put underscores prior to an L suffix
  2. Ex. **float** a = 10\_100\_00\_F; // Error, cannot put underscores prior to an F suffix
* You cannot use underscore in positions where a string of digits is expected.

## Underscores in Numeric Literals Example

1. **public** **class** UnderscoreInNumericLiteralExample {
2. **public** **static** **void** main(String[] args) {
3. // Underscore in integral literal
4. **int** a = 10\_00000;
5. System.out.println("a = "+a);
6. // Underscore in floating literal
7. **float** b = 10.5\_000f;
8. System.out.println("b = "+b);
9. // Underscore in binary literal
10. **int** c = 0B10\_10;
11. System.out.println("c = "+c);
12. // Underscore in hexadecimal literal
13. **int** d = 0x1\_1;
14. System.out.println("d = "+d);
15. // Underscore in octal literal
16. **int** e = 01\_1;
17. System.out.println("e = "+e);
18. }
19. }

Output:

a = 1000000

b = 10.5

c = 10

d = 17

e = 9

# **Java 7 JDBC Improvements**

JDBC (Java Database Connectivity) provides universal data access from the Java programming language. You can access any data from database, spreadsheets or flat files by using JDBC.

In Java 7, Java has introduced the following features:

1) It provides the ability to use a try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement.

2) RowSet 1.1: The introduction of the RowSetFactory interface and the RowSetProvider class, which enable you to create all types of row sets supported by your JDBC driver.

### **RowSetFactory Interface**

It defines the implementation of a factory that is used to obtain different types of RowSet.

### **RowSetFactory Interface Methods**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| CachedRowSet createCachedRowSet() throws SQLException | It creates a new instance of a FilteredRowSet. It throws SQLException, if a CachedRowSet cannot be created. |
| FilteredRowSet createFilteredRowSet() throws SQLException | It creates a new instance of a FilteredRowSet. It throws SQLException, if a FilteredRowSet cannot be created. |
| JdbcRowSet createJdbcRowSet() throws SQLException | It creates a new instance of a JdbcRowSet. It throws SQLException, if a JdbcRowSet cannot be created. |
| JoinRowSet createJoinRowSet() throws SQLException | It creates a new instance of a JoinRowSet. It throws SQLException, if a JoinRowSet cannot be created. |
| WebRowSet createWebRowSet() throws SQLException | It creates a new instance of a WebRowSet. It throws SQLException, if a WebRowSet cannot be created. |

## Java RowSetProvider Class

It is a factory API that helps to applications to get a RowSetFactory implementation that can be used to create different types of RowSet.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static RowSetFactory newFactory() throws SQLException | It creates a new instance of a RowSetFactory implementation. It throws SQLException, if the default factory class cannot be loaded or instantiated. |
| public static RowSetFactory newFactory(String factoryClassName, ClassLoader cl) throws SQLException | It creates a new instance of a RowSetFactory from the specified factory class name. This function is useful when there are multiple providers in the classpath. It gives more control to the application as it can specify which provider should be loaded. It throws SQLException, if factoryClassName is null, or the factory class cannot be loaded. |

## JDBC Example: Mysql Connection by using Try-With-Resources

1. **import** java.sql.Connection;
2. **import** java.sql.DriverManager;
3. **import** java.sql.ResultSet;
4. **import** java.sql.Statement;
5. **class** JdbcExample{
6. **public** **static** **void** main(String args[]){
7. **try**(// --------------try-with-resources begin-------------//
8. // Creating connection
9. Connection con = DriverManager.getConnection( "jdbc:mysql://localhost:3306/student","root","mysql");
10. // Creating statement
11. Statement stmt=con.createStatement();
12. // Executing Sql query
13. ResultSet rs=stmt.executeQuery("select \* from user");
14. )// --------------try-with-resources end--------------//
15. { // ----------------try block begin---------------------//
16. // Iterating ResultSet elements
17. **while**(rs.next()){
18. System.out.println(rs.getInt(1)+""+rs.getString(2)+""+rs.getString(3));
19. }
20. } // ----------------try block end----------------------//
21. **catch**(Exception e){ // Exception handler
22. System.out.println(e.getMessage());
23. }
24. }
25. }

In the above example, we have used try-with-resources. It is used to close resources after completing try block. Now, you don't need to close database connection explicitly.

Make sure you are using JDBC version 4.0 or higher and Java version 1.6 or higher.

## RowSet 1.1

In earlier versions of Java, you have created instances of JdbcRowSet, CachedRowSet, FilteredRowSet etc by using JdbcRowSetImpl class.

Now, Java 7 has added a new RowSet 1.1. So, you can create instance of JdbcRowSet by using RowSetFactory interface.

## Java CachedRowSet

Itstores (caches) data into memory so that is can perform operations on its own data rather than data stored in the database. It can operate without being connected to its data source, that why, it is also known as disconnectedRowSet.

### **Java JDBC Example: CachedRowSet**

1. **import** java.sql.Connection;
2. **import** java.sql.DriverManager;
3. **import** java.sql.ResultSet;
4. **import** java.sql.Statement;
5. **import** javax.sql.rowset.CachedRowSet;
6. **import** javax.sql.rowset.RowSetProvider;
7. **class** JdbcExample{
8. **public** **static** **void** main(String args[]) **throws** Exception{
9. **try**(// --------------try-with-resources begin-------------//
10. // Creating connection
11. Connection con = DriverManager.getConnection( "jdbc:mysql://localhost:3306/student","root","mysql");
12. // Creating statement
13. Statement stmt=con.createStatement();
14. // Executing query
15. ResultSet rs=stmt.executeQuery("select \* from user");
16. )// --------------try-with-resources end--------------//
17. { // ----------------try block begin---------------------//
18. // Creating CachedRowSet
19. CachedRowSet cRS = RowSetProvider.newFactory().createCachedRowSet();
20. // Populating ResultSet data into CachedRowSet
21. cRS.populate(rs);
22. **while**(cRS.next()){
23. System.out.println(cRS.getInt(1)+""+cRS.getString(2)+""+cRS.getString(3));
24. }
25. } // ----------------try block end----------------------//
26. **catch**(Exception e){ // Exception handler
27. System.out.println(e);
28. }
29. }
30. }

## Java JdbcRowSet

It is an improvedResultSet object which is used to maintain connection to a data source. It is similar to ResultSet, but the big difference is that it provides set of properties and listener like a JavaBeans.The main purpose of JdbcRowSet is to make a ResultSet scrollable and updatable.

In the following example, we are creating instance of JdbcRowSet by using new approach.

### **Java JdbcRowSet Example 1**

1. **import** javax.sql.rowset.JdbcRowSet;
2. **import** javax.sql.rowset.RowSetProvider;
3. **class** JdbcExample{
4. **public** **static** **void** main(String args[]) **throws** Exception{
5. **try**(// --------------try-with-resources begin-------------//
6. // Creating connection
7. JdbcRowSet jRS = RowSetProvider.newFactory().createJdbcRowSet();
8. )// --------------try-with-resources end--------------//
9. { // ----------------try block begin---------------------//
10. // Set database connection
11. jRS.setUrl("jdbc:mysql://localhost:3306/student");
12. // Set database username
13. jRS.setUsername("root");
14. // Set database password
15. jRS.setPassword("mysql");
16. // Set sql query to execute
17. jRS.setCommand("select \* from user");
18. // Execute query
19. jRS.execute();
20. **while**(jRS.next()){
21. System.out.println(jRS.getInt(1)+""+jRS.getString(2)+""+jRS.getString(3));
22. }
23. } // ----------------try block end----------------------//
24. **catch**(Exception e){ // Exception handler
25. System.out.println(e);
26. }
27. }
28. }

### **Java JdbcRowSet Example: Updating Row**

1. **import** javax.sql.rowset.JdbcRowSet;
2. **import** javax.sql.rowset.RowSetProvider;
3. **class** JdbcExample{
4. **public** staticvoid main(String args[]) **throws** Exception{
5. **try**(// --------------try-with-resources begin-------------//
6. // Creating connection
7. JdbcRowSet jRS = RowSetProvider.newFactory().createJdbcRowSet();
8. )// --------------try-with-resources end--------------//
9. { // ----------------try block begin---------------------//
10. // Set database connection
11. jRS.setUrl("jdbc:mysql://localhost:3306/student");
12. // Set database username
13. jRS.setUsername("root");
14. // Set database password
15. jRS.setPassword("mysql");
16. // Set sql query to execute
17. jRS.setCommand("select \* from user");
18. // Execute query
19. jRS.execute();
20. // Getting 3rd row because it is scrollable by default
21. jRS.absolute(3);
22. System.out.println(jRS.getInt(1)+""+jRS.getString(2)+""+jRS.getString(3));
23. // Updating 3rd row
24. jRS.updateString("name", "Neraj Kumar Singh");
25. jRS.updateRow();
26. // Fetching 3rd row again
27. System.out.println(jRS.getInt(1)+""+jRS.getString(2)+""+jRS.getString(3));
28. } // ----------------try block end----------------------//
29. **catch**(Exception e){ // Exception handler
30. System.out.println(e);
31. }
32. }
33. }

Output:

3 Neraj kumar 8562697858

3 Neraj Kumar Singh 8562697858

# **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. }

[download this example](https://www.javatpoint.com/src/newjdk/assertion1.zip)

|  |
| --- |
| 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.

# **For-each loop (Advanced or Enhanced For loop):**

The for-each loop introduced in Java5. It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

## Advantage of for-each loop:

* It makes the code more readable.
* It elimnates the possibility of programming errors.

## Syntax of for-each loop:

1. **for**(data\_type variable : array | collection){}

### **Simple Example of for-each loop for traversing the array elements:**

2. **class** ForEachExample1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={12,13,14,44};
6. **for**(**int** i:arr){
7. System.out.println(i);
8. }
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=ForEachExample1)

Output:12

13

14

44

### **Simple Example of for-each loop for traversing the collection elements:**

1. **import** java.util.\*;
2. **class** ForEachExample2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();
5. list.add("vimal");
6. list.add("sonoo");
7. list.add("ratan");
9. **for**(String s:list){
10. System.out.println(s);
11. }
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=ForEachExample2)

Output:vimal

sonoo

ratan

# **Variable Argument (Varargs):**

The varrags allows the method to accept zero or muliple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

## Advantage of Varargs:

We don't have to provide overloaded methods so less code.

## Syntax of varargs:

The varargs uses ellipsis i.e. three dots after the data type. Syntax is as follows:

1. return\_type method\_name(data\_type... variableName){}

### **Simple Example of Varargs in java:**

2. **class** VarargsExample1{
4. **static** **void** display(String... values){
5. System.out.println("display method invoked ");
6. }
8. **public** **static** **void** main(String args[]){
10. display();//zero argument
11. display("my","name","is","varargs");//four arguments
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=VarargsExample1)

Output:display method invoked

display method invoked

### **Another Program of Varargs in java:**

2. **class** VarargsExample2{
4. **static** **void** display(String... values){
5. System.out.println("display method invoked ");
6. **for**(String s:values){
7. System.out.println(s);
8. }
9. }
11. **public** **static** **void** main(String args[]){
13. display();//zero argument
14. display("hello");//one argument
15. display("my","name","is","varargs");//four arguments
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=VarargsExample2)

Output:display method invoked

display method invoked

hello

display method invoked

my

name

is

varargs

[download this example](https://www.javatpoint.com/src/newjdk/varargs2.zip)

## Rules for varargs:

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

* There can be only one variable argument in the method.
* Variable argument (varargs) must be the last argument.

## Examples of varargs that fails to compile:

2. **void** method(String... a, **int**... b){}//Compile time error
4. **void** method(**int**... a, String b){}//Compile time error

### **Example of Varargs that is the last argument in the method:**

2. **class** VarargsExample3{
4. **static** **void** display(**int** num, String... values){
5. System.out.println("number is "+num);
6. **for**(String s:values){
7. System.out.println(s);
8. }
9. }
11. **public** **static** **void** main(String args[]){
13. display(500,"hello");//one argument
14. display(1000,"my","name","is","varargs");//four arguments
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=VarargsExample3)

Output:number is 500

hello

number is 1000

my

name

is

varargs

[**next →**](https://www.javatpoint.com/autoboxing-and-unboxing)[**← prev**](https://www.javatpoint.com/varargs)

# **Java Static Import**

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

## Advantage of static import:

* Less coding is required if you have access any static member of a class oftenly.

## Disadvantage of static import:

* If you overuse the static import feature, it makes the program unreadable and unmaintainable.

### **Simple Example of static import**

1. **import** **static** java.lang.System.\*;
2. **class** StaticImportExample{
3. **public** **static** **void** main(String args[]){
5. out.println("Hello");//Now no need of System.out
6. out.println("Java");
8. }
9. }

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

Output:Hello

Java

# **Autoboxing and Unboxing:**

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

## Advantage of Autoboxing and Unboxing:

|  |
| --- |
| No need of conversion between primitives and Wrappers manually so less coding is required. |

### **Simple Example of Autoboxing in java:**

2. **class** BoxingExample1{
3. **public** **static** **void** main(String args[]){
4. **int** a=50;
5. Integer a2=**new** Integer(a);//Boxing
7. Integer a3=5;//Boxing
9. System.out.println(a2+" "+a3);
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=BoxingExample1)

Output:50 5

[download this example](https://www.javatpoint.com/src/newjdk/boxing1.zip)

### **Simple Example of Unboxing in java:**

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

2. **class** UnboxingExample1{
3. **public** **static** **void** main(String args[]){
4. Integer i=**new** Integer(50);
5. **int** a=i;
7. System.out.println(a);
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=UnboxingExample1)

Output:50

### **Autoboxing and Unboxing with comparison operators**

|  |
| --- |
| Autoboxing can be performed with comparison operators. Let's see the example of boxing with comparison operator: |

2. **class** UnboxingExample2{
3. **public** **static** **void** main(String args[]){
4. Integer i=**new** Integer(50);
6. **if**(i<100){            //unboxing internally
7. System.out.println(i);
8. }
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=UnboxingExample2)

Output:50

### **Autoboxing and Unboxing with method overloading**

|  |
| --- |
| In method overloading, boxing and unboxing can be performed. There are some rules for method overloading with boxing:   * **Widening beats boxing** * **Widening beats varargs** * **Boxing beats varargs** |

### **1) Example of Autoboxing where widening beats boxing**

|  |
| --- |
| If there is possibility of widening and boxing, widening beats boxing. |

2. **class** Boxing1{
3. **static** **void** m(**int** i){System.out.println("int");}
4. **static** **void** m(Integer i){System.out.println("Integer");}
6. **public** **static** **void** main(String args[]){
7. **short** s=30;
8. m(s);
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Boxing1)

Output:int

### **2) Example of Autoboxing where widening beats varargs**

|  |
| --- |
| If there is possibility of widening and varargs, widening beats var-args. |

2. **class** Boxing2{
3. **static** **void** m(**int** i, **int** i2){System.out.println("int int");}
4. **static** **void** m(Integer... i){System.out.println("Integer...");}
6. **public** **static** **void** main(String args[]){
7. **short** s1=30,s2=40;
8. m(s1,s2);
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Boxing2)

Output:int int

### **3) Example of Autoboxing where boxing beats varargs**

|  |
| --- |
| Let's see the program where boxing beats variable argument: |

2. **class** Boxing3{
3. **static** **void** m(Integer i){System.out.println("Integer");}
4. **static** **void** m(Integer... i){System.out.println("Integer...");}
6. **public** **static** **void** main(String args[]){
7. **int** a=30;
8. m(a);
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Boxing3)

Output:Integer

### **Method overloading with Widening and Boxing**

|  |
| --- |
| Widening and Boxing can't be performed as given below: |

2. **class** Boxing4{
3. **static** **void** m(Long l){System.out.println("Long");}
5. **public** **static** **void** main(String args[]){
6. **int** a=30;
7. m(a);
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Boxing4)

Output:Compile Time Error

### **What is the difference between import and static import?**

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

# **Java Enum**

**Enum in java** is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly. It is available from JDK 1.5.

Java Enums can be thought of as classes that have fixed set of constants.

## Points to remember for Java Enum

* enum improves type safety
* enum can be easily used in switch
* enum can be traversed
* enum can have fields, constructors and methods
* enum may implement many interfaces but cannot extend any class because it internally extends Enum class

### **Simple example of java enum**

1. **class** EnumExample1{
2. **public** **enum** Season { WINTER, SPRING, SUMMER, FALL }
4. **public** **static** **void** main(String[] args) {
5. **for** (Season s : Season.values())
6. System.out.println(s);
8. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=EnumExample1)

Output:WINTER

SPRING

SUMMER

FALL

[download this enum example](https://www.javatpoint.com/src/newjdk/enum.zip)

### **What is the purpose of values() method in enum?**

The java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

### **Internal code generated by the compiler for the above example of enum type**

The java compiler internally creates a static and final class that extends the Enum class as shown in the below example:

1. **public** **static** **final** **class** EnumExample1$Season **extends** Enum
2. {
3. **private** EnumExample1$Season(String s, **int** i)
4. {
5. **super**(s, i);
6. }
8. **public** **static** EnumExample1$Season[] values()
9. {
10. **return** (EnumExample1$Season[])$VALUES.clone();
11. }
13. **public** **static** EnumExample1$Season valueOf(String s)
14. {
15. **return** (EnumExample1$Season)Enum.valueOf(EnumExample1$Season, s);
16. }
18. **public** **static** **final** EnumExample1$Season WINTER;
19. **public** **static** **final** EnumExample1$Season SPRING;
20. **public** **static** **final** EnumExample1$Season SUMMER;
21. **public** **static** **final** EnumExample1$Season FALL;
22. **private** **static** **final** EnumExample1$Season $VALUES[];
24. **static**
25. {
26. WINTER = **new** EnumExample1$Season("WINTER", 0);
27. SPRING = **new** EnumExample1$Season("SPRING", 1);
28. SUMMER = **new** EnumExample1$Season("SUMMER", 2);
29. FALL = **new** EnumExample1$Season("FALL", 3);
30. $VALUES = (**new** EnumExample1$Season[] {
31. WINTER, SPRING, SUMMER, FALL
32. });
33. }
35. }

## Defining Java enum

The enum can be defined within or outside the class because it is similar to a class.

### **Java enum example: defined outside class**

1. **enum** Season { WINTER, SPRING, SUMMER, FALL }
2. **class** EnumExample2{
3. **public** **static** **void** main(String[] args) {
4. Season s=Season.WINTER;
5. System.out.println(s);
6. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=EnumExample2)

Output:WINTER

### **Java enum example: defined inside class**

1. **class** EnumExample3{
2. **enum** Season { WINTER, SPRING, SUMMER, FALL; }//semicolon(;) is optional here
3. **public** **static** **void** main(String[] args) {
4. Season s=Season.WINTER;//enum type is required to access WINTER
5. System.out.println(s);
6. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=EnumExample3)

Output:WINTER

### **Initializing specific values to the enum constants**

The enum constants have initial value that starts from 0, 1, 2, 3 and so on. But we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors and methods.

### **Example of specifying initial value to the enum constants**

1. **class** EnumExample4{
2. **enum** Season{
3. WINTER(5), SPRING(10), SUMMER(15), FALL(20);
5. **private** **int** value;
6. **private** Season(**int** value){
7. **this**.value=value;
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. **for** (Season s : Season.values())
12. System.out.println(s+" "+s.value);
14. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=EnumExample4)

[download this enum example](https://www.javatpoint.com/src/newjdk/enum4.zip)

Output:WINTER 5

SPRING 10

SUMMER 15

FALL 20

#### Constructor of enum type is private. If you don't declare private compiler internally creates private constructor.

1. **enum** Season{
2. WINTER(10),SUMMER(20);
3. **private** **int** value;
4. Season(**int** value){
5. **this**.value=value;
6. }
7. }

### **Internal code generated by the compiler for the above example of enum type**

1. **final** **class** Season **extends** Enum
2. {
3. **public** **static** Season[] values()
4. {
5. **return** (Season[])$VALUES.clone();
6. }
7. **public** **static** Season valueOf(String s)
8. {
9. **return** (Season)Enum.valueOf(Season, s);
10. }
11. **private** Season(String s, **int** i, **int** j)
12. {
13. **super**(s, i);
14. value = j;
15. }
16. **public** **static** **final** Season WINTER;
17. **public** **static** **final** Season SUMMER;
18. **private** **int** value;
19. **private** **static** **final** Season $VALUES[];
20. **static**
21. {
22. WINTER = **new** Season("WINTER", 0, 10);
23. SUMMER = **new** Season("SUMMER", 1, 20);
24. $VALUES = (**new** Season[] {
25. WINTER, SUMMER
26. });
27. }
28. }

### **Can we create the instance of enum by new keyword?**

|  |
| --- |
| No, because it contains private constructors only. |

### **Can we have abstract method in enum?**

Yes, ofcourse! we can have abstract methods and can provide the implementation of these methods.

## Java enum in switch statement

We can apply enum on switch statement as in the given example:

### **Example of applying enum on switch statement**

1. **class** EnumExample5{
2. **enum** Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
3. **public** **static** **void** main(String args[]){
4. Day day=Day.MONDAY;
6. **switch**(day){
7. **case** SUNDAY:
8. System.out.println("sunday");
9. **break**;
10. **case** MONDAY:
11. System.out.println("monday");
12. **break**;
13. **default**:
14. System.out.println("other day");
15. }
16. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=EnumExample5)

[download this enum example](https://www.javatpoint.com/src/newjdk/enum5.zip)

Output:monday

# **Java Annotations**

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces.

First, we will learn some built-in annotations then we will move on creating and using custom annotations.

## Built-In Java Annotations

There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.

## Built-In Java Annotations used in java code

* @Override
* @SuppressWarnings
* @Deprecated

## Built-In Java Annotations used in other annotations

* @Target
* @Retention
* @Inherited
* @Documented

## Understanding Built-In Annotations in java

Let's understand the built-in annotations first.

## @Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

1. **class** Animal{
2. **void** eatSomething(){System.out.println("eating something");}
3. }
5. **class** Dog **extends** Animal{
6. @Override
7. **void** eatsomething(){System.out.println("eating foods");}//should be eatSomething
8. }
10. **class** TestAnnotation1{
11. **public** **static** **void** main(String args[]){
12. Animal a=**new** Dog();
13. a.eatSomething();
14. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnotation1)

Output:Comple Time Error

## @SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

1. **import** java.util.\*;
2. **class** TestAnnotation2{
3. @SuppressWarnings("unchecked")
4. **public** **static** **void** main(String args[]){
5. ArrayList list=**new** ArrayList();
6. list.add("sonoo");
7. list.add("vimal");
8. list.add("ratan");
10. **for**(Object obj:list)
11. System.out.println(obj);
13. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnotation2)

Now no warning at compile time.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

## @Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

1. **class** A{
2. **void** m(){System.out.println("hello m");}
4. @Deprecated
5. **void** n(){System.out.println("hello n");}
6. }
8. **class** TestAnnotation3{
9. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnotation3)

## At Compile Time:

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

## At Runtime:

hello n

## Custom Annotation

To create and use custom java annotation, visit the next page.

# **Java Custom Annotation**

**Java Custom annotations** or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

1. **@interface** MyAnnotation{}

Here, MyAnnotation is the custom annotation name.

## Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

## Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

## 1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

1. **@interface** MyAnnotation{}

The @Override and @Deprecated are marker annotations.

## 2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

1. **@interface** MyAnnotation{
2. **int** value();
3. }

We can provide the default value also. For example:

1. **@interface** MyAnnotation{
2. **int** value() **default** 0;
3. }

## How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

1. @MyAnnotation(value=10)

The value can be anything.

## 3) Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

1. **@interface** MyAnnotation{
2. **int** value1();
3. String value2();
4. String value3();
5. }
6. }

We can provide the default value also. For example:

1. **@interface** MyAnnotation{
2. **int** value1() **default** 1;
3. String value2() **default** "";
4. String value3() **default** "xyz";
5. }

## How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

1. @MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")

## Built-in Annotations used in custom annotations in java

* @Target
* @Retention
* @Inherited
* @Documented

## @Target

**@Target** tag is used to specify at which type, the annotation is used.

The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

|  |  |
| --- | --- |
| **Element Types** | **Where the annotation can be applied** |
| TYPE | class, interface or enumeration |
| FIELD | fields |
| METHOD | methods |
| CONSTRUCTOR | constructors |
| LOCAL\_VARIABLE | local variables |
| ANNOTATION\_TYPE | annotation type |
| PARAMETER | parameter |

## Example to specify annoation for a class

1. @Target(ElementType.TYPE)
2. **@interface** MyAnnotation{
3. **int** value1();
4. String value2();
5. }

## Example to specify annotation for a class, methods or fields

1. @Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
2. **@interface** MyAnnotation{
3. **int** value1();
4. String value2();
5. }

## @Retention

**@Retention** annotation is used to specify to what level annotation will be available.

|  |  |
| --- | --- |
| **RetentionPolicy** | **Availability** |
| RetentionPolicy.SOURCE | refers to the source code, discarded during compilation. It will not be available in the compiled class. |
| RetentionPolicy.CLASS | refers to the .class file, available to java compiler but not to JVM . It is included in the class file. |
| RetentionPolicy.RUNTIME | refers to the runtime, available to java compiler and JVM . |

## Example to specify the RetentionPolicy

1. @Retention(RetentionPolicy.RUNTIME)
2. @Target(ElementType.TYPE)
3. **@interface** MyAnnotation{
4. **int** value1();
5. String value2();
6. }

## Example of custom annotation: creating, applying and accessing annotation

Let's see the simple example of creating, applying and accessing annotation.

*File: Test.java*

1. //Creating annotation
2. **import** java.lang.annotation.\*;
3. **import** java.lang.reflect.\*;
5. @Retention(RetentionPolicy.RUNTIME)
6. @Target(ElementType.METHOD)
7. **@interface** MyAnnotation{
8. **int** value();
9. }
11. //Applying annotation
12. **class** Hello{
13. @MyAnnotation(value=10)
14. **public** **void** sayHello(){System.out.println("hello annotation");}
15. }
17. //Accessing annotation
18. **class** TestCustomAnnotation1{
19. **public** **static** **void** main(String args[])**throws** Exception{
21. Hello h=**new** Hello();
22. Method m=h.getClass().getMethod("sayHello");
24. MyAnnotation manno=m.getAnnotation(MyAnnotation.**class**);
25. System.out.println("value is: "+manno.value());
26. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCustomAnnotation1)

Output:value is: 10

[download this example](https://www.javatpoint.com/src/newjdk/annotation.zip)

## How built-in annotaions are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

## @Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

1. @Inherited
2. **@interface** ForEveryone { }//Now it will be available to subclass also
4. **@interface** ForEveryone { }
5. **class** Superclass{}
7. **class** Subclass **extends** Superclass{}

## @Documented

The @Documented Marks the annotation for inclusion in the documentation.

# **Generics in Java**

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

#### **Advantage of Java Generics**

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety :** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

1. List list = **new** ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting

After Generics, we don't need to typecast the object.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. String s = list.get(0);

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. list.add(32);//Compile Time Error

**Syntax** to use generic collection

1. ClassOrInterface<Type>

**Example** to use Generics in java

1. ArrayList<String>

## Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

1. **import** java.util.\*;
2. **class** TestGenerics1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();
5. list.add("rahul");
6. list.add("jai");
7. //list.add(32);//compile time error
9. String s=list.get(1);//type casting is not required
10. System.out.println("element is: "+s);
12. Iterator<String> itr=list.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGenerics1)

Output:element is: jai

rahul

jai

## Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

1. **import** java.util.\*;
2. **class** TestGenerics2{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(1,"vijay");
6. map.put(4,"umesh");
7. map.put(2,"ankit");
9. //Now use Map.Entry for Set and Iterator
10. Set<Map.Entry<Integer,String>> set=map.entrySet();
12. Iterator<Map.Entry<Integer,String>> itr=set.iterator();
13. **while**(itr.hasNext()){
14. Map.Entry e=itr.next();//no need to typecast
15. System.out.println(e.getKey()+" "+e.getValue());
16. }
18. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGenerics2)

Output:1 vijay

2 ankit

4 umesh

## Generic class

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let’s see the simple example to create and use the generic class.

**Creating generic class:**

1. **class** MyGen<T>{
2. T obj;
3. **void** add(T obj){**this**.obj=obj;}
4. T get(){**return** obj;}
5. }

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

**Using generic class:**

Let’s see the code to use the generic class.

1. **class** TestGenerics3{
2. **public** **static** **void** main(String args[]){
3. MyGen<Integer> m=**new** MyGen<Integer>();
4. m.add(2);
5. //m.add("vivek");//Compile time error
6. System.out.println(m.get());
7. }}

Output:2

## Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

## Generic Method

Like generic class, we can create generic method that can accept any type of argument.

Let’s see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

1. **public** **class** TestGenerics4{
3. **public** **static** < E > **void** printArray(E[] elements) {
4. **for** ( E element : elements){
5. System.out.println(element );
6. }
7. System.out.println();
8. }
9. **public** **static** **void** main( String args[] ) {
10. Integer[] intArray = { 10, 20, 30, 40, 50 };
11. Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };
13. System.out.println( "Printing Integer Array" );
14. printArray( intArray  );
16. System.out.println( "Printing Character Array" );
17. printArray( charArray );
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGenerics4)

Output:Printing Integer Array

10

20

30

40

50

Printing Character Array

J

A

V

A

T

P

O

I

N

T

## Wildcard in Java Generics

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

Let's understand it by the example given below:

1. **import** java.util.\*;
2. **abstract** **class** Shape{
3. **abstract** **void** draw();
4. }
5. **class** Rectangle **extends** Shape{
6. **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle **extends** Shape{
9. **void** draw(){System.out.println("drawing circle");}
10. }

13. **class** GenericTest{
14. //creating a method that accepts only child class of Shape
15. **public** **static** **void** drawShapes(List<? **extends** Shape> lists){
16. **for**(Shape s:lists){
17. s.draw();//calling method of Shape class by child class instance
18. }
19. }
20. **public** **static** **void** main(String args[]){
21. List<Rectangle> list1=**new** ArrayList<Rectangle>();
22. list1.add(**new** Rectangle());
24. List<Circle> list2=**new** ArrayList<Circle>();
25. list2.add(**new** Circle());
26. list2.add(**new** Circle());
28. drawShapes(list1);
29. drawShapes(list2);
30. }}

drawing rectangle

drawing circle

drawing circle

# **RMI (Remote Method Invocation)**

1. [Remote Method Invocation (RMI)](https://www.javatpoint.com/RMI)
2. [Understanding stub and skeleton](https://www.javatpoint.com/RMI#rmistubandskeleton)
   1. [stub](https://www.javatpoint.com/RMI#stub)
   2. [skeleton](https://www.javatpoint.com/RMI#skeleton)
3. [Requirements for the distributed applications](https://www.javatpoint.com/RMI#reqdistributed)
4. [Steps to write the RMI program](https://www.javatpoint.com/RMI#rmisteps)
5. [RMI Example](https://www.javatpoint.com/RMI#rmiex)

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

### **Understanding stub and skeleton**

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

### **stub**

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

### **skeleton**

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.

## Understanding requirements for the distributed applications

If any application performs these tasks, it can be distributed application.

.

1. The application need to locate the remote method
2. It need to provide the communication with the remote objects, and
3. The application need to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.

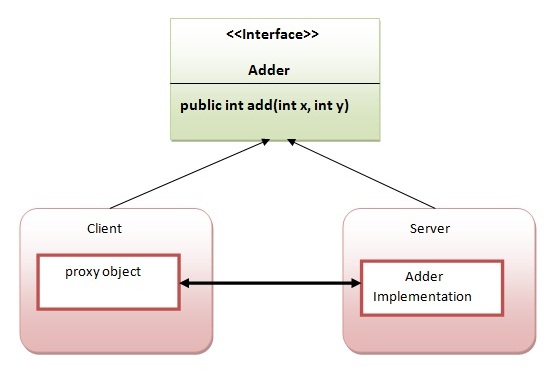
## Java RMI Example

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**

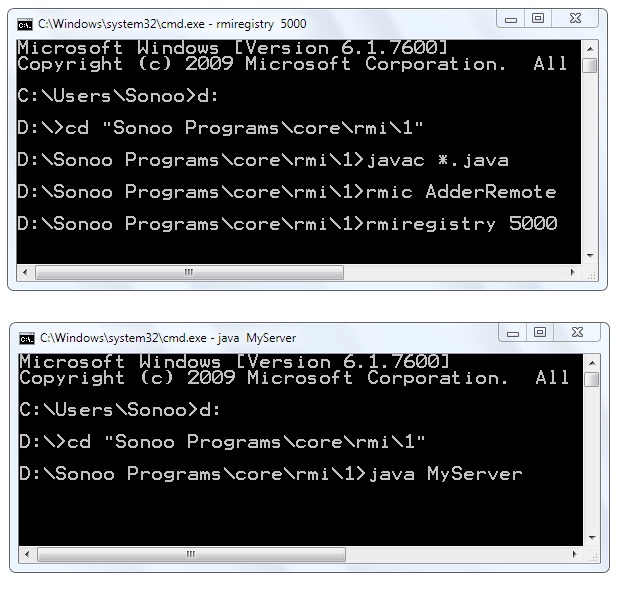
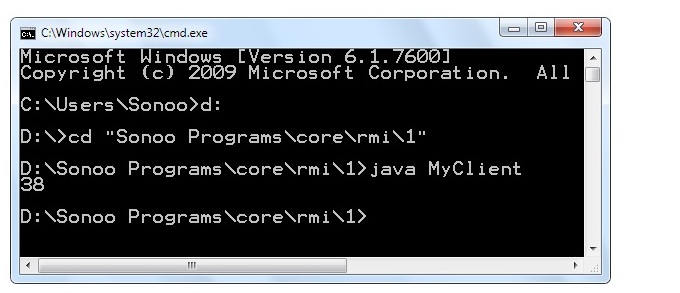
At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine,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

[download this example of rmi](https://www.javatpoint.com/src/rmi/rmi1.zip)

1. For running **this** rmi example,
3. 1) compile all the java files
5. javac \*.java
7. 2)create stub and skeleton object by rmic tool
9. rmic AdderRemote
11. 3)start rmi registry in one command prompt
13. rmiregistry 5000
15. 4)start the server in another command prompt
17. java MyServer
19. 5)start the client application in another command prompt
21. java MyClient

### **Output of this RMI example**

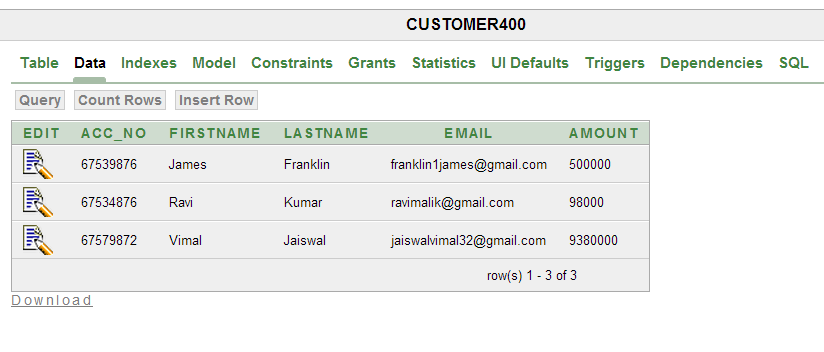
### **Meaningful example of RMI application with database**

Consider a scenario, there are two applications running in different machines. Let's say MachineA and MachineB, machineA is located in United States and MachineB in India. MachineB want to get list of all the customers of MachineA application.

Let's develop the RMI application by following the steps.

#### **1) Create the table**

First of all, we need to create the table in the database. Here, we are using Oracle10 database.



#### **2) Create Customer class and Remote interface**

*File: Customer.java*

1. **package** com.javatpoint;
2. **public** **class** Customer **implements** java.io.Serializable{
3. **private** **int** acc\_no;
4. **private** String firstname,lastname,email;
5. **private** **float** amount;
6. //getters and setters
7. }

#### Note: Customer class must be Serializable.

*File: Bank.java*

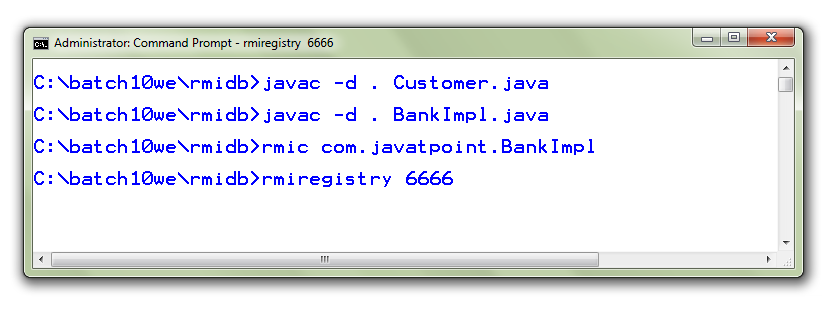
1. **package** com.javatpoint;
2. **import** java.rmi.\*;
3. **import** java.util.\*;
4. **interface** Bank **extends** Remote{
5. **public** List<Customer> getCustomers()**throws** RemoteException;
6. }

#### **3) Create the class that provides the implementation of Remote interface**

*File: BankImpl.java*

1. **package** com.javatpoint;
2. **import** java.rmi.\*;
3. **import** java.rmi.server.\*;
4. **import** java.sql.\*;
5. **import** java.util.\*;
6. **class** BankImpl **extends** UnicastRemoteObject **implements** Bank{
7. BankImpl()**throws** RemoteException{}
9. **public** List<Customer> getCustomers(){
10. List<Customer> list=**new** ArrayList<Customer>();
11. **try**{
12. Class.forName("oracle.jdbc.driver.OracleDriver");
13. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
14. PreparedStatement ps=con.prepareStatement("select \* from customer400");
15. ResultSet rs=ps.executeQuery();
17. **while**(rs.next()){
18. Customer c=**new** Customer();
19. c.setAcc\_no(rs.getInt(1));
20. c.setFirstname(rs.getString(2));
21. c.setLastname(rs.getString(3));
22. c.setEmail(rs.getString(4));
23. c.setAmount(rs.getFloat(5));
24. list.add(c);
25. }
27. con.close();
28. }**catch**(Exception e){System.out.println(e);}
29. **return** list;
30. }//end of getCustomers()
31. }

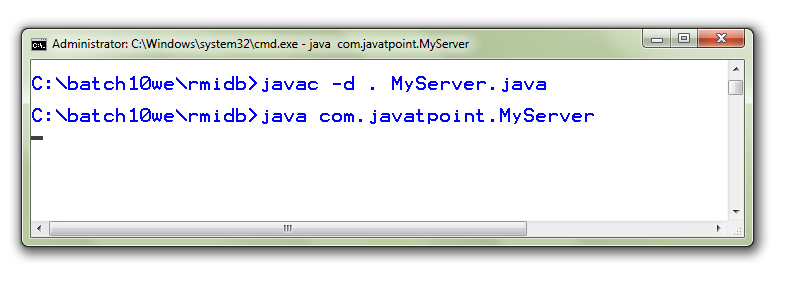
#### **4) Compile the class rmic tool and start the registry service by rmiregistry tool**



#### **5) Create and run the Server**

*File: MyServer.java*

1. **package** com.javatpoint;
2. **import** java.rmi.\*;
3. **public** **class** MyServer{
4. **public** **static** **void** main(String args[])**throws** Exception{
5. Remote r=**new** BankImpl();
6. Naming.rebind("rmi://localhost:6666/javatpoint",r);
7. }}



#### **6) Create and run the Client**

*File: MyClient.java*

1. **package** com.javatpoint;
2. **import** java.util.\*;
3. **import** java.rmi.\*;
4. **public** **class** MyClient{
5. **public** **static** **void** main(String args[])**throws** Exception{
6. Bank b=(Bank)Naming.lookup("rmi://localhost:6666/javatpoint");
8. List<Customer> list=b.getCustomers();
9. **for**(Customer c:list){
10. System.out.println(c.getAcc\_no()+" "+c.getFirstname()+" "+c.getLastname()
11. +" "+c.getEmail()+" "+c.getAmount());
12. }
14. }}