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 is 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

@FunctionalInterface

**interface** sayable{

**void** say(String msg);

}

**public** **class** FunctionalInterfaceExample **implements** sayable{

**public** **void** say(String msg){

        System.out.println(msg);

    }

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

        FunctionalInterfaceExample fie = **new** FunctionalInterfaceExample();

        fie.say("Hello there");

    }

}

Example 2

@FunctionalInterface

**interface** sayable{

**void** say(String msg);   // abstract method

    // It can contain any number of Object class methods.

**int** hashCode();

    String toString();

**boolean** equals(Object obj);

}

**public** **class** FunctionalInterfaceExample2 **implements** sayable{

**public** **void** say(String msg){

        System.out.println(msg);

    }

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

        FunctionalInterfaceExample2 fie = **new** FunctionalInterfaceExample2();

        fie.say("Hello there");

    }

}

Invalid Functional Interface

A functional interface can extends another interface only when it does not have any abstract method.

**interface** sayable{

**void** say(String msg);   // abstract method

}

@FunctionalInterface

**interface** Doable **extends** sayable{

    // Invalid '@FunctionalInterface' annotation; Doable is not a functional interface

**void** doIt();

}

Example 3

In the following example, a functional interface is extending to a non-functional interface.

**interface** Doable{

**default** **void** doIt(){

        System.out.println("Do it now");

    }

}

@FunctionalInterface

**interface** Sayable **extends** Doable{

**void** say(String msg);   // abstract method

}

**public** **class** FunctionalInterfaceExample3 **implements** Sayable{

**public** **void** say(String msg){

        System.out.println(msg);

    }

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

        FunctionalInterfaceExample3 fie = **new** FunctionalInterfaceExample3();

        fie.say("Hello there");

        fie.doIt();

    }

}

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

**interface** Sayable{

    // Default method

**default** **void** say(){

        System.out.println("Hello, this is default method");

    }

    // Abstract method

**void** sayMore(String msg);

}

**public** **class** DefaultMethods **implements** Sayable{

**public** **void** sayMore(String msg){        // implementing abstract method

        System.out.println(msg);

    }

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

        DefaultMethods dm = **new** DefaultMethods();

        dm.say();   // calling default method

        dm.sayMore("Work is worship");  // calling abstract method

    }

}

## **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?

**interface** Sayable{

    // default method

**default** **void** say(){

        System.out.println("Hello, this is default method");

    }

    // Abstract method

**void** sayMore(String msg);

    // static method

**static** **void** sayLouder(String msg){

        System.out.println(msg);

    }

}

**public** **class** DefaultMethods **implements** Sayable{

**public** **void** sayMore(String msg){     // implementing abstract method

        System.out.println(msg);

    }

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

        DefaultMethods dm = **new** DefaultMethods();

        dm.say();                       // calling default method

        dm.sayMore("Work is worship");      // calling abstract method

        Sayable.sayLouder("Helloooo...");   // calling static method

    }

}

# 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.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

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 annotation @FunctionalInterface, which is used to declare an interface as functional interface.

Prime Ministers of India | List of Prime Minister of India (1947-2020)

## **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.

**No Parameter Syntax**

() -> {

//Body of no parameter lambda

}

**One Parameter Syntax**

(p1) -> {

//Body of single parameter lambda

}

**Two Parameter Syntax**

(p1,p2) -> {

//Body of multiple parameter lambda

}

Let's see a scenario where we are not implementing Java lambda expression. Here, we are implementing an interface without using lambda expression.

## **Without Lambda Expression**

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample {

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

**int** width=10;

        //without lambda, Drawable implementation using anonymous class

        Drawable d=**new** Drawable(){

**public** **void** draw(){System.out.println("Drawing "+width);}

        };

        d.draw();

    }

}

Output:

Drawing 10

## **Java Lambda Expression Example**

Now, we are going to implement the above example with the help of Java lambda expression.

@FunctionalInterface  //It is optional

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample2 {

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

**int** width=10;

        //with lambda

        Drawable d2=()->{

            System.out.println("Drawing "+width);

        };

        d2.draw();

    }

}

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**

**interface** Sayable{

**public** String say();

}

**public** **class** LambdaExpressionExample3{

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

    Sayable s=()->{

**return** "I have nothing to say.";

    };

    System.out.println(s.say());

}

}

Output:

I have nothing to say.

## **Java Lambda Expression Example: Single Parameter**

**interface** Sayable{

**public** String say(String name);

}

**public** **class** LambdaExpressionExample4{

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

        // Lambda expression with single parameter.

        Sayable s1=(name)->{

**return** "Hello, "+name;

        };

        System.out.println(s1.say("Sonoo"));

        // You can omit function parentheses

        Sayable s2= name ->{

**return** "Hello, "+name;

        };

        System.out.println(s2.say("Sonoo"));

    }

}

Output:

Hello, Sonoo

Hello, Sonoo

## **Java Lambda Expression Example: Multiple Parameters**

**interface** Addable{

**int** add(**int** a,**int** b);

}

**public** **class** LambdaExpressionExample5{

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

        // Multiple parameters in lambda expression

        Addable ad1=(a,b)->(a+b);

        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression

        Addable ad2=(**int** a,**int** b)->(a+b);

        System.out.println(ad2.add(100,200));

    }

}

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.

**interface** Addable{

**int** add(**int** a,**int** b);

}

**public** **class** LambdaExpressionExample6 {

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

        // Lambda expression without return keyword.

        Addable ad1=(a,b)->(a+b);

        System.out.println(ad1.add(10,20));

        // Lambda expression with return keyword.

        Addable ad2=(**int** a,**int** b)->{

**return** (a+b);

                            };

        System.out.println(ad2.add(100,200));

    }

}

Output:

30

300

## **Java Lambda Expression Example: Foreach Loop**

**import** java.util.\*;

**public** **class** LambdaExpressionExample7{

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

        List<String> list=**new** ArrayList<String>();

        list.add("ankit");

        list.add("mayank");

        list.add("irfan");

        list.add("jai");

        list.forEach(

            (n)->System.out.println(n)

        );

    }

}

Output:

ankit

mayank

irfan

jai

## **Java Lambda Expression Example: Multiple Statements**

@FunctionalInterface

**interface** Sayable{

    String say(String message);

}

**public** **class** LambdaExpressionExample8{

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

        // You can pass multiple statements in lambda expression

        Sayable person = (message)-> {

            String str1 = "I would like to say, ";

            String str2 = str1 + message;

**return** str2;

        };

            System.out.println(person.say("time is precious."));

    }

}

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.

**public** **class** LambdaExpressionExample9{

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

        //Thread Example without lambda

        Runnable r1=**new** Runnable(){

**public** **void** run(){

                System.out.println("Thread1 is running...");

            }

        };

        Thread t1=**new** Thread(r1);

        t1.start();

        //Thread Example with lambda

        Runnable r2=()->{

                System.out.println("Thread2 is running...");

        };

        Thread t2=**new** Thread(r2);

        t2.start();

    }

}

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**

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** LambdaExpressionExample10{

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

        List<Product> list=**new** ArrayList<Product>();

        //Adding Products

        list.add(**new** Product(1,"HP Laptop",25000f));

        list.add(**new** Product(3,"Keyboard",300f));

        list.add(**new** Product(2,"Dell Mouse",150f));

        System.out.println("Sorting on the basis of name...");

        // implementing lambda expression

        Collections.sort(list,(p1,p2)->{

**return** p1.name.compareTo(p2.name);

        });

**for**(Product p:list){

            System.out.println(p.id+" "+p.name+" "+p.price);

        }

    }

}

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**

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** LambdaExpressionExample11{

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

        List<Product> list=**new** ArrayList<Product>();

        list.add(**new** Product(1,"Samsung A5",17000f));

        list.add(**new** Product(3,"Iphone 6S",65000f));

        list.add(**new** Product(2,"Sony Xperia",25000f));

        list.add(**new** Product(4,"Nokia Lumia",15000f));

        list.add(**new** Product(5,"Redmi4 ",26000f));

        list.add(**new** Product(6,"Lenevo Vibe",19000f));

        // using lambda to filter data

        Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);

        // using lambda to iterate through collection

        filtered\_data.forEach(

                product -> System.out.println(product.name+": "+product.price)

        );

    }

}

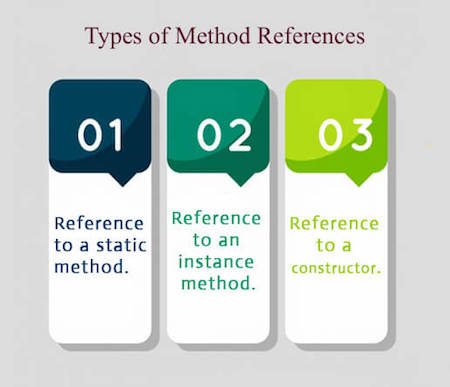
# Java Method References

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

## **Types of Method References**

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. 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().

**interface** Sayable{

**void** say();

}

**public** **class** MethodReference {

**public** **static** **void** saySomething(){

        System.out.println("Hello, this is static method.");

    }

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

        // Referring static method

        Sayable sayable = MethodReference::saySomething;

        // Calling interface method

        sayable.say();

    }

}

Output:

Hello, this is static method.

### Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

**public** **class** MethodReference2 {

**public** **static** **void** ThreadStatus(){

        System.out.println("Thread is running...");

    }

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

        Thread t2=**new** Thread(MethodReference2::ThreadStatus);

        t2.start();

    }

}

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.

**import** java.util.function.BiFunction;

**class** Arithmetic{

**public** **static** **int** add(**int** a, **int** b){

**return** a+b;

}

}

**public** **class** MethodReference3 {

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

BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;

**int** result = adder.apply(10, 20);

System.out.println(result);

}

}

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.

**import** java.util.function.BiFunction;

**class** Arithmetic{

**public** **static** **int** add(**int** a, **int** b){

**return** a+b;

}

**public** **static** **float** add(**int** a, **float** b){

**return** a+b;

}

**public** **static** **float** add(**float** a, **float** b){

**return** a+b;

}

}

**public** **class** MethodReference4 {

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

BiFunction<Integer, Integer, Integer>adder1 = Arithmetic::add;

BiFunction<Integer, Float, Float>adder2 = Arithmetic::add;

BiFunction<Float, Float, Float>adder3 = Arithmetic::add;

**int** result1 = adder1.apply(10, 20);

**float** result2 = adder2.apply(10, 20.0f);

**float** result3 = adder3.apply(10.0f, 20.0f);

System.out.println(result1);

System.out.println(result2);

System.out.println(result3);

}

}

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.

**interface** Sayable{

**void** say();

}

**public** **class** InstanceMethodReference {

**public** **void** saySomething(){

        System.out.println("Hello, this is non-static method.");

    }

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

InstanceMethodReference methodReference = **new** InstanceMethodReference(); // Creating object

        // Referring non-static method using reference

            Sayable sayable = methodReference::saySomething;

        // Calling interface method

            sayable.say();

            // Referring non-static method using anonymous object

            Sayable sayable2 = **new** InstanceMethodReference()::saySomething; // You can use anonymous object also

            // Calling interface method

            sayable2.say();

    }

}

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.

**public** **class** InstanceMethodReference2 {

**public** **void** printnMsg(){

        System.out.println("Hello, this is instance method");

    }

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

    Thread t2=**new** Thread(**new** InstanceMethodReference2()::printnMsg);

        t2.start();

    }

}

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.

**import** java.util.function.BiFunction;

**class** Arithmetic{

**public** **int** add(**int** a, **int** b){

**return** a+b;

}

}

**public** **class** InstanceMethodReference3 {

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

BiFunction<Integer, Integer, Integer>adder = **new** Arithmetic()::add;

**int** result = adder.apply(10, 20);

System.out.println(result);

}

}

Output:

30

## **3) 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

ClassName::**new**

### Example

**interface** Messageable{

    Message getMessage(String msg);

}

**class** Message{

    Message(String msg){

        System.out.print(msg);

    }

}

**public** **class** ConstructorReference {

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

        Messageable hello = Message::**new**;

        hello.getMessage("Hello");

    }

}

Output:

Hello

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.

**import** java.util.\*;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList = **new** ArrayList<Float>();

**for**(Product product: productsList){

            // filtering data of list

**if**(product.price<30000){

                productPriceList.add(product.price);    // adding price to a productPriceList

            }

        }

        System.out.println(productPriceList);   // displaying data

    }

}

**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.

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList2 =productsList.stream()

                                     .filter(p -> p.price > 30000)// filtering data

                                     .map(p->p.price)        // fetching price

                                     .collect(Collectors.toList()); // collecting as list

        System.out.println(productPriceList2);

    }

}

**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.

**import** java.util.stream.\*;

**public** **class** JavaStreamExample {

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

        Stream.iterate(1, element->element+1)

        .filter(element->element%5==0)

        .limit(5)

        .forEach(System.out::println);

    }

}

**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.

**import** java.util.\*;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        // This is more compact approach for filtering data

        productsList.stream()

                             .filter(product -> product.price == 30000)

                             .forEach(product -> System.out.println(product.name));

    }

}

**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.

**import** java.util.\*;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        // This is more compact approach for filtering data

        Float totalPrice = productsList.stream()

                    .map(product->product.price)

                   .reduce(0.0f,(sum, price)->sum+price);   // accumulating price

        System.out.println(totalPrice);

        // More precise code

**float** totalPrice2 = productsList.stream()

                .map(product->product.price)

                .reduce(0.0f,Float::sum);   // accumulating price, by referring methodof Float class

        System.out.println(totalPrice2);

    }

}

**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.

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

       productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        // Using Collectors's method to sum the prices.

**double** totalPrice3 = productsList.stream()

                        .collect(Collectors.summingDouble(product->product.price));

        System.out.println(totalPrice3);

    }

}

**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.

**import** java.util.\*;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        // max() method to get max Product price

        Product productA = productsList.stream().max((product1, product2)->product1.price > product2.price ? 1: -1).get();

        System.out.println(productA.price);

        // min() method to get min Product price

        Product productB = productsList.stream().min((product1, product2)->product1.price > product2.price ? 1: -1).get();

        System.out.println(productB.price);

    }

}

**Output:**

90000.0

25000.0

Java Stream Example: count() Method in Collection

**import** java.util.\*;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        // count number of products based on the filter

**long** count = productsList.stream()

                    .filter(product->product.price<30000)

                    .count();

        System.out.println(count);

    }

}

**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

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        // Converting product List into Set

        Set<Float> productPriceList =

            productsList.stream()

            .filter(product->product.price < 30000)   // filter product on the base of price

            .map(product->product.price)

            .collect(Collectors.toSet());   // collect it as Set(remove duplicate elements)

        System.out.println(productPriceList);

    }

}

**Output:**

[25000.0, 28000.0]

Java Stream Example : Convert List into Map

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        // Converting Product List into a Map

        Map<Integer,String> productPriceMap =

            productsList.stream()

                        .collect(Collectors.toMap(p->p.id, p->p.name));

        System.out.println(productPriceMap);

    }

}

**Output:**

{1=HP Laptop, 2=Dell Laptop, 3=Lenevo Laptop, 4=Sony Laptop, 5=Apple Laptop}

Method Reference in stream

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

**public** **int** getId() {

**return** id;

    }

**public** String getName() {

**return** name;

    }

**public** **float** getPrice() {

**return** price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList =

                productsList.stream()

                            .filter(p -> p.price > 30000) // filtering data

                            .map(Product::getPrice)         // fetching price by referring getPrice method

                            .collect(Collectors.toList());  // collecting as list

        System.out.println(productPriceList);

    }

}

**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.

Prime Ministers of India | List of Prime Minister of India (1947-2020)

### Return

It returns a new stream.

### Java Stream filter() example

In the following example, we are fetching and iterating filtered data.

**import** java.util.\*;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        productsList.stream()

                    .filter(p ->p.price> 30000)   // filtering price

                    .map(pm ->pm.price)          // fetching price

                    .forEach(System.out::println);  // iterating price

    }

}

Output:

90000.0

### Java Stream filter() example 2

In the following example, we are fetching filtered data as a list.

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> pricesList =  productsList.stream()

                    .filter(p ->p.price> 30000)   // filtering price

                    .map(pm ->pm.price)          // fetching price

                    .collect(Collectors.toList());

        System.out.println(pricesList);

    }

}

Output:

[90000.0]

# 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 Example: Fetching data as a List

**import** java.util.stream.Collectors;

**import** java.util.List;

**import** java.util.ArrayList;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** CollectorsExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList =

                productsList.stream()

                            .map(x->x.price)         // fetching price

                            .collect(Collectors.toList());  // collecting as list

        System.out.println(productPriceList);

    }

}

Output:

[25000.0, 30000.0, 28000.0, 28000.0, 90000.0]

Java Collectors Example: Converting Data as a Set

**import** java.util.stream.Collectors;

**import** java.util.Set;

**import** java.util.List;

**import** java.util.ArrayList;

classProduct{

    intid;

    String name;

    floatprice;

**public** Product(intid, String name, floatprice) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

Public class CollectorsExample {

    Public static void main(String[] args) {

        List<Product>productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(newProduct(1,"HP Laptop",25000f));

        productsList.add(newProduct(2,"Dell Laptop",30000f));

        productsList.add(newProduct(3,"Lenevo Laptop",28000f));

        productsList.add(newProduct(4,"Sony Laptop",28000f));

        productsList.add(newProduct(5,"Apple Laptop",90000f));

        Set<Float>productPriceList =

                productsList.stream()

                            .map(x->x.price)         // fetching price

                            .collect(Collectors.toSet());   // collecting as list

        System.out.println(productPriceList);

    }

}

Output:

HTML Tutorial

[25000.0, 30000.0, 28000.0, 90000.0]

Java Collectors Example: using sum method

**import** java.util.stream.Collectors;

**import** java.util.List;

**import** java.util.ArrayList;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** CollectorsExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        Double sumPrices =

                productsList.stream()

                      .collect(Collectors.summingDouble(x->x.price));  // collecting as list

       System.out.println("Sum of prices: "+sumPrices);

        Integer sumId =

                productsList.stream().collect(Collectors.summingInt(x->x.id));

        System.out.println("Sum of id's: "+sumId);

    }

}

Output:

Sum of prices: 201000.0

Sum of id's: 15

Java Collectors Example: Getting Product Average Price

**import** java.util.stream.Collectors;

**import** java.util.List;

**import** java.util.ArrayList;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** CollectorsExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        Double average = productsList.stream()

                         .collect(Collectors.averagingDouble(p->p.price));

        System.out.println("Average price is: "+average);

    }

}

Output:

Average price is: 40200.0

Java Collectors Example: Counting Elements

**import** java.util.stream.Collectors;

**import** java.util.List;

**import** java.util.ArrayList;

**class** Product{

    int id;

 String name;

    floatprice;

**public** Product(intid, String name, floatprice) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

public int getId() {

        return id;

}

**public** String getName() {

        return name;

    }

    public float getPrice() {

       return price;

    }

}

public class CollectorsExample {

    public static void main(String[] args) {

        List<Product>productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        Long noOfElements = productsList.stream()

                               .collect(Collectors.counting());

        System.out.println("Total elements : "+noOfElements);

    }

}

Output:

Total elements : 5

# Java Date and Time

The java.time, java.util, java.sql and java.text packages contains classes for representing date and time. Following classes are important for dealing with date in java.

## **Java 8 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.

* [java.time.LocalDate class](https://www.javatpoint.com/java-localdate)
* [java.time.LocalTime class](https://www.javatpoint.com/java-localtime)
* [java.time.LocalDateTime class](https://www.javatpoint.com/java-localdatetime)
* [java.time.MonthDay class](https://www.javatpoint.com/java-monthday)
* [java.time.OffsetTime class](https://www.javatpoint.com/java-offsettime)
* [java.time.OffsetDateTime class](https://www.javatpoint.com/java-offsetdatetime)
* [java.time.Clock class](https://www.javatpoint.com/java-clock)
* [java.time.ZonedDateTime class](https://www.javatpoint.com/java-zoneddatetime)
* [java.time.ZoneId class](https://www.javatpoint.com/java-zoneid)
* [java.time.ZoneOffset class](https://www.javatpoint.com/java-zoneoffset)
* [java.time.Year class](https://www.javatpoint.com/java-year)
* [java.time.YearMonth class](https://www.javatpoint.com/java-yearmonth)
* [java.time.Period class](https://www.javatpoint.com/java-period)
* [java.time.Duration class](https://www.javatpoint.com/java-duration)
* [java.time.Instant class](https://www.javatpoint.com/java-instant)
* [java.time.DayOfWeek enum](https://www.javatpoint.com/java-dayofweek-enum)
* [java.time.Month enum](https://www.javatpoint.com/java-month-enum)

# Java LocalDate class

Java LocalDate class is an immutable class that represents Date with a default format of yyyy-MM-dd. It inherits Object class and implements the ChronoLocalDate interface

Example:

**import** java.time.LocalDate;

**public** **class** LocalDateExample1 {

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

    LocalDate date = LocalDate.now();

    LocalDate yesterday = date.minusDays(1);

    LocalDate tomorrow = yesterday.plusDays(2);

    System.out.println("Today date: "+date);

    System.out.println("Yesterday date: "+yesterday);

    System.out.println("Tommorow date: "+tomorrow);

  }

}

Output:

Today date: 2021-07-21

Yesterday date: 2021-07-20

Tommorow date: 2021-07-22

## **Java LocalDate Example: isLeapYear()**

**import** java.time.LocalDate;

**public** **class** LocalDateExample2 {

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

    LocalDate date1 = LocalDate.of(2017, 1, 13);

    System.out.println(date1.isLeapYear());

    LocalDate date2 = LocalDate.of(2016, 9, 23);

    System.out.println(date2.isLeapYear());

  }

}

Output:

False

true

## **Java LocalDate Example: atTime()**

**import** java.time.\*;

**public** **class** LocalDateExample3 {

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

 LocalDate date = LocalDate.*of*(2021, 7, 21);

LocalDateTime datetime = date.atTime(1,50,9);

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

  }

}

Output:

2021-07-21T01:50:09

2017-01-13T01:50:09

# Java LocalTime class

Java LocalTime class is an immutable class that represents time with a default format of hour-minute-second. It inherits Object class and implements the Comparable interface.

## **Java LocalTime Example: now()**

**import** java.time.LocalTime;

**public** **class** LocalTimeExample1 {

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

    LocalTime time = LocalTime.now();

    System.out.println(time);

  }

}

Output:

15:19:47.459

## **Java LocalTime Example: of()**

**import** java.time.LocalTime;

**public** **class** LocalTimeExample2 {

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

    LocalTime time = LocalTime.of(10,43,12);

    System.out.println(time);

  }

}

Output:

10:43:12

## **Java LocalTime Example: minusHours() and minusMinutes()**

**import** java.time.LocalTime;

**public** **class** LocalTimeExample3 {

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

    LocalTime time1 = LocalTime.of(10,43,12);

    System.out.println(time1);

    LocalTime time2=time1.minusHours(2);

    LocalTime time3=time2.minusMinutes(34);

    System.out.println(time3);

  }

}

Output:

10:43:12

08:09:12

## **Java LocalTime Example: plusHours() and plusMinutes()**

**import** java.time.LocalTime;

**public** **class** LocalTimeExample4 {

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

    LocalTime time1 = LocalTime.of(10,43,12);

    System.out.println(time1);

    LocalTime time2=time1.plusHours(4);

    LocalTime time3=time2.plusMinutes(18);

    System.out.println(time3);

  }

}

Output:

10:43:12

15:01:12

## **Java LocalTime Example**

**import** java.time.\*;

**import** java.time.temporal.ChronoUnit;

**public** **class** LocalTimeExample5 {

**public** **static** **void** main(String... args) {

    ZoneId zone1 = ZoneId.of("Asia/Kolkata");

   ZoneId zone2 = ZoneId.of("Asia/Tokyo");

    LocalTime time1 = LocalTime.now(zone1);

    System.out.println("India Time Zone: "+time1);

    LocalTime time2 = LocalTime.now(zone2);

    System.out.println("Japan Time Zone: "+time2);

**long** hours = ChronoUnit.HOURS.between(time1, time2);

    System.out.println("Hours between two Time Zone: "+hours);

**long** minutes = ChronoUnit.MINUTES.between(time1, time2);

    System.out.println("Minutes between two time zone: "+minutes);

  }

}

Output:

India Time Zone: 14:56:43.087

Japan Time Zone: 18:26:43.103

Hours between two Time Zone: 3

Minutes between two time zone: 210

# Java LocalDateTime class

Java LocalDateTime class is an immutable date-time object that represents a date-time, with the default format as yyyy-MM-dd-HH-mm-ss.zzz. It inherits object class and implements the ChronoLocalDateTime interface.

## **Java LocalDateTime Example**

**import** java.time.LocalDateTime;

**import** java.time.format.DateTimeFormatter;

**public** **class** LocalDateTimeExample1 {

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

        LocalDateTime now = LocalDateTime.now();

        System.out.println("Before Formatting: " + now);

        DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

        String formatDateTime = now.format(format);

        System.out.println("After Formatting: " + formatDateTime);

    }

}

Output:

Before Formatting: 2021-07-21T06:45:49.432

After Formatting: 21-07-2021 06:45:49

## **Java LocalDateTime Example: now()**

**import** java.time.LocalDateTime;

**import** java.time.format.DateTimeFormatter;

**public** **class** LocalDateTimeExample2 {

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

    LocalDateTime datetime1 = LocalDateTime.now();

    DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    String formatDateTime = datetime1.format(format);

    System.out.println(formatDateTime);

  }

}

Output: Catch

14-01-2017 11:42:32

## **Java LocalDateTime Example: get()**

**import** java.time.LocalDateTime;

**import** java.time.temporal.ChronoField;

**public** **class** LocalDateTimeExample3 {

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

    LocalDateTime a = LocalDateTime.of(2021, 7, 21, 15, 56);

    System.out.println(a.get(ChronoField.DAY\_OF\_WEEK));

    System.out.println(a.get(ChronoField.DAY\_OF\_YEAR));

    System.out.println(a.get(ChronoField.DAY\_OF\_MONTH));

    System.out.println(a.get(ChronoField.HOUR\_OF\_DAY));

    System.out.println(a.get(ChronoField.MINUTE\_OF\_DAY));

  }

}

Output:

3

202

21

15

956

956

## **Java LocalDateTime Example: minusDays()**

**import** java.time.LocalDateTime;

**import** java.time.format.DateTimeFormatter;

**public** **class** LocalDateTimeExample4 {

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

  LocalDateTime datetime1 = LocalDateTime.of(2021, 7, 21, 10, 34);

  LocalDateTime datetime2 = datetime1.minusDays(100);

  System.out.println("Before Formatting: " + datetime2);

  DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");

  String formatDateTime = datetime2.format(format);

  System.out.println("After Formatting: " + formatDateTime );

}

}

Output:

Before Formatting: 2021-04-12T10:34

After Formatting: 12-04-2021 10:34

## **Java LocalDateTime Example: plusDays()**

**import** java.time.LocalDateTime;

**import** java.time.format.DateTimeFormatter;

**public** **class** LocalDateTimeExample5 {

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

  LocalDateTime datetime1 = LocalDateTime.of(2021, 7, 21, 10, 34);

  LocalDateTime datetime2 = datetime1.plusDays(120);

  System.out.println("Before Formatting: " + datetime2);

  DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");

  String formatDateTime = datetime2.format(format);

  System.out.println("After Formatting: " + formatDateTime );

}

}

Output:

Before Formatting: 2021-11-18T10:34

After Formatting: 18-11-2021 10:34

# Java MonthDay class

Java MonthDay class is an immutable date-time object that represents the combination of a month and day-of-month. It inherits Object class and implements the Comparable interface.

## **Java MonthDay class Example**

**import** java.time.\*;

**public** **class** MonthDayExample1 {

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

    MonthDay month = MonthDay.now();

    LocalDate date = month.atYear(1994);

    System.out.println(date);

  }

}

Output:

1994-01-17

## **Java MonthDay class Example: isValidYear()**

**import** java.time.\*;

**public** **class** MonthDayExample2 {

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

    MonthDay month = MonthDay.now();

**boolean** b = month.isValidYear(2012);

    System.out.println(b);

  }

}

Output:

True

## **Java MonthDay class Example: get()**

**import** java.time.\*;

**import** java.time.temporal.\*;

**public** **class** MonthDayExample3{

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

    MonthDay month = MonthDay.now();

**long** n = month.get(ChronoField.MONTH\_OF\_YEAR);

    System.out.println(n);

  }

}

Output:

1

## **Java MonthDay class Example: range()**

**import** java.time.\*;

**import** java.time.temporal.\*;

**public** **class** MonthDayExample4 {

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

    MonthDay month = MonthDay.now();

    ValueRange r1 = month.range(ChronoField.MONTH\_OF\_YEAR);

    System.out.println(r1);

    ValueRange r2 = month.range(ChronoField.DAY\_OF\_MONTH);

    System.out.println(r2);

  }

}

Output:

1 - 12

1 - 31

# Java OffsetTime class

Java OffsetTime class is an immutable date-time object that represents a time, often viewed as hour-minute-second offset. It inherits Object class and implements the Comparable interface.

## **Java OffsetTime class Example**

**import** java.time.OffsetTime;

**import** java.time.temporal.ChronoField;

**public** **class** OffsetTimeExample1 {

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

    OffsetTime offset = OffsetTime.now();

**int** h = offset.get(ChronoField.HOUR\_OF\_DAY);

    System.out.println(h);

**int** m = offset.get(ChronoField.MINUTE\_OF\_DAY);

    System.out.println(m);

**int** s = offset.get(ChronoField.SECOND\_OF\_DAY);

    System.out.println(s);

  }

}

Output:

16

970

58224

## **Java OffsetTime class Example: getHour()**

**import** java.time.OffsetTime;

**public** **class** OffsetTimeExample2 {

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

    OffsetTime offset = OffsetTime.now();

**int** h = offset.getHour();

    System.out.println(h+ " hour");

  }

}

Output4.3M

119

C++ vs Java

15 hour

## **Java OffsetTime class Example: getMinute()**

**import** java.time.OffsetTime;

**public** **class** OffsetTimeExample3 {

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

    OffsetTime offset = OffsetTime.now();

**int** m = offset.getMinute();

    System.out.println(m+ " minute");

  }

}

Output:

24 minute

## **Java OffsetTime class Example: getSecond()**

**import** java.time.OffsetTime;

**public** **class** OffsetTimeExample4 {

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

    OffsetTime offset = OffsetTime.now();

**int** s = offset.getSecond();

    System.out.println(s+ " second");

  }

}

Output:

8 second

# Java Year class

Java Year class is an immutable date-time object that represents a year. It inherits the Object class and implements the Comparable interface.

## **Java Year Example: now()**

**import** java.time.Year;

**public** **class** YearExample1 {

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

    Year y = Year.now();

    System.out.println(y);

  }

}

Output:

2021

## **Java Year Example: atDay()**

**import** java.time.LocalDate;

**import** java.time.Year;

**public** **class** YearExample2{

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

    Year y = Year.of(2021);

    LocalDate l = y.atDay(123);

    System.out.println(l);

  }

}

Output:

2021-05-03

## **Java Year Example: length()**

**import** java.time.Year;

**public** **class** YearExample3 {

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

    Year year = Year.of(2021);

    System.out.println(year.length());

  }

}

Output:

365

## **Java Year Example: isLeap()**

**import** java.time.Year;

**public** **class** YearExample4 {

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

    Year year = Year.of(2016);

    System.out.println(year.isLeap());

  }

}

Output:

true

# Java YearMonth class

Java YearMonth class is an immutable date-time object that represents the combination of a year and month. It inherits the Object class and implements the Comparable interface.

### Java YearMonth Example: now()

**import** java.time.YearMonth;

**public** **class** YearMonthExample1 {

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

  YearMonth ym = YearMonth.now();

  System.out.println(ym);

}

}

Output:

2021-07

### Java YearMonth Example: format()

**import** java.time.YearMonth;

**import** java.time.format.DateTimeFormatter;

**public** **class** YearMonthExample2 {

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

    YearMonth ym = YearMonth.now();

    String s = ym.format(DateTimeFormatter.ofPattern("MM yyyy"));

    System.out.println(s);

  }

}

Output:

07 2021

### Java YearMonth Example: get()

**import** java.time.YearMonth;

**import** java.time.temporal.ChronoField;

**pub**   **lic** **class** YearMonthExample3 {

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

YearMonth y = YearMonth.now();

**long** l1 = y.get(ChronoField.YEAR);

    System.out.println(l1);

**long** l2 = y.get(ChronoField.MONTH\_OF\_YEAR);

    System.out.println(l2);

  }

}

Output:

2021

1

### Java YearMonth Example: plus()

**import** java.time.\*;

**public** **class** YearMonthExample4 {

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

    YearMonth ym1 = YearMonth.now();

    YearMonth ym2 = ym1.plus(Period.ofYears(2));

    System.out.println(ym2);

  }

}

Output:

2023-07

### Java YearMonth Example: minus()

**import** java.time.\*;

**public** **class** YearMonthExample5 {

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

    YearMonth ym1 = YearMonth.now();

    YearMonth ym2 = ym1.minus(Period.ofYears(2));

    System.out.println(ym2);

  }

}

Output:

2019-07

# Java Period class

Java Period class is used to measures time in years, months and days. It inherits the Object class and implements the ChronoPeriod interface.

### Java Period Example: addTo()

**import** java.time.\*;

**import** java.time.temporal.Temporal;

**public** **class** PeriodExample1 {

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

    Period period = Period.ofDays(24);

    Temporal temp = period.addTo(LocalDate.now());

    System.out.println(temp);

  }

}

Output:

2021-08-14

### Java Period Example: of()

**import** java.time.Period;

**public** **class** PeriodExample2 {

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

  Period period = Period.of(2021,07,21);

  System.out.println(period.toString());

}

}

Output:

P2021Y7M21D

OPtional

**import** java.util.Optional;

**public** **class** OptionalExample {

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

        String[] str = **new** String[10];

        Optional<String> checkNull = Optional.ofNullable(str[5]);

**if**(checkNull.isPresent()){  // check for value is present or not

            String lowercaseString = str[5].toLowerCase();

            System.out.print(lowercaseString);

        }**else**

            System.out.println("string value is not present");

    }

}

**import** java.util.Optional;

**public** **class** OptionalExample {

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

        String[] str = **new** String[10];

        str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index

        Optional<String> checkNull = Optional.ofNullable(str[5]);

        checkNull.ifPresent(System.out::println);   // printing value by using method reference

        System.out.println(checkNull.get());    // printing value by using get method

        System.out.println(str[5].toLowerCase());

    }

}

# 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.

**import** java.util.StringJoiner;

**public** **class** StringJoinerExample {

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

        StringJoiner joinNames = **new** StringJoiner(","); // passing comma(,) as delimiter

        // Adding values to StringJoiner

        joinNames.add("Rahul");

        joinNames.add("Raju");

        joinNames.add("Peter");

        joinNames.add("Raheem");

        System.out.println(joinNames);

    }

}

**import** java.util.StringJoiner;

**public** **class** StringJoinerExample {

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

        StringJoiner joinNames = **new** StringJoiner(",", "[", "]");   // passing comma(,) and square-brackets as delimiter

        // Adding values to StringJoiner

        joinNames.add("Rahul");

        joinNames.add("Raju");

        joinNames.add("Peter");

        joinNames.add("Raheem");

        System.out.println(joinNames);

    }

}

**import** java.util.StringJoiner;

**public** **class** StringJoinerExample {

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

        StringJoiner joinNames = **new** StringJoiner(",", "[", "]");   // passing comma(,) and square-brackets as delimiter

        // Adding values to StringJoiner

        joinNames.add("Rahul");

        joinNames.add("Raju");

        // Creating StringJoiner with :(colon) delimiter

        StringJoiner joinNames2 = **new** StringJoiner(":", "[", "]");  // passing colon(:) and square-brackets as delimiter

        // Adding values to StringJoiner

        joinNames2.add("Peter");

        joinNames2.add("Raheem");

        // Merging two StringJoiner

        StringJoiner merge = joinNames.merge(joinNames2);

        System.out.println(merge);

    }

}