## **Java 8 New Features**

**1) Lambda Expressions ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::**

**2) Functional Interfaces ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::**

**3) Default methods :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::**

**4) Predicates :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::**

**5) Functions ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::**

**6) Double colon operator (::) :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::**

**7) Stream API ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::**

**8) Date and Time API :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::**

java 7 - July 28th 2011

Java 8 - March 18th 2014

Java 9 - September 22nd 2016

Java 10 - 2018

After Java 1.5version, Java 8 is the next major version.

Before Java 8, sun people gave importance only for objects but in 1.8version oracle people gave the importance for functional aspects of programming to bring its benefits to Java. i.e, it doesn't mean Java is functional oriented programming language.

### **Lambda Expression :**

History:

* Lambda calculus is a big change in mathematical world which has been introduced in 1930. Because of benefits of Lambda calculus slowly this concepts started using in programming world.
* LISP is the first programming which uses Lambda Expression.
* The other languages which uses lambda expressions are: C#, .Net, C Objective, C, C++, Python, Ruby etc. and finally in Java also.

The Main Objective of Lambda Expression is to bring benefits of functional programming into Java.

What is Lambda Expression:

* Lambda Expression is just an anonymous (nameless) function. That means the function which doesn't have the name, return type and access modifiers.
* Lambda Expression also known as anonymous functions or closures.

Ex: 1

public void m1() {

sop("hello"");

}

lambda ex:

() ->{

sop("hello"");

}

() -> { sop("hello"); }

() -> sop("hello");

Ex:2

public void add(int a, int b) {

sop(a+b);

}

lambda ex:

(int a, int b) -> sop(a+b);

* If the type of the parameter can be decided by compiler automatically based on the context then we can remove types also.
* The above Lambda expression we can rewrite as

(a,b) -> sop (a+b);

Ex: 3

public String str(String str) {

return str;

}

Lambda ex:

(String str) -> return str;

(str) -> str;

### **Conclusions:**

1. A lambda expression can have zero or more number of parameters (arguments).

Ex:

() -> sop("hello");

(int a ) -> sop(a);

(int a, int b) -> return a+b;

2. Usually we can specify type of parameter. If the compiler expects the type based on the context then we can remove type. i.e., programmer is not required.

Ex:

(int a, int b) -> sop(a+b);

(a,b) -> sop(a+b);

3. If multiple parameters present then these parameters should be separated with comma (,).

4. If zero number of parameters available then we have to use empty parameter [ like ()].

Ex: () -> sop("hello");

5. If only one parameter is available and if the compiler can expect the type then we can remove the type and parenthesis also.

Ex:

(int a) -> sop(a);

(a)-> sop(a);

a -> sop(a);

6. Similar to method body lambda expression body also can contain multiple statements. If more than one statements present then we have to enclose inside within curly braces. If one statement present then curly braces are optional.

7. Once we write lambda expression we can call that expression just like a method, for this functional interfaces are required.

## **Functional Interfaces**

If an interface contain only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

Ex:

1) Runnable -> It contains only run() method

2) Comparable -> It contains only compareTo() method

3) ActionListener -> It contains only actionPerformed()

4) Callable -> It contains only call() method

Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.

Ex:

interface Interf {

public abstract void m1();

default void m2() {

System.out.println ("hello");

}

}

In Java 8, Sun MicroSystem introduced @FunctionalInterface annotation to specify that the interface is Functional Interface.

Ex:

@FunctionalInterface

interface Interf { //This code compiles without any compilation errors.

public void m1();

}

Inside Functional Interface we can take only one abstract method, if we take more than one abstract method then compiler raise an error message that is called we will get compilation error.

Ex:

@FunctionalInterface

interface Interf {

public void m1(); //This code gives compilation error.

public void m2();

}

Inside Functional Interface we have to take exactly only one abstract method. If we are not declaring that abstract method then compiler gives an error message.

Ex:

@FunctionalInterface

interface Interface { //compilation error

}

### 

### **Functional Interface with respect to Inheritance:**

If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also Functional Interface

Ex:

@FunctionalInterface

interface A {

public void methodOne();

}

@FunctionalInterface

interface B extends A { //No Compile Time Error

}

In the child interface we can define exactly same parent interface abstract method.

Ex:

@FunctionalInterface

interface A {

public void methodOne();

}

@FunctionalInterface

interface B extends A {

public void methodOne(); //No Compile Time Error

}

In the child interface we can't define any new abstract methods otherwise child interface won't be Functional Interface and if we are trying to use @FunctionalInterface annotation then compiler gives an error message.

@FunctionalInterface {

interface A {

public void methodOne();

}

@FunctionalInterface

interface B extends A {

public void methodTwo(); //Compile Time Error

}

Ex:

@FunctionalInterface

interface A {

public void methodOne(); //No Compile Time Error

}

interface B extends A {

public void methodTwo(); //This's Normal interface so that code compiles without error

}

In the above example in both parent & child interface we can write any number of default methods and there are no restrictions. Restrictions are applicable only for abstract methods.

## **Functional Interface Vs Lambda Expressions:**

Once we write Lambda expressions to invoke it's functionality, then Functional Interface is required.  
 We can use Functional Interface reference to refer Lambda Expression.  
 Where ever Functional Interface concept is applicable there we can use Lambda Expressions

Ex:1 Without Lambda Expression

Main.java

----------------

**interface** Interf {

**public** **void** methodOne();

}

**public** **class** Demo **implements** Interf {

**public** **void** methodOne() {

System.***out***.println("method one execution");

}

}

Test.java

------------

**public** **class** Test {

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

Interf i = **new** Demo();

i.methodOne();

}

}

Above code With Lambda expression

**interface** Interf {

**public** **void** methodOne();

}

**class** Test {

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

Interf i = () -> System.***out***.println("method one execution");

i.methodOne();

}

}

Ex 2: Without Lambda Expression

Demo.java

-------------------

//Functional interface with a single abstract method

**interface** Interf {

**void** sum(**int** a, **int** b);

}

//Implementation of the interface

**class** Demo **implements** Interf {

**public** **void** sum(**int** a, **int** b) {

System.***out***.println("The sum: " + (a + b));

}

}

Test.java

-------------

**public** **class** Test {

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

// Using Demo class as an implementation of Interf

Interf i = **new** Demo();

i.sum(20, 5); // Output: The sum: 25

}

}

Above code With Lambda Expression

**interface** Interf {

// Method declaration in the functional interface

**void** sum(**int** a, **int** b);

}

**public** **class** Test {

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

// Lambda expression to implement the method from Interf

Interf i = (a, b) -> System.***out***.println("The Sum: " + (a + b));

// Calling the method using the lambda expression

i.sum(5, 10); // Output: The Sum: 15

}

}

--------------------------------------------------------------------------------------------------------------------

Ex 3: Without Lambda Expressions

Main.java

-------------------

**interface** Interf {

**public** **int** square(**int** x);

}

**class** Demo **implements** Interf {

**public** **int** square(**int** x) {

**return** x \* x; // Calculate and return the square of x

}

}

**Test.java**

**---------------**

**public** **class** Test {

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

Interf i = **new** Demo(); // Creating an instance of Demo and referring it by Interf interface

System.***out***.println("The Square of 7 is: " + i.square(7)); // Calling square method using interface reference

}

}

Above code with Lambda Expression

**interface** Interf {

**public** **int** square(**int** x);

}

**public** **class** Test {

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

Interf i = x -> x \* x; // Lambda expression to implement the square method

System.***out***.println("The Square of 5 is: " + i.square(5)); // Calling square method using lambda expression

}

}

Ex 4: Without Lambda expression

**class** MyRunnable **implements** Runnable {

**public** **void** run() {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread");

}

}

}

**public** **class** ThreadDemo {

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

// Create an instance of MyRunnable

Runnable r = **new** MyRunnable();

// Create a new thread using the MyRunnable instance

Thread t = **new** Thread(r);

// Start the thread

t.start();

// Main thread execution

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread");

}

}

}

With Lambda expression

**class** ThreadDemo {

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

// Lambda expression to implement the run() method of Runnable

Runnable r = () -> {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread");

}

};

// Create a new thread with the lambda expression

Thread t = **new** Thread(r);

// Start the thread

t.start();

// Main thread execution

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread");

}

}

}

## **Default Methods**

* Until 1.7 version onwards inside interface we can take only public abstract methods and public static final variables (every method present inside interface is always public and abstract whether we are declaring or not).
* Every variable declared inside interface is always public static final whether we are declaring or not.
* But from 1.8 version onwards in addition to these, we can declare default concrete methods also inside interface, which are also known as defender methods.
* We can declare default method with the keyword "default" as follows

· default void m1(){

· System.out.println ("Default Method");

· }

* Interface default methods are by-default available to all implementation classes. Based on requirement implementation class can use these default methods directly or can override.

Ex:

1) interface Interf {

2) default void m1() {

3) System.out.println("Default Method");

4) }

5) }

6) class Test implements Interf {

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

8) Test t = new Test();

9) t.m1();

10) }

11) }

* Default methods also known as defender methods or virtual extension methods.
* The main advantage of default methods is without effecting implementation classes we can add new functionality to the interface (backward compatibility).

Note: We can't override object class methods as default methods inside interface otherwise we get compile time error.

Ex:

1) interface Interf {

2) default int hashCode() {

3) return 10;

4) }

5) }

CompileTimeError

**Reason:** Object class methods are by-default available to every Java class hence it's not required to bring through default methods.

#### 

#### Default method vs multiple inheritance

Two interfaces can contain default method with same signature then there may be a chance of ambiguity problem (diamond problem) to the implementation class. To overcome this problem compulsory we should override default method in the implementation class otherwise we get compile time error.

1) Eg 1:

2) interface Left {

3) default void m1() {

4) System.out.println("Left Default Method");

5) }

6) }

7)

8) Eg 2:

9) interface Right {

10) default void m1() {

11) System.out.println("Right Default Method");

12) }

13) }

14)

15)

Eg 3:

16) class Test implements Left, Right {}

#### **How to override default method in the implementation class ?**

In the implementation class we can provide complete new implementation or we can call any interface method as follows.

interfacename.super.m1();

Ex:

1) class Test implements Left, Right {

2) public void m1() {

3) System.out.println("Test Class Method"); // OR Left.super.m1();

4) }

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

6) Test t = new Test();

7) t.m1();

8) }

9) }

#### **Differences between interface with default methods and abstract class**

Even though we can add concrete methods in the form of default methods to the interface, it won't be equal to abstract class.

| **Interface with Default Methods** | **Abstract Class** |
| --- | --- |
| Inside interface every variable is always public static final and there is no chance of instance variables | Inside abstract class there may be a chance of instance variables which are required to the child class. |
| Interface never talks about state of Object. | Abstract class can talk about state of Object. |
| Inside interface we can't declare constructors. | Inside abstract class we can declare constructors. |
| Inside interface we can't declare instance and static blocks. | Inside abstract class we can declare instance and static blocks. |
| Functional interface with default methods can refer lambda expression. | Abstract class can't refer lambda Expressions. |
| Inside interface we can't override Object class methods. | Inside abstract class we can override Object class methods. |

**interface with default method != abstract class**

## **Static methods inside interface:**

· From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions.

· Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can't call interface static methods. We should call interface static methods by using interface name.

Ex:

1) interface Interf {

2) public static void sum(int a, int b) {

3) System.out.println("The Sum:"+(a+b));

4) }

5) }

6) class Test implements Interf {

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

8) Test t = new Test();

9) t.sum(10, 20); //CE

10) Test.sum(10, 20); //CE

11) Interf.sum(10, 20);

12) }

13) }

* As interface static methods by default not available to the implementation class, overriding concept is not applicable.
* Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

Ex:1

1) interface Interf {

2) public static void m1() {}

3) }

4) class Test implements Interf {

5) public static void m1() {}

6) }

It's valid but not overriding

Ex:2

1) interface Interf {

2) public static void m1() {}

3) }

4) class Test implements Interf {

5) public void m1() {}

6) }

This's valid but not overriding

Ex3:

1) class P {

2) private void m1() {}

3) }

4) class C extends P {

5) public void m1() {}

6) }

This's valid but not overriding

From 1.8 version onwards we can write main() method inside interface and hence we can run interface directly from the command prompt.

Ex:

1) interface Interf {

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

3) System.out.println("Interface Main Method");

4) }

5) }

At the command prompt:

javac Interf.Java

java

## **Predicate:**

* A predicate is a function with a single argument and returns boolean value.
* To implement predicate functions in Java, Oracle people introduced Predicate interface in 1.8 version (i.e.,Predicate<T>).
* Predicate interface present in Java.util.function package.
* It's a functional interface and it contains only one method i.e., test()

Ex:

interface Predicate<T> {

public boolean test(T t);

}

As predicate is a functional interface and hence it can refers lambda expression

Ex:1 Write a predicate to check whether the given integer is greater than 10 or not.

public boolean test(Integer I) {

if (I >10) {

return true;

} else {

return false;

}

}

(Integer I) -> {

if(I > 10)

return true;

else

return false;

}

I -> (I>10);

Predicate<Integer> p = I ->(I >10);

System.out.println (p.test(100)); //true

System.out.println (p.test(7)); //false

**import** java.util.function.Predicate;

**public** **class** PredicateExample {

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

// Predicate to check if an integer is greater than 10

Predicate<Integer> greaterThan10 = i -> i > 10;

// Test cases

System.***out***.println(greaterThan10.test(5)); // false

System.***out***.println(greaterThan10.test(15)); // true

System.***out***.println(greaterThan10.test(10)); // false (not greater than 10)

}

}

# 1 Write a predicate to check the length of given string is greater than 3 or not.

Predicate<String> p = s -> (s.length() > 3);

System.out.println (p.test("rvkb")); //true

System.out.println (p.test("rk")); //false

**import** java.util.function.Predicate;

**public** **class** StringLengthPredicateExample {

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

// Example strings

String str1 = "Hello";

String str2 = "Hi";

// Predicate to check if string length is greater than 3

Predicate<String> lengthGreaterThan3 = s -> s.length() > 3;

// Test the predicate

System.***out***.println("Is length of \"" + str1 + "\" greater than 3? " + lengthGreaterThan3.test(str1)); // true

System.***out***.println("Is length of \"" + str2 + "\" greater than 3? " + lengthGreaterThan3.test(str2)); // false

}

}

#-2 write a predicate to check whether the given collection is empty or not.

Predicate p = c -> c.isEmpty();

**import** java.util.Collection;

**import** java.util.List;

**import** java.util.function.Predicate;

**public** **class** PredicateExample {

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

// Example collections

Collection<String> collection1 = List.*of*(); // Empty collection

Collection<String> collection2 = List.*of*("apple", "banana", "orange"); // Non-empty collection

// Predicate to check if a collection is empty

Predicate<Collection<?>> isEmptyCollection = c -> c.isEmpty();

// Test cases

System.***out***.println(isEmptyCollection.test(collection1)); // true (empty collection)

System.***out***.println(isEmptyCollection.test(collection2)); // false (non-empty collection)

}

}

Predicate joining  
 It's possible to join predicates into a single predicate by using the following methods.

* and()
* or()
* negate()

these are exactly same as logical AND ,OR complement operators

Ex:

**import** java.util.function.Predicate;

**public** **class** Test {

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

// Array of integers

**int**[] x = { 0, 5, 10, 15, 20, 25, 30 };

// Predicate to check if integer is greater than 10

Predicate<Integer> p1 = i -> i > 10;

// Predicate to check if integer is even

Predicate<Integer> p2 = i -> i % 2 == 0;

// Printing numbers greater than 10

System.***out***.println("The Numbers Greater Than 10:");

*m1*(p1, x); // Calling method m1 with predicate p1

// Printing even numbers

System.***out***.println("The Even Numbers Are:");

*m1*(p2, x); // Calling method m1 with predicate p2

// Printing numbers not greater than 10 using negate()

System.***out***.println("The Numbers Not Greater Than 10:");

*m1*(p1.negate(), x); // Calling method m1 with negated predicate p1

// Printing numbers greater than 10 and even using and()

System.***out***.println("The Numbers Greater Than 10 And Even Are:");

*m1*(p1.and(p2), x); // Calling method m1 with p1 and p2 combined with and()

// Printing numbers greater than 10 or even using or()

System.***out***.println("The Numbers Greater Than 10 OR Even:");

*m1*(p1.or(p2), x); // Calling method m1 with p1 and p2 combined with or()

}

// Method to filter and print elements from array based on predicate

**public** **static** **void** m1(Predicate<Integer> p, **int**[] x) {

**for** (**int** x1 : x) {

**if** (p.test(x1)) {

System.***out***.println(x1);

}

}

}

}

# **Functions**

· **Functions are exactly same as predicates except that functions can return any type of result but function should (can) return only one value and that value can be any type as per our requirement.**

· **To implement functions oracle people introduced Function interface in 1.8version.**

· **Function interface present in *Java.util.function* package.**

· **Functional interface contains only one method i.e., apply()**

interface function(T{ public Rapply(T t);

}

**public** **class** Test {

// Method to calculate the square of an integer

**public** **static** **int** squareIt(**int** n) {

**return** n \* n; // Return the square of n

}

// Main method where the program starts execution

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

// Print the square of 4

System.***out***.println("The square of 4: " + *squareIt*(4));

// Print the square of 5

System.***out***.println("The square of 5: " + *squareIt*(5));

}

}

**Same program we are using functions:**

----------------------------------

**import** java.util.function.\*;

**class** Test {

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

// Function to calculate the square of an integer

Function<Integer, Integer> f = i -> i \* i;

// Print the square of 40

System.***out***.println("The square of 40: " + f.apply(40));

// Print the square of 50

System.***out***.println("The square of 50: " + f.apply(50));

}

}

Assignment: Write a function to find length of given input string.

**import** java.util.function.\*;

**class** Test {

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

// Function to calculate the length of a string

Function<String, Integer> f = s -> s.length();

// Test cases

System.***out***.println(f.apply("malli")); // Output: 5 (length of "malli")

System.***out***.println(f.apply("Soft")); // Output: 4 (length of "Soft")

}

}

Note: Function is a functional interface and hence it can refer lambda expression.

###### **Differences between predicate and function**

| **Predicate** | **Function** |
| --- | --- |
| **To implement conditional checks We should go for predicate** | **To perform certain operation And to return some result we Should go for**  **function.** |
| **Predicate can take one type Parameter which represents Input argument type.**  **Predicate<T>** | **Function can take 2 type Parameters. First one represent Input argument type and Second one represent return Type.**  **Function<T,R>** |
| **Predicate interface defines**  **only one method called test()** | **Function interface defines only one**  **Method called apply().** |
| **public boolean test(T t)** | **public R apply(T t)** |
| **Predicate can return only boolean value.** | **Function can return any type of value** |

Note: Predicate is a boolean valued function and(), or(), negate() are default methods present inside Predicate interface.

Write a function program to display the grade marks of student

**import** java.util.function.\*;

**class** Student {

String name;

**int** marks;

Student(String name, **int** marks) {

**this**.name = name;

**this**.marks = marks;

}

}

**public** **class** Test {

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

// Function to calculate grade based on marks

Function<Student, String> f = s -> {

**int** marks = s.marks;

String grade = "";

**if** (marks >= 80)

grade = "A[Distinction]";

**else** **if** (marks >= 60)

grade = "B[First class]";

**else** **if** (marks >= 50)

grade = "C[Second class]";

**else** **if** (marks >= 35)

grade = "D[Third class]";

**else**

grade = "E[Failed]";

**return** grade;

};

// Array of Student objects

Student[] students = { **new** Student("malli", 100), **new** Student("Tillu", 65), **new** Student("radhika", 55),

**new** Student("Arjun", 25), **new** Student("Hari", 47) };

// Iterating through each student and printing details

**for** (Student student : students) {

System.***out***.println("Student name: " + student.name);

System.***out***.println("Student marks: " + student.marks);

System.***out***.println("Student grade: " + f.apply(student));

System.***out***.println(); // Empty line for separation

}

}

}

Function chaining:-

**import** java.util.function.\*;

**public** **class** Test {

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

// Define two Functions

Function<Integer, Integer> f1 = i -> 2 \* i; // Multiplies input by 2

Function<Integer, Integer> f2 = i -> i \* i \* i; // Computes cube of input

// Using andThen method: f1.andThen(f2)

// This will first apply f1 and then apply f2 to the result of f1

System.***out***.println(f1.andThen(f2).apply(2)); // Output: f2(f1(2)) = f2(4) = 64

// Using compose method: f1.compose(f2)

// This will first apply f2 and then apply f1 to the result of f2

System.***out***.println(f1.compose(f2).apply(2)); // Output: f1(f2(2)) = f1(8) = 16

}

}

**Consumer:-**

**consumer is always taking some input value and won’t return anything. just it print the statements never return anything.**

**interface Consumer<T>**

**{**

**Public void accept(T t)**

**}**

**Ex**

**-----**

**import java.util.function.\*;**

**class Test {**

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

**// Consumer to print the input string**

**Consumer<String> c = s -> System.out.println(s);**

**// Using the Consumer to print "malli"**

**c.accept("malli");**

**}**

**}**

**Example-2**

**--------------**

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Consumer;

**public** **class** ConsumerExample {

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

// Example list of strings

List<String> languages = Arrays.*asList*("Java", "Python", "JavaScript", "C#", "Ruby");

// Example using Consumer to print each element

System.***out***.println("Printing all languages:");

*forEach*(languages, (str) -> System.***out***.println(str));

// Example using Consumer to convert each string to uppercase

System.***out***.println("\nConverting all languages to uppercase:");

*forEach*(languages, (str) -> System.***out***.println(str.toUpperCase()));

// Example using Consumer to append "!" to each string

System.***out***.println("\nAppending '!' to each language:");

*forEach*(languages, (str) -> System.***out***.println(str + "!"));

}

// Generic method to apply a Consumer to each element of a list

**public** **static** <T> **void** forEach(List<T> list, Consumer<T> consumer) {

**for** (T t : list) {

consumer.accept(t);

}

}

}

**Example-3**

**-------------**

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Consumer;

**public** **class** ConsumerExample {

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

// Create a list of integers

List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 5);

// Example of a Consumer: Print each element in the list

Consumer<Integer> printConsumer = num -> System.***out***.println(num);

// Iterate through the list and apply the Consumer to each element

System.***out***.println("Printing elements using Consumer:");

numbers.forEach(printConsumer);

// Example of chaining Consumers using andThen method

Consumer<Integer> multiplyConsumer = num -> System.***out***.println(num \* 2);

System.***out***.println("Printing elements after multiplication using andThen:");

numbers.forEach(printConsumer.andThen(multiplyConsumer));

}

}

**Supplier:-**

**Just supply my required objects and it won’t take any input the we should go for supplier method.**

**Interface supplier<R>**

**{**

**public R get();**

**}**

**Ex: write a java program to supply system date**

**import** java.util.function.\*;

**import** java.util.Date;

**public** **class** Test {

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

// Supplier to generate current date and time

Supplier<Date> s = () -> **new** Date();

// Using the Supplier to get current date and time

System.***out***.println(s.get()); // Output 1

System.***out***.println(s.get()); // Output 2

System.***out***.println(s.get()); // Output 3

System.***out***.println(s.get()); // Output 4

}

}

**Ex: Write a java program to supplie the OTP.**

**import** java.util.function.\*;

**import** java.util.Date;

**public** **class** Test {

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

// Supplier to generate OTPs

Supplier<String> s = () -> {

String otp = "";

// Generate a 6-digit OTP

**for**(**int** i = 0; i < 6; i++) {

otp = otp + (**int**)(Math.*random*() \* 10); // Append random digits (0-9) to OTP

}

**return** otp; // Return the generated OTP

};

// Generate and print OTPs

System.***out***.println(s.get());

System.***out***.println(s.get());

System.***out***.println(s.get());

System.***out***.println(s.get());

System.***out***.println(s.get());

}

}

**Example**

**------------**

**import** java.util.function.\*;

**public** **class** SupplierExample {

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

// Supplier to generate even numbers

Supplier<Integer> evenNumberSupplier = **new** Supplier<Integer>() {

**private** **int** currentEven = 0; // Initial even number

@Override

**public** Integer get() {

// Return the next even number, and then increment for the next call

currentEven += 2;

**return** currentEven;

}

};

// Print the next 5 even numbers using the Supplier

System.***out***.println("Generating even numbers:");

**for** (**int** i = 0; i < 5; i++) {

**int** nextEven = evenNumberSupplier.get();

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

}

}

}

**BiPredicate:-**

Normal predicate we can take only one input argument and perform some conditional check. Sometimes our programming requirement is we have to take 2 input arguments and perform some conditional check, for this requirement we should go for BiPredicate.

Bipredicate is exactly same as Predicate except that it will take 2 input arguments.

Interface BiPredicate<T1,T2>

{

Public Boolean test(T1 t1,T2 t2);

}

Ex: to check the sum of 2 given integers is even or not by using BiPredicate

**import** java.util.function.\*;

**public** **class** Test {

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

// BiPredicate to check if the sum of two integers is even

BiPredicate<Integer, Integer> p = (a, b) -> (a + b) % 2 == 0;

// Test the BiPredicate with different pairs of integers

System.***out***.println(p.test(10, 20)); // Output: true (10 + 20 = 30, which is even)

System.***out***.println(p.test(15, 20)); // Output: false (15 + 20 = 35, which is odd)

}

}

**BiFunction**

**---------------**

In Java 1.8, a **BiFunction** is a functional interface that takes two arguments and produces a result. It’s part of the java.util.function package and can be very useful in various scenarios where you need to combine two inputs to produce an output.

**Declaration**

The BiFunction interface is defined as follows:

@FunctionalInterface

public interface BiFunction<T, U, R> {

R apply(T t, U u);

}

**Example Program**

Here’s an example program that demonstrates how to use BiFunction. This program takes two integers and returns their sum and product:

import java.util.function.BiFunction;

public class BiFunctionExample {

public static void main(String[] args) {

// Creating a BiFunction to calculate the sum of two integers

BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;

// Creating a BiFunction to calculate the product of two integers

BiFunction<Integer, Integer, Integer> product = (a, b) -> a \* b;

// Testing the BiFunctions

int num1 = 5;

int num2 = 10;

int sumResult = sum.apply(num1, num2);

int productResult = product.apply(num1, num2);

System.out.println("Sum: " + sumResult); // Output: Sum: 15

System.out.println("Product: " + productResult); // Output: Product: 50

}

}

**Explanation**

1. **BiFunction Declaration**: We declare two BiFunctions, one for sum and another for product.

2. **Lambda Expressions**: Each BiFunction is implemented using a lambda expression that takes two integers and returns their sum or product.

3. **Using apply Method**: We call the apply method of each BiFunction to perform the operations with the provided integers.

**BiConsumer**

**---------------**

In Java 1.8, a **BiConsumer** is a functional interface that accepts two arguments and performs an operation on them, without returning a result. It’s part of the java.util.function package and is often used for operations like processing data or updating values.

**Declaration**

The BiConsumer interface is defined as follows:

@FunctionalInterface

public interface BiConsumer<T, U> {

void accept(T t, U u);

}

**Example Program**

Here’s an example program that demonstrates how to use BiConsumer. This program takes two strings and prints them in a formatted way:

import java.util.function.BiConsumer;

public class BiConsumerExample {

public static void main(String[] args) {

// Creating a BiConsumer to print two strings

BiConsumer<String, String> printFullName = (firstName, lastName) ->

System.out.println("Full Name: " + firstName + " " + lastName);

// Testing the BiConsumer

String firstName = "John";

String lastName = "Doe";

printFullName.accept(firstName, lastName); // Output: Full Name: John Doe

}

}

**Explanation**

1. **BiConsumer Declaration**: We declare a BiConsumer that takes two String arguments.

2. **Lambda Expression**: The BiConsumer is implemented using a lambda expression that formats and prints the full name.

3. **Using accept Method**: We call the accept method of the BiConsumer to perform the operation with the provided strings.

**BiSupplier**

**---------------**

In Java 1.8, a **BiSupplier** isn't a standard interface in the java.util.function package, but you can create a custom one if needed. The concept of a "supplier" is that it provides a result without taking any input parameters. A BiSupplier would ideally take two arguments and return a result.

**Custom BiSupplier Interface**

You can define a BiSupplier interface like this:

@FunctionalInterface

public interface BiSupplier<T, U, R> {

R get(T t, U u);

}

**Example Program**

Here’s an example of how you might use a custom BiSupplier to combine two values of different types:

public class BiSupplierExample {

public static void main(String[] args) {

// Creating a BiSupplier to concatenate two strings

BiSupplier<String, String, String> concatSupplier = (s1, s2) -> s1 + " " + s2;

// Using the BiSupplier

String firstName = "Jane";

String lastName = "Doe";

String fullName = concatSupplier.get(firstName, lastName);

System.out.println("Full Name: " + fullName); // Output: Full Name: Jane Doe

}

}

**Explanation**

1. **Custom BiSupplier Declaration**: We define a BiSupplier interface that takes two arguments and returns a result.

2. **Lambda Expression**: The concatSupplier uses a lambda expression to concatenate the two string arguments.

3. **Using get Method**: We call the get method of the BiSupplier with two strings, producing a full name.

### **Note**

While Java doesn’t provide a built-in BiSupplier, this pattern can be useful for specific cases where you want to supply a result based on two input parameters. The standard Java libraries mainly provide single-parameter suppliers, but you can extend functionality as shown above.

### **Conclusion**

Using a custom BiSupplier interface allows you to create flexible and reusable logic that combines two inputs to produce an output. This can be especially useful in more complex data processing scenarios.

**Streams**

To process objects of the collection, in 1.8 version Streams concept introduced.

### **What is the differences between java.util.stream and java.io streams ?**

* java.util streams meant for processing objects from the collection. i.e, it represents a stream of objects from the collection
* but java.io streams meant for processing binary and character data with respect to file. i.e., it represents stream of binary data or character data from the file.  
   hence java.io streams and java.util streams both are different.

### **What is the difference between collection and stream ?**

* If we want to represent a group of individual objects as a single entity then we should go for collection.
* If we want to process a group of objects from the collection then we should go for streams.
* We can create a stream object to the collection by using stream() method of Collection interface.
* stream() method is a default method added to the Collection in 1.8 version.

· default Stream stream()

· Ex:

· Stream s = c.stream();

* Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.
* We can process the objects in the following 2 phases

1. Configuration

2. Processing

### 

### **1) Configuration:**

We can configure either by using filter mechanism or by using map mechanism.

#### Filtering:

* We can configure a filter to filter elements from the collection based on some boolean condition by using filter() method of Stream interface.

public Stream filter(Predicate<T> t)

here (Predicate<T > t ) can be a boolean valued function/lambda expression

Ex:

Stream s = c.stream();

Stream s1 = s.filter(i -> i%2==0);

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** FilterExample {

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

// Create a list of integers

List<Integer> numbers = **new** ArrayList<>();

**for** (**int** i = 1; i <= 10; i++) {

numbers.add(i);

}

// Create a stream from the list

Stream<Integer> stream = numbers.stream();

// Use filter method with a Predicate lambda expression to filter even numbers

Stream<Integer> evenNumbersStream = stream.filter(i -> i % 2 == 0);

// Print the filtered even numbers

evenNumbersStream.forEach(System.***out***::println);

}

}

Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

#### Mapping:

* If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

public Stream map (Function f);

It can be lambda expression also

Ex:

Stream s = c.stream();

Stream s1 = s.map(i-> i+10);

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** MapExample {

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

// Create a list of integers

List<Integer> numbers = **new** ArrayList<>();

**for** (**int** i = 1; i <= 5; i++) {

numbers.add(i);

}

// Create a stream from the list

Stream<Integer> stream = numbers.stream();

// Use map method with a lambda expression to add 10 to each element

Stream<Integer> transformedStream = stream.map(i -> i + 10);

// Print the transformed elements

transformedStream.forEach(System.***out***::println);

}

}

Once we performed configuration we can process objects by using several methods.

### **2) Processing**

* processing by collect() method
* Processing by count() method
* Processing by sorted() method
* Processing by min() and max() methods
* forEach() method
* toArray() method
* Stream.of()method

#### I. Processing by collect() method

This method collects the elements from the stream and adding to the specified to the collection indicated (specified) by argument.

Ex 1: To collect only even numbers from the array list  
  
 Approach-1: Without Streams

**import** java.util.\*;

**public** **class** Test {

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

// Create ArrayList l1 and add numbers from 0 to 10

ArrayList<Integer> l1 = **new** ArrayList<Integer>();

**for** (**int** i = 0; i <= 10; i++) {

l1.add(i);

}

// Print the contents of l1

System.***out***.println(l1); // Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// Create ArrayList l2 to store even numbers from l1

ArrayList<Integer> l2 = **new** ArrayList<Integer>();

**for** (Integer i : l1) {

// Check if the number is even (i % 2 == 0)

**if** (i % 2 == 0) {

l2.add(i); // Add even number to l2

}

}

// Print the contents of l2

System.***out***.println(l2); // Output: [0, 2, 4, 6, 8, 10]

}

}

Approach-2: With Streams

**import** java.util.\*;

**import** java.util.stream.\*;

**public** **class** Test {

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

// Creating an ArrayList and adding integers from 0 to 10

ArrayList<Integer> l1 = **new** ArrayList<Integer>();

**for** (**int** i = 0; i <= 10; i++) {

l1.add(i);

}

// Printing the initial ArrayList

System.***out***.println("Original List: " + l1);

// Using Stream API to filter even numbers and collect them into a new List

List<Integer> l2 = l1.stream().filter(i -> i % 2 == 0) // Filter condition: keep even numbers only

.collect(Collectors.*toList*()); // Collect filtered elements into a new List

// Printing the filtered List of even numbers

System.***out***.println("List of Even Numbers: " + l2);

}

}

Ex: Program for map() and collect() Method

----------------------------------------------------------------

**import** java.util.\*;

**import** java.util.stream.\*;

**public** **class** Test {

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

// Create an ArrayList of Strings

ArrayList<String> l = **new** ArrayList<String>();

l.add("rvk");

l.add("rk");

l.add("rkv");

l.add("rvki");

l.add("rvkir");

// Print the original ArrayList

System.***out***.println(l); // Output: [rvk, rk, rkv, rvki, rvkir]

// Use Stream and map to convert each string to uppercase

List<String> l2 = l.stream().map(s -> s.toUpperCase()).collect(Collectors.*toList*());

// Print the transformed ArrayList

System.***out***.println(l2); // Output: [RVK, RK, RKV, RVKI, RVKIR]

}

}

#### II. Processing by count() method

This method returns number of elements present in the stream.  
 *public long count()*

Ex:

long count = l.stream().filter(s ->s.length()==5).count();

sop("The number of 5 length strings is:"+count);

**import** java.util.\*;

**public** **class** Test {

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

// Create an ArrayList of Strings

ArrayList<String> l = **new** ArrayList<>();

l.add("apple");

l.add("orange");

l.add("banana");

l.add("grape");

l.add("pear");

l.add("kiwi");

// Count the number of strings with length 5 using streams and filter

**long** count = l.stream().filter(s -> s.length() == 5).count();

// Print the count

System.***out***.println("The number of 5 length strings is: " + count); // Output: The number of 5 length strings is:

// 2

}

}

#### III. Processing by sorted() method

* If we sort the elements present inside stream then we should go for sorted() method.
* the sorting can either default natural sorting order or customized sorting order specified by comparator.

sorted()- default natural sorting order

sorted(Comparator c)-customized sorting order.

Ex:

List<String> l3=l.stream().sorted().collect(Collectors.toList());

sop("according to default natural sorting order:"+l3);

List<String>l4=l.stream().sorted((s1,s2)->- s1.compareTo(s2)).collect(Collectors.toList());

sop("according to customized sorting order:"+l4);

**ex**

**------**

**import** java.util.\*;

**import** java.util.stream.\*;

**public** **class** Test {

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

// Create an ArrayList of Strings

ArrayList<String> l = **new** ArrayList<>();

l.add("banana");

l.add("apple");

l.add("orange");

l.add("grape");

l.add("pear");

// Sorting in natural order using default natural sorting order of String

List<String> l3 = l.stream().sorted().collect(Collectors.*toList*());

System.***out***.println("According to default natural sorting order: " + l3);

// Output: According to default natural sorting order: [apple, banana, grape,

// orange, pear]

// Sorting in customized order (reverse alphabetical order)

List<String> l4 = l.stream().sorted((s1, s2) -> -s1.compareTo(s2)).collect(Collectors.*toList*());

System.***out***.println("According to customized sorting order: " + l4);

// Output: According to customized sorting order: [pear, orange, grape, banana,

// apple]

}

}

#### IV. Processing by min() and max() methods

min(Comparator c)

returns minimum value according to specified comparator.

max(Comparator c)

returns maximum value according to specified comparator

Ex:

String min=l.stream().min((s1,s2) -> s1.compareTo(s2)).get();

sop("minimum value is:"+min);

String max=l.stream().max((s1,s2) -> s1.compareTo(s2)).get();

sop("maximum value is:"+max);

**ex**

**------**

**import** java.util.\*;

**public** **class** MinMaxExample {

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

// Sample list of strings

List<String> l = Arrays.*asList*("apple", "orange", "banana", "grape", "pear");

// Finding minimum value using Stream min() with Comparator

String min = l.stream().min((s1, s2) -> s1.compareTo(s2)).get();

System.***out***.println("Minimum value is: " + min);

// Finding maximum value using Stream max() with Comparator

String max = l.stream().max((s1, s2) -> s1.compareTo(s2)).get();

System.***out***.println("Maximum value is: " + max);

}

}

#### 

#### V. forEach() method

* This method will not return anything.
* This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

Ex:

l.stream().forEach(s->sop(s));

l3.stream().forEach(System.out:: println);

Ex:

1) import java.util.\*;

2) import java.util.stream.\*;

3) class Test1 {

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

5) ArrayList<Integer> l1 = new ArrayaList<Integer>();

6) l1.add(0);

l1.add(15);

l1.add(10);

l1.add(5);

l1.add(30);

l1.add(25);

l1.add(20);

7) System.out.println(l1);

8) ArrayList<Integer> l2=l1.stream().map(i-> i+10).collect(Collectors.toList());

9) System.out.println(l2);

10) long count = l1.stream().filter(i->i%2==0).count();

11) System.out.println(count);

12) List<Integer> l3=l1.stream().sorted().collect(Collectors.toList());

13) System.out.println(l3);

14) Comparator<Integer> comp=(i1,i2)->i1.compareTo(i2);

15) List<Integer> l4=l1.stream().sorted(comp).collect(Collectors.toList());

16) System.out.println(l4);

17) Integer min=l1.stream().min(comp).get();

18) System.out.println(min);

19) Integer max=l1.stream().max(comp).get();

20) System.out.println(max);

21) l3.stream().forEach(i->sop(i));

22) l3.stream().forEach(System.out:: println);

23)

24) }

25) }

**Ex**

**-----**

**import** java.util.\*;

**import** java.util.stream.\*;

**public** **class** Test1 {

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

// Create an ArrayList of Integers

ArrayList<Integer> l1 = **new** ArrayList<>();

l1.add(0);

l1.add(15);

l1.add(10);

l1.add(5);

l1.add(30);

l1.add(25);

l1.add(20);

// Print the original ArrayList

System.***out***.println("Original List: " + l1);

// Using streams to map each element by adding 10

ArrayList<Integer> l2 = l1.stream().map(i -> i + 10).collect(Collectors.*toCollection*(ArrayList::**new**));

System.***out***.println("Mapped List (adding 10 to each element): " + l2);

// Counting even numbers in the list

**long** count = l1.stream().filter(i -> i % 2 == 0).count();

System.***out***.println("Count of even numbers: " + count);

// Sorting the list in natural order

List<Integer> l3 = l1.stream().sorted().collect(Collectors.*toList*());

System.***out***.println("Sorted List (natural order): " + l3);

// Sorting the list using a comparator

Comparator<Integer> comp = (i1, i2) -> i1.compareTo(i2);

List<Integer> l4 = l1.stream().sorted(comp).collect(Collectors.*toList*());

System.***out***.println("Sorted List (using comparator): " + l4);

// Finding the minimum value in the list

Integer min = l1.stream().min(comp).get();

System.***out***.println("Minimum value: " + min);

// Finding the maximum value in the list

Integer max = l1.stream().max(comp).get();

System.***out***.println("Maximum value: " + max);

// Iterating over the elements and printing them using forEach

System.***out***.println("Iterating over sorted list:");

l3.stream().forEach(i -> System.***out***.print(i + " "));

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

// Using method reference to print elements

System.***out***.println("Printing elements using method reference:");

l3.stream().forEach(System.***out***::println);

}

}

#### 

#### VI. toArray() method

We can use toArray() method to copy elements present in the stream into specified array

Integer[] ir = l1.stream().toArray(Integer[] :: new);

for(Integer i: ir) {

sop(i);

}

**Ex**

**---**

**import** java.util.\*;

**import** java.util.stream.\*;

**public** **class** Test1 {

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

// Create an ArrayList of Integers

ArrayList<Integer> l1 = **new** ArrayList<>();

l1.add(0);

l1.add(15);

l1.add(10);

l1.add(5);

l1.add(30);

l1.add(25);

l1.add(20);

// Convert ArrayList to Integer array using streams

Integer[] ir = l1.stream().toArray(Integer[]::**new**);

// Iterate over the Integer array and print each element

**for** (Integer i : ir) {

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

}

}

}

#### VII. Stream.of()method

We can also apply a stream for group of values and for arrays.

Ex:

Stream s=Stream.of(99,999,9999,99999);

s.forEach(System.out:: println);

Double[] d={10.0,10.1,10.2,10.3};

Stream s1=Stream.of(d);

s1.forEach(System.out :: println);

**import** java.util.stream.Stream;

**public** **class** StreamExample {

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

// Creating a Stream of integers

Stream<Integer> s = Stream.*of*(99, 999, 9999, 99999);

// Printing each element of the Stream

s.forEach(System.***out***::println);

}

}

## **Date and Time API: (Joda-Time API)**

* Until Java 1.7version the classes present in java.util package to handle Date and Time (like Date, Calendar, TimeZoneetc) are not up to the mark with respect to convenience and performance.
* To overcome this problem in the 1.8version oracle people introduced Joda-Time API. This API developed by joda.org and available in Java in the form of java.time package.

# program for to display System Date and time

**import** java.time.\*;

**public** **class** DateTime {

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

// Getting the current date

LocalDate date = LocalDate.*now*();

System.***out***.println("Current Date: " + date);

// Getting the current time

LocalTime time = LocalTime.*now*();

System.***out***.println("Current Time: " + time);

}

}

Once we get LocalDate object we can call the following methods on that object to retrieve Day,month and year values separately.

Ex:

**import** java.time.\*;

**public** **class** Test {

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

// Getting the current date

LocalDate date = LocalDate.*now*();

System.***out***.println("Current Date: " + date);

// Getting day, month, and year components separately

**int** dd = date.getDayOfMonth(); // Day of the month (1-31)

**int** mm = date.getMonthValue(); // Month (1-12)

**int** yy = date.getYear(); // Year

// Printing day, month, and year separately

System.***out***.println("Day: " + dd);

System.***out***.println("Month: " + mm);

System.***out***.println("Year: " + yy);

// Printing formatted date

System.***out***.printf("\nFormatted Date: %d-%d-%d", dd, mm, yy);

}

}

o/p

------

Current Date: 2024-07-19

Day: 19

Month: 7

Year: 2024

Formatted Date: 19-7-2024

Once we get LocalTime object we can call the following methods on that object.

**import** java.time.LocalTime;

**public** **class** Test {

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

// Get the current time

LocalTime time = LocalTime.*now*();

// Extract individual components: hours, minutes, seconds, nanoseconds

**int** h = time.getHour(); // Hour of the day (0-23)

**int** m = time.getMinute(); // Minute of the hour (0-59)

**int** s = time.getSecond(); // Second of the minute (0-59)

**int** n = time.getNano(); // Nanosecond of the second (0-999,999,999)

// Print the time components using formatted output

System.***out***.printf("%d:%d:%d:%d\n", h, m, s, n);

}

}

o/p

-----

17:16:2:503436500

If we want to represent both Date and Time then we should go for LocalDateTime object.

LocalDateTime dt = LocalDateTime.now();

System.out.println(dt);

O/p: 2024-1-4T12:57:24.531

**Ex:**

**-----**

**import** java.time.LocalTime;

**import** java.time.LocalDateTime;

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

**public** **class** DateTimeExample {

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

// Get current date and time

LocalDateTime dt = LocalDateTime.*now*();

// Print the current date and time using default format

System.***out***.println("Current Date and Time: " + dt);

// Format the LocalDateTime using DateTimeFormatter

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

String formattedDateTime = dt.format(formatter);

System.***out***.println("Formatted Date and Time: " + formattedDateTime);

}

}

o/p:

------

Current Date and Time: 2024-07-19T17:18:36.383445300

Formatted Date and Time: 2024-07-19 17:18:36

We can represent a particular Date and Time by using LocalDateTime object as follows.

Ex:

LocalDateTime dt1 = LocalDateTime.of(1995,Month.APRIL,28,12,45);

sop(dt1);

Ex:

LocalDateTime dt1=LocalDateTime.of(1995,04,28,12,45);

Sop(dt1);

Sop("After six months:"+dt.plusMonths(6));

Sop("Before six months:"+dt.minusMonths(6));

**Ex:**

**-----**

**import** java.time.LocalDateTime;

**import** java.time.Month;

**public** **class** LocalDateTimeExample {

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

// Example 1: Creating LocalDateTime with Month enumeration

LocalDateTime dt1 = LocalDateTime.*of*(1995, Month.***APRIL***, 28, 12, 45);

System.***out***.println("Example 1:");

System.***out***.println(dt1); // Output: 1995-04-28T12:45

// Example 2: Creating LocalDateTime with numeric month

LocalDateTime dt2 = LocalDateTime.*of*(1995, 4, 28, 12, 45);

System.***out***.println("\nExample 2:");

System.***out***.println(dt2); // Output: 1995-04-28T12:45

// Adding and subtracting months from LocalDateTime

System.***out***.println("\nAfter six months from dt2:");

System.***out***.println(dt2.plusMonths(6)); // Output: 1995-10-28T12:45

System.***out***.println("\nBefore six months from dt2:");

System.***out***.println(dt2.minusMonths(6)); // Output: 1994-10-28T12:45

}

}

o/p

------

xample 1:

1995-04-28T12:45

Example 2:

1995-04-28T12:45

After six months from dt2:

1995-10-28T12:45

Before six months from dt2:

1994-10-28T12:45

To Represent Zone: ZoneId object can be used to represent Zone.

Ex:

1) import java.time.\*;

2) class ProgramOne {

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

4) ZoneId zone = ZoneId.systemDefault();

5) System.out.println(zone);

6) }

7) }

**Ex:**

**-----**

**import** java.time.\*;

**public** **class** ProgramOne {

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

// Get the system default time zone

ZoneId zone = ZoneId.*systemDefault*();

// Print the time zone identifier

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

}

}

o/p:

-------

Asia/Calcutta

We can create ZoneId for a particular zone as follows

Ex:

ZoneId la = ZoneId.of("America/Los\_Angeles");

ZonedDateTime zt = ZonedDateTime.now(la);

System.out.println(zt);

**Ex:**

**----**

**import** java.time.\*;

**public** **class** ZonedDateTimeExample {

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

// Creating a ZoneId for Los Angeles

ZoneId la = ZoneId.*of*("America/Los\_Angeles");

// Getting the current date and time in Los Angeles time zone

ZonedDateTime zt = ZonedDateTime.*now*(la);

// Printing the current date and time in Los Angeles time zone

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

}

}

o/p:

--------

2024-07-19T04:54:26.018242400-07:00[America/Los\_Angeles]

Period Object: Period object can be used to represent quantity of time

Ex:

LocalDate today = LocalDate.now();

LocalDate birthday = LocalDate.of(1989,06,15);

Period p = Period.between(birthday,today);

System.out.printf("age is %d year %d months %d days",p.getYears(),p.getMonths(),p.getDays());

**Ex:**

**------**

**import** java.time.\*;

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

**public** **class** AgeCalculator {

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

// Get the current date

LocalDate today = LocalDate.*now*();

// Define the birthdate

LocalDate birthday = LocalDate.*of*(1989, 6, 15);

// Calculate the period between birthday and today

Period p = Period.*between*(birthday, today);

// Print the calculated age

System.***out***.printf("Age is %d years, %d months, and %d days.%n", p.getYears(), p.getMonths(), p.getDays());

}

}

o/p:

------

Age is 35 years, 1 months, and 4 days.