Try with Resources: ARM:

package com.jdk8;

import java.io.\*;

public class TDemo {

public static void main(String[] args) {

File f=new File("text.java");

try(FileReader fr=new FileReader(f))

{

System.out.println(fr);

}

catch(Exception e)

{

System.out.println(e);

}

// System.out.println(fr);

}

}

Functional interface :

package com.jdk8;

import org.omg.Messaging.SyncScopeHelper;

interface calc

{

void add();

}

@FunctionalInterface

interface shape

{

double area();

//double volumn();

}

public class FDemo {

public static void main(String[] args) {

shape circle=new shape() {

@Override

public double area() {

// TODO Auto-generated method stub

System.out.println(volumn());

return 2.3;

}

//@Override

public double volumn() {

// TODO Auto-generated method stub

return 0;

} };

System.out.println(circle.area());

}

}

Static and default method in interfaces:

package com.jdk8;

public interface Employee {

void addEmp();

void delEmp();

String cmp="Atos";

static void setCmp(String x)

{

String cmp=x;

System.out.println(cmp);

}

default String setLoc(String x)

{

String loc=x;

return loc;

}

}

Static and non static

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.

package com.jdk8;

@FunctionalInterface

interface shape1

{

double area();

}

public class Lamda1 {

public static void main(String[] args) {

shape1 c=()->

{

return 3.14\*2.2\*2.2;

};

System.out.println(c.area());

}

}

package com.jdk8;

@FunctionalInterface

interface shape1

{

double area(double x ,double y);

}

public class Lamda1 {

public static void main(String[] args) {

shape1 c=(a,b)->

{

return a\*b\*b;

};

System.out.println(c.area(3.14,2.2));

}

}

No Parameter Syntax

1. () -> {
2. //Body of no parameter lambda
3. }

One Parameter Syntax

1. (p1) -> {
2. //Body of single parameter lambda
3. }

Two Parameter Syntax

1. (p1,p2) -> {
2. //Body of multiple parameter lambda
3. }

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

1. interface Drawable{
2. public void draw();
3. }
4. public class LambdaExpressionExample {
5. public static void main(String[] args) {
6. int width=10;
8. //without lambda, Drawable implementation using anonymous class
9. Drawable d=new Drawable(){
10. public void draw(){System.out.println("Drawing "+width);}
11. };
12. d.draw();
13. }
14. }

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

Output:

Drawing 10

Java Lambda Expression Example

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

1. @FunctionalInterface  //It is optional
2. interface Drawable{
3. public void draw();
4. }
6. public class LambdaExpressionExample2 {
7. public static void main(String[] args) {
8. int width=10;
10. //with lambda
11. Drawable d2=()->{
12. System.out.println("Drawing "+width);
13. };
14. d2.draw();
15. }
16. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample2)

Output:

Drawing 10

A lambda expression can have zero or any number of arguments. Let's see the examples:

Java Lambda Expression Example: No Parameter

1. interface Sayable{
2. public String say();
3. }
4. public class LambdaExpressionExample3{
5. public static void main(String[] args) {
6. Sayable s=()->{
7. return "I have nothing to say.";
8. };
9. System.out.println(s.say());
10. }
11. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample3)

Output:

I have nothing to say.

Java Lambda Expression Example: Single Parameter

1. interface Sayable{
2. public String say(String name);
3. }
5. public class LambdaExpressionExample4{
6. public static void main(String[] args) {
8. // Lambda expression with single parameter.
9. Sayable s1=(name)->{
10. return "Hello, "+name;
11. };
12. System.out.println(s1.say("Sonoo"));
14. // You can omit function parentheses
15. Sayable s2= name ->{
16. return "Hello, "+name;
17. };
18. System.out.println(s2.say("Sonoo"));
19. }
20. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample4)

Output:

Hello, Sonoo

Hello, Sonoo

Java Lambda Expression Example: Multiple Parameters

1. interface Addable{
2. int add(int a,int b);
3. }
5. public class LambdaExpressionExample5{
6. public static void main(String[] args) {
8. // Multiple parameters in lambda expression
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Multiple parameters with data type in lambda expression
13. Addable ad2=(int a,int b)->(a+b);
14. System.out.println(ad2.add(100,200));
15. }
16. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample5)

Output:

30

300

Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

1. interface Addable{
2. int add(int a,int b);
3. }
5. public class LambdaExpressionExample6 {
6. public static void main(String[] args) {
8. // Lambda expression without return keyword.
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Lambda expression with return keyword.
13. Addable ad2=(int a,int b)->{
14. return (a+b);
15. };
16. System.out.println(ad2.add(100,200));
17. }
18. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample6)

Output:

30

300

Java Lambda Expression Example: Foreach Loop

1. import java.util.\*;
2. public class LambdaExpressionExample7{
3. public static void main(String[] args) {
5. List<String> list=new ArrayList<String>();
6. list.add("ankit");
7. list.add("mayank");
8. list.add("irfan");
9. list.add("jai");
11. list.forEach(
12. (n)->System.out.println(n)
13. );
14. }
15. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample7)

Output:

ankit

mayank

irfan

jai

Java Lambda Expression Example: Multiple Statements

1. @FunctionalInterface
2. interface Sayable{
3. String say(String message);
4. }
6. public class LambdaExpressionExample8{
7. public static void main(String[] args) {
9. // You can pass multiple statements in lambda expression
10. Sayable person = (message)-> {
11. String str1 = "I would like to say, ";
12. String str2 = str1 + message;
13. return str2;
14. };
15. System.out.println(person.say("time is precious."));
16. }
17. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample8)

Output:

I would like to say, time is precious.

Java Lambda Expression Example: Creating Thread

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

1. public class LambdaExpressionExample9{
2. public static void main(String[] args) {
4. //Thread Example without lambda
5. Runnable r1=new Runnable(){
6. public void run(){
7. System.out.println("Thread1 is running...");
8. }
9. };
10. Thread t1=new Thread(r1);
11. t1.start();
12. //Thread Example with lambda
13. Runnable r2=()->{
14. System.out.println("Thread2 is running...");
15. };
16. Thread t2=new Thread(r2);
17. t2.start();
18. }
19. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample9)

Output:

Thread1 is running...

Thread2 is running...

Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data. Following are some lambda and collection examples provided.

Java Lambda Expression Example: Comparator

1. import java.util.ArrayList;
2. import java.util.Collections;
3. import java.util.List;
4. class Product{
5. int id;
6. String name;
7. float price;
8. public Product(int id, String name, float price) {
9. super();
10. this.id = id;
11. this.name = name;
12. this.price = price;
13. }
14. }
15. public class LambdaExpressionExample10{
16. public static void main(String[] args) {
17. List<Product> list=new ArrayList<Product>();
19. //Adding Products
20. list.add(new Product(1,"HP Laptop",25000f));
21. list.add(new Product(3,"Keyboard",300f));
22. list.add(new Product(2,"Dell Mouse",150f));
24. System.out.println("Sorting on the basis of name...");
26. // implementing lambda expression
27. Collections.sort(list,(p1,p2)->{
28. return p1.name.compareTo(p2.name);
29. });
30. for(Product p:list){
31. System.out.println(p.id+" "+p.name+" "+p.price);
32. }
34. }
35. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample10)

Output:

Sorting on the basis of name...

2 Dell Mouse 150.0

1 HP Laptop 25000.0

3 Keyboard 300.0

Java Lambda Expression Example: Filter Collection Data

1. import java.util.ArrayList;
2. import java.util.List;
3. import java.util.stream.Stream;
4. class Product{
5. int id;
6. String name;
7. float price;
8. public Product(int id, String name, float price) {
9. super();
10. this.id = id;
11. this.name = name;
12. this.price = price;
13. }
14. }
15. public class LambdaExpressionExample11{
16. public static void main(String[] args) {
17. List<Product> list=new ArrayList<Product>();
18. list.add(new Product(1,"Samsung A5",17000f));
19. list.add(new Product(3,"Iphone 6S",65000f));
20. list.add(new Product(2,"Sony Xperia",25000f));
21. list.add(new Product(4,"Nokia Lumia",15000f));
22. list.add(new Product(5,"Redmi4 ",26000f));
23. list.add(new Product(6,"Lenevo Vibe",19000f));
25. // using lambda to filter data
26. Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);
28. // using lambda to iterate through collection
29. filtered\_data.forEach(
30. product -> System.out.println(product.name+": "+product.price)
31. );
32. }
33. }

[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample11)

Output:

Iphone 6S: 65000.0

Sony Xperia: 25000.0

Redmi4 : 26000.0

Java Lambda Expression Example: Event Listener

1. import javax.swing.JButton;
2. import javax.swing.JFrame;
3. import javax.swing.JTextField;
4. public class LambdaEventListenerExample {
5. public static void main(String[] args) {
6. JTextField tf=new JTextField();
7. tf.setBounds(50, 50,150,20);
8. JButton b=new JButton("click");
9. b.setBounds(80,100,70,30);
11. // lambda expression implementing here.
12. b.addActionListener(e-> {tf.setText("hello swing");});
14. JFrame f=new JFrame();
15. f.add(tf);f.add(b);
16. f.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
17. f.setLayout(null);
18. f.setSize(300, 200);
19. f.setVisible(true);
21. }
23. }

Method Reference:

# Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

## Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.
4. interface Sayable{
5. void say();
6. }
7. public class MethodReference {
8. public static void saySomething(){
9. System.out.println("Hello, this is static method.");
10. }
11. public static void main(String[] args) {
12. // Referring static method
13. Sayable sayable = MethodReference::saySomething;
14. // Calling interface method
15. sayable.say();
16. }
17. }

### Example 2

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

1. public class MethodReference2 {
2. public static void ThreadStatus(){
3. System.out.println("Thread is running...");
4. }
5. public static void main(String[] args) {
6. Thread t2=new Thread(MethodReference2::ThreadStatus);
7. t2.start();
8. }
9. }

You can also override static methods by referring methods. In the following example, we have defined and overloaded three add methods.

1. import java.util.function.BiFunction;
2. class Arithmetic{
3. public static int add(int a, int b){
4. return a+b;
5. }
6. public static float add(int a, float b){
7. return a+b;
8. }
9. public static float add(float a, float b){
10. return a+b;
11. }
12. }
13. public class MethodReference4 {
14. public static void main(String[] args) {
15. BiFunction<Integer, Integer, Integer>adder1 = Arithmetic::add;
16. BiFunction<Integer, Float, Float>adder2 = Arithmetic::add;
17. BiFunction<Float, Float, Float>adder3 = Arithmetic::add;
18. int result1 = adder1.apply(10, 20);
19. float result2 = adder2.apply(10, 20.0f);
20. float result3 = adder3.apply(10.0f, 20.0f);
21. System.out.println(result1);
22. System.out.println(result2);
23. System.out.println(result3);
24. }
25. }

## Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

1. containingObject::instanceMethodName

### Example 1

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

1. interface Sayable{
2. void say();
3. }
4. public class InstanceMethodReference {
5. public void saySomething(){
6. System.out.println("Hello, this is non-static method.");
7. }
8. public static void main(String[] args) {
9. InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating object
10. // Referring non-static method using reference
11. Sayable sayable = methodReference::saySomething;
12. // Calling interface method
13. sayable.say();
14. // Referring non-static method using anonymous object
15. Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use anonymous object also
16. // Calling interface method
17. sayable2.say();
18. }
19. }

### Example 2

In the following example, we are referring instance (non-static) method. Runnable interface contains only one abstract method. So, we can use it as functional interface.

1. public class InstanceMethodReference2 {
2. public void printnMsg(){
3. System.out.println("Hello, this is instance method");
4. }
5. public static void main(String[] args) {
6. Thread t2=new Thread(new InstanceMethodReference2()::printnMsg);
7. t2.start();
8. }
9. }

### Example 3

In the following example, we are using BiFunction interface. It is a predefined interface and contains a functional method apply(). Here, we are referring add method to apply method.

1. import java.util.function.BiFunction;
2. class Arithmetic{
3. public int add(int a, int b){
4. return a+b;
5. }
6. }
7. public class InstanceMethodReference3 {
8. public static void main(String[] args) {
9. BiFunction<Integer, Integer, Integer>adder = new Arithmetic()::add;
10. int result = adder.apply(10, 20);
11. System.out.println(result);
12. }
13. }

## 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

1. ClassName::new

### Example

1. interface Messageable{
2. Message getMessage(String msg);
3. }
4. class Message{
5. Message(String msg){
6. System.out.print(msg);
7. }
8. }
9. public class ConstructorReference {
10. public static void main(String[] args) {
11. Messageable hello = Message::new;
12. hello.getMessage("Hello");
13. }
14. }

# 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

1. @FunctionalInterface
2. interface sayable{
3. void say(String msg);
4. }
5. public class FunctionalInterfaceExample implements sayable{
6. public void say(String msg){
7. System.out.println(msg);
8. }
9. public static void main(String[] args) {
10. FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
11. fie.say("Hello there");
12. }
13. }

### Invalid Functional Interface

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

1. interface sayable{
2. void say(String msg);   // abstract method
3. }
4. @FunctionalInterface
5. interface Doable extends sayable{
6. // Invalid '@FunctionalInterface' annotation; Doable is not a functional interface
7. void doIt();
8. }

# Java Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

### Java Default Method Example

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Let's see a simple

1. interface Sayable{
2. // Default method
3. default void say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. void sayMore(String msg);
8. }
9. public class DefaultMethods implements Sayable{
10. public void sayMore(String msg){        // implementing abstract method
11. System.out.println(msg);
12. }
13. public static void main(String[] args) {
14. DefaultMethods dm = new DefaultMethods();
15. dm.say();   // calling default method
16. dm.sayMore("Work is worship");  // calling abstract method
18. }
19. }

Output:

Hello, this is default method

Work is worship

## Static Methods inside Java 8 Interface

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

1. interface Sayable{
2. // default method
3. default void say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. void sayMore(String msg);
8. // static method
9. static void sayLouder(String msg){
10. System.out.println(msg);
11. }
12. }
13. public class DefaultMethods implements Sayable{
14. public void sayMore(String msg){     // implementing abstract method
15. System.out.println(msg);
16. }
17. public static void main(String[] args) {
18. DefaultMethods dm = new DefaultMethods();
19. dm.say();                       // calling default method
20. dm.sayMore("Work is worship");      // calling abstract method
21. Sayable.sayLouder("Helloooo...");   // calling static method
22. }
23. }

Output:

Hello there

Work is worship

Helloooo...

|  |
| --- |
| Abstract Class vs Java 8 Interface After having default and static methods inside the interface, we think about the need of abstract class in Java. An interface and an abstract class is almost similar except that you can create constructor in the abstract class whereas you can't do this in interface. |

For Each Method

# Java forEach loop

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

## forEach() Signature in Iterable Interface

1. default void forEach(Consumer<super T>action)

### Java 8 forEach() example 1

1. import java.util.ArrayList;
2. import java.util.List;
3. public class ForEachExample {
4. public static void main(String[] args) {
5. List<String> gamesList = new ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing lambda expression--------------");
11. gamesList.forEach(games -> System.out.println(games));
13. }
14. }

Collectors class:

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.

package com.jdk8.Collector;

import java.util.\*;

import java.util.stream.Collector;

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 CDemo1 {

public static void main(String ar[])

{

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);

Double sumPrices =

productsList.stream()

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

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

Double avg =

productsList.stream().collect(Collectors.averagingDouble(x->x.price));

System.out.println("Avg of prices: "+avg);

Long noOfElements = productsList.stream()

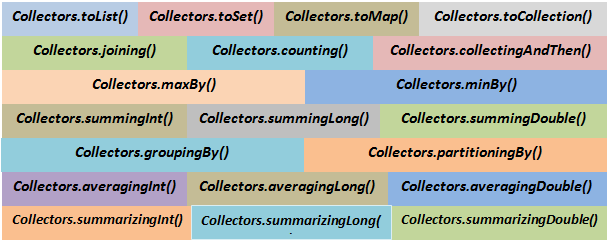
.collect(Collectors.counting());

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

}

}

<https://javaconceptoftheday.com/java-8-collectors-tutorial/>



String Joiner:

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.

**package** com.jdk8.StringJoiner;

**import** java.util.StringJoiner;

**public** **class** Demo1 {

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

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

// Prints nothing because it is empty

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

// We can set default empty value.

joinNames.setEmptyValue("It is empty");

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

// Adding values to StringJoiner

joinNames.add("Rahul");

joinNames.add("Raju");

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

// Returns length of StringJoiner

**int** length = joinNames.length();

System.***out***.println("Length: "+length);

// Returns StringJoiner as String type

String str = joinNames.toString();

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

// Now, we can apply String methods on it

**char** ch = str.charAt(3);

System.***out***.println("Character at index 3: "+ch);

// Adding one more element

joinNames.add("Sorabh");

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

// Returns length

**int** newLength = joinNames.length();

System.***out***.println("New Length: "+newLength);

}

}

Optional Class:

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

Java NIO package:

Date Time API:

## New Date Time API (Java 8 Onwards)

The new date api tries to fix the above problems with legacy classes. It contains mainly the following classes:

* [java.time.LocalDate](https://howtodoinjava.com/java/date-time/compare-localdates/) : represents a year-month-day in the ISO calendar and is useful for representing**a date without a time**. It can be used to represent a date only information such as a birth date or wedding date.
* [java.time.LocalTime](https://howtodoinjava.com/java/date-time/java-localtime/) : deals in **time only**. It is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library.
* [java.time.LocalDateTime](https://howtodoinjava.com/java/date-time/compare-localdatetime/) : handles **both date and time, without a time zone**. It is a combination of LocalDate with LocalTime.
* [java.time.ZonedDateTime](https://howtodoinjava.com/java/date-time/zoneddatetime-comparison/) : combines the **LocalDateTime class with the zone information** given in ZoneId class. It represent a complete date time stamp along with timezone information.
* java.time.OffsetTime : handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.
* java.time.OffsetDateTime : handles a **date and time with a corresponding time zone offset from Greenwich/UTC**, without a time zone ID.
* java.time.Clock : provides access to the **current instant, date and time in any given time-zone**. Although the use of the Clock class is optional, this feature allows us to test your code for other time zones, or by using a fixed clock, where time does not change.
* java.time.Instant : represents the **start of a nanosecond on the timeline (since EPOCH)** and useful for generating a timestamp to represent machine time. An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.
* java.time.Duration : **Differnce between two instants** and measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes.
* [java.time.Period](https://howtodoinjava.com/java/date-time/java8-period/) : To define the **difference between dates in date-based values** (years, months, days).
* java.time.ZoneId : specifies a **time zone identifier** and provides rules for converting between an Instant and a LocalDateTime.
* java.time.ZoneOffset : specifies a **time zone offset from Greenwich/UTC time**.
* [java.time.format.DateTimeFormatter](https://howtodoinjava.com/java/date-time/java8-datetimeformatter-example/) : provides numerous predefined formatters, or we can define our own. It provides parse() or format() method to **parsing and formatting the date time values**.
* [TemporalAdjusters](https://howtodoinjava.com/java/date-time/java8-temporal-adjusters/): provide many useful **inbuilt adjusters for handling recurring events.**
* [TemporalQuery](https://howtodoinjava.com/java/date-time/temporalquery/): be used as the assignment target for a [lambda expression](https://howtodoinjava.com/java8/lambda-expressions/) or [method reference](https://howtodoinjava.com/java8/lambda-method-references-example/).
* [DayOfWeek](https://howtodoinjava.com/java/date-time/find-dayofweek/): an **enum representing the seven days of the week** – Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday.

Stream API:

# 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.

**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

 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);

// This is more compact approach for filtering data

        productsList.stream()

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

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

 // 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 method of Float class

        System.out.println(totalPrice2);

 // 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);

}

}

**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);

    }

}

# Java Nashorn

Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool jjs which is used to execute JavaScript code.

You can execute JavaScript code by using jjs command-line tool and by embedding into Java source code.

JDK 1.9

Try with resources:

**Example Java 7 Resource Declared within resource block:**

**Resource is declared locally:**

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**public** **class** FinalVariable {

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

**try**(FileOutputStream fileStream=**new** FileOutputStream("javatpoint.txt");){

             String greeting = "Welcome to javaTpoint.";

**byte** b[] = greeting.getBytes();

                fileStream.write(b);

                System.out.println("File written");

        }**catch**(Exception e) {

            System.out.println(e);

        }

    }

}

This code executes fine with Java 7 and even with Java 9 because Java maintains it's legacy.

But the below program would not work with Java 7 because **we can't put resource declared outside the try-with-resource.**

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**public** **class** FinalVariable {

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

        FileOutputStream fileStream=**new** FileOutputStream("student.txt");

**try**(fileStream){

             String greeting = "Welcome to Atos.";

**byte** b[] = greeting.getBytes();

                fileStream.write(b);

                System.out.println("File written");

        }**catch**(Exception e) {

            System.out.println(e);

        }

    }

}

import java.io.FileNotFoundException;

import java.io.\*;

public class FinalVariable {

public static void main(String[] args) throws FileNotFoundException , IOException{

FileOutputStream fileStream=new FileOutputStream("student.txt");

try(fileStream){

String greeting = "Welcome to Atos.";

byte b[] = greeting.getBytes();

fileStream.write(b);

System.out.println("File written");

}

catch(Exception e) {

System.out.println(e);

}

fileStream.write(66);

System.out.println(fileStream);

}

}

In this case, **if we execute the above program using Java 9 compiler, it will execute nicely without any compile error.**

**In both resources are closed implicitly.**

# Java 9 Anonymous Inner Classes Improvement

Java 9 introduced a new feature that allows us to use diamond operator with anonymous classes. Using the diamond with anonymous classes was not allowed in Java 7.

In Java 9, as long as the inferred type is denotable, we can use the diamond operator when we create an anonymous inner class.

Data types that can be written in Java program like int, String etc are called denotable types. Java 9 compiler is enough smart and now can infer type.

### Java 9 Anonymous Inner Classes Example

**abstract** **class** ABCD<T>{

**abstract** T show(T a, T b);

}

**public** **class** TypeInferExample {

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

        ABCD<String> a = **new** ABCD<>() { // diamond operator is empty, compiler infer type

            String show(String a, String b) {

**return** a+b;

            }

        };

        String result = a.show("Java","9");

        System.out.println(result);

    }

}

Output:

Java9

Although we can specifying type in diamond operator explicitly and compiler does not produce any error message. See, the following example, type is specified explicitly.

### Java 9 Anonymous Inner Classes Example

**abstract** **class** ABCD<T>{

**abstract** T show(T a, T b);

}

**public** **class** TypeInferExample {

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

        ABCD<String> a = **new** ABCD<String>() { // diamond operator is not empty

            String show(String a, String b) {

**return** a+b;

            }

        };

        String result = a.show("Java","9");

        System.out.println(result);

    }

}

And we get the same result.

Output:

Java9

What happens? If we compile the following code using Java 8.

### Anonymous Inner Class Example

**abstract** **class** ABCD<T>{

**abstract** T show(T a, T b);

}

**public** **class** TypeInferExample {

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

        ABCD<String> a = **new** ABCD<>() { // diamond operator is empty

            String show(String a, String b) {

**return** a+b;

            }

        };

        String result = a.show("Java","9");

        System.out.println(result);

    }

}

Java 8 compiler throws compile time error because it can't infer type. The error message looks like the below.

Output:

TypeInferExample.java:7: error: cannot infer type arguments for ABCD<T>

ABCD<String> a = new ABCD<>() {

^

reason: cannot use '<>' with anonymous inner classes

where T is a type-variable:

T extends Object declared in class ABCD

1 error

# Java 9 Private Interface Methods

In Java 9, we can create private methods inside an interface. Interface allows us to declare private methods that help to **share** common code between **non-abstract** methods.

Before Java 9, creating private methods inside an interface cause a compile time error. The following example is compiled using Java 8 compiler and throws a compile time error.

**interface** Sayable{

**default** **void** say() {

        saySomething();

    }

    // Private method inside interface

**private** **void** saySomething() {

        System.out.println("Hello... I'm private method");

    }

}

**public** **class** PrivateInterface **implements** Sayable {

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

        Sayable s = **new** PrivateInterface();

        s.say();

    }

}

### Java 9 Private Static Methods Example

**interface** Sayable{

**default** **void** say() {

        saySomething(); // Calling private method

        sayPolitely(); //  Calling private static method

    }

    // Private method inside interface

**private** **void** saySomething() {

        System.out.println("Hello... I'm private method");

    }

    // Private static method inside interface

**private** **static** **void** sayPolitely() {

        System.out.println("I'm private static method");

    }

}

**public** **class** PrivateInterface **implements** Sayable {

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

        Sayable s = **new** PrivateInterface();

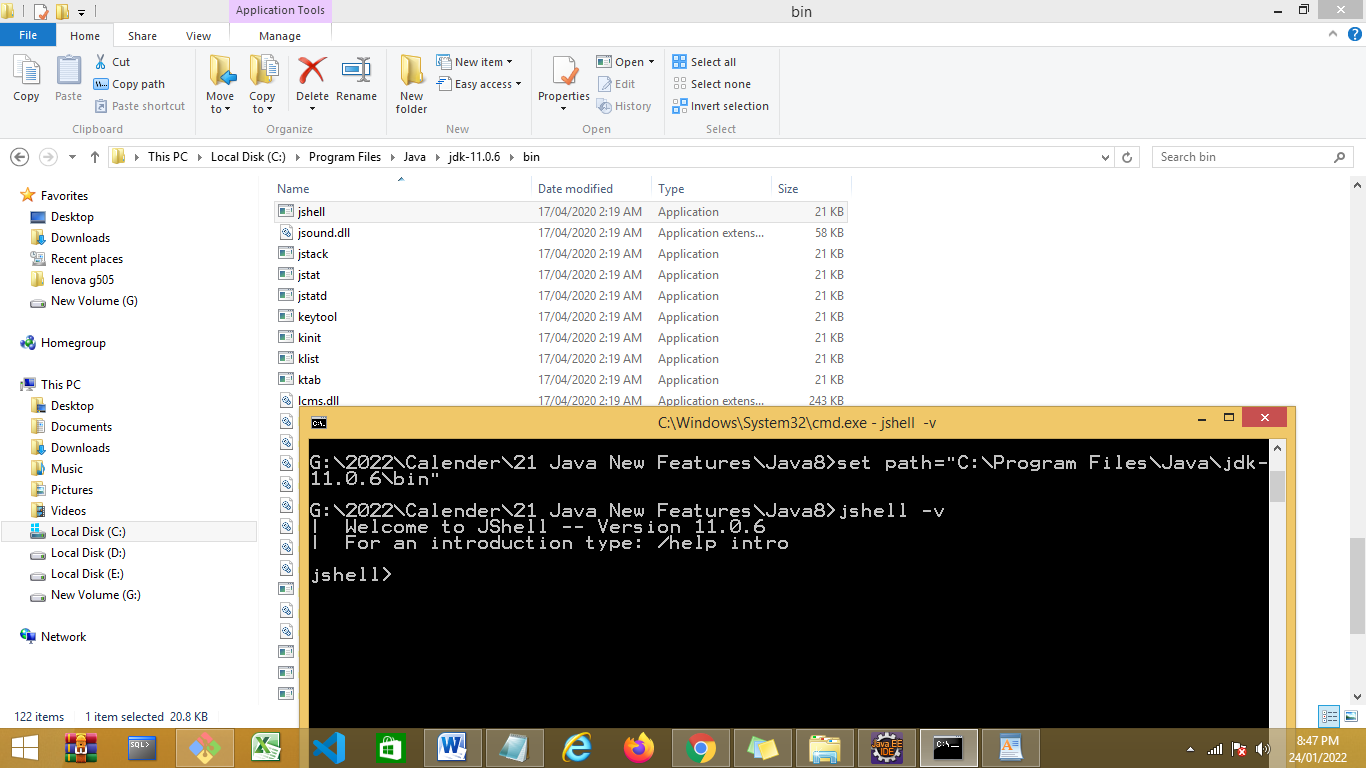
        s.say();

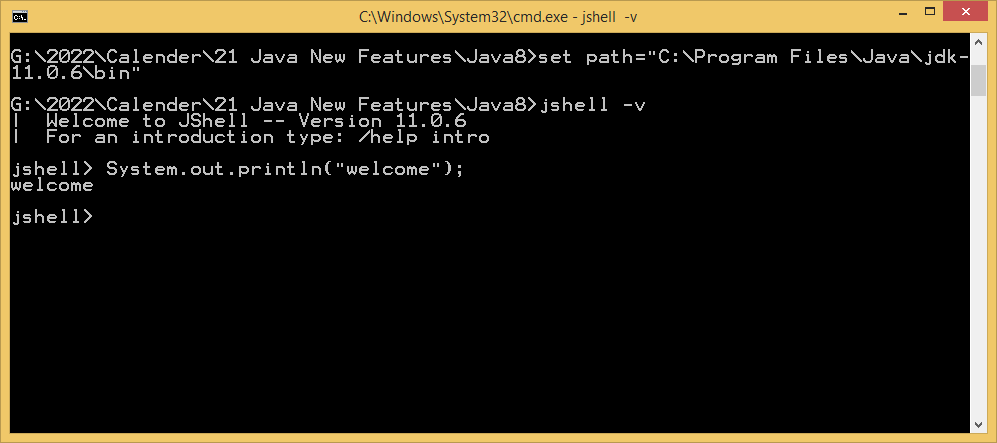
    }

}

# Java Shell Tool (JShell)

It is an interactive Java Shell tool, it allows us to execute Java code from the shell and shows output immediately. JShell is a REPL (Read Evaluate Print Loop) tool and run from the command line.





Module: