**Java New Features**

**Java 7**

**Java <> Operator**

Before Java 7

Map<String, Integer> params = new HashMap<String, Integer>();

After Java 7

Map<String, Integer> params = new HashMap<>();

**try-with-resources**

Before Java 7

public class ResourceManagementBeforeJava7

{

    public static void main(String[] args)

    {

        BufferedReader br = null;

        try

        {

            String sCurrentLine;

            br = new BufferedReader(new FileReader(“C:/temp/test.txt”));

            while ((sCurrentLine = br.readLine()) != null)

            {

                System.out.println(sCurrentLine);

            }

        }

        catch (IOException e)

        {

            e.printStackTrace();

        }

        finally

        {

            try

            {

                if (br != null)

                    br.close();

            }

            catch (IOException ex) {

                ex.printStackTrace();

            }

        }

    }

}

After Java 7

public class ResourceManagementInJava7

{

    public static void main(String[] args)

    {

        try (BufferedReader br = new BufferedReader(new FileReader(“C:/temp/test.txt”)))

        {

            String sCurrentLine;

            while ((sCurrentLine = br.readLine()) != null)

            {

                System.out.println(sCurrentLine);

            }

        }

        catch (IOException e)

        {

            e.printStackTrace();

        }

    }

}

JDBC before Java 7

Connection con = null;

Statement stmt = null;

ResultSet rs = null;

//Registering the Driver

try {

   con = DriverManager.getConnection(mysqlUrl, "root", "password");

   stmt = con.createStatement();

} catch (SQLException e) {

   e.printStackTrace();

}   finally {

   try {

      rs.close();

      stmt.close();

      con.close();

   } catch(SQLException e) {

      e.printStackTrace();

   }

}

JDBC After Java 7

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TryWithResources\_Example {

   public static void main(String args[]) {

      String mysqlUrl = "jdbc:mysql://localhost/mydatabase";

      try(Connection con = DriverManager.getConnection(mysqlUrl, "root", "password");

      Statement stmt = con.createStatement(); ) {

         try(ResultSet rs = stmt.executeQuery("select \* from MyPlayers");){

            while(rs.next()) {

               System.out.print(rs.getInt("ID")+", ");

               System.out.print(rs.getString("First\_Name")+", ");

               System.out.print(rs.getString("Last\_Name")+", ");

               System.out.print(rs.getDate("Date\_Of\_Birth")+", ");

               System.out.print(rs.getString("Place\_Of\_Birth")+", ");

               System.out.print(rs.getString("Country"));

            }

         } catch (SQLException e) {

            e.printStackTrace();

         }

      } catch (SQLException e) {

            e.printStackTrace();

      }

   }

}

**Multi Catch**

Before Java 7

**public** **class** MultipleExceptionExample{

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

**try**{

**int** array[] = newint[10];

            array[10] = 30/0;

        }

**catch**(ArithmeticException e){

            System.out.println(e.getMessage());

        }

**catch**(ArrayIndexOutOfBoundsException e){

            System.out.println(e.getMessage());

        }

**catch**(Exception e){

            System.out.println(e.getMessage());

        }

     }

}

After Java 7

**public** **class** MultipleExceptionExample{

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

**try**{

**int** array[] = newint[10];

            array[10] = 30/0;

        }

**catch**(ArithmeticException | ArrayIndexOutOfBoundsException e){

            System.out.println(e.getMessage());

        }

**catch**(Exception e){

            System.out.println(e.getMessage());

        }

     }

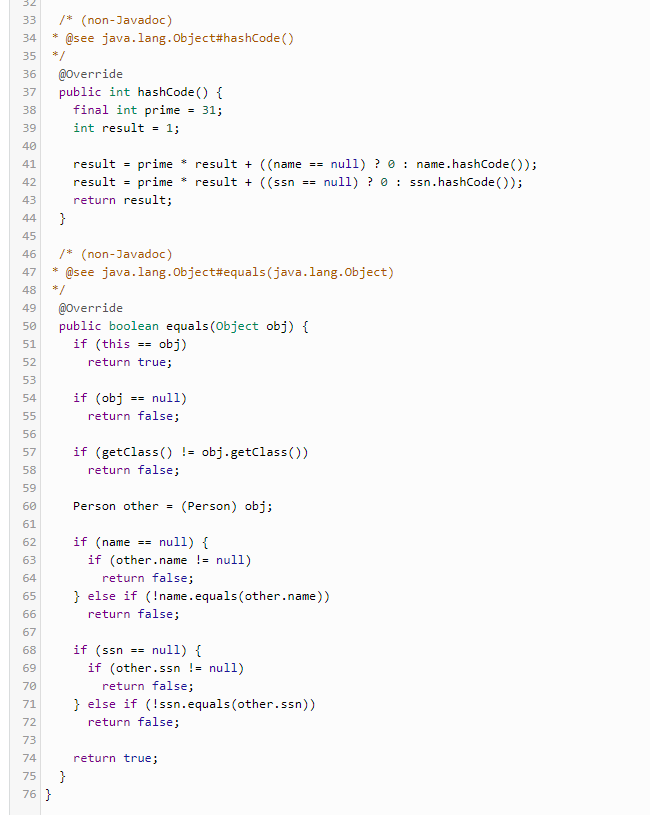
}

**java.util.Objects**

Most applications require an extensive hashing implementation, and writing a good hashing algorithm is always a tedious job. To get rid of this ugly and tedious hashCode and equals implementation, Java introduced an API called java.util.Objects in Java 7, which contains a set of utility methods and is useful in combination with Object instances.

Before Java 6





public class Person {

String ssn;

String name;

public Person(){}

public Person(String ssn, String name){

this.ssn = ssn;

this.name = name;

}

public String getId() {

return ssn;

}

public void setId(String id) {

this.ssn = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

/\* (non-Javadoc)

\* @see java.lang.Object#hashCode()

\*/

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((name == null) ? 0 : name.hashCode());

result = prime \* result + ((ssn == null) ? 0 : ssn.hashCode());

return result;

}

/\* (non-Javadoc)

\* @see java.lang.Object#equals(java.lang.Object)

\*/

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Person other = (Person) obj;

if (name == null) {

if (other.name != null)

return false;

} else if (!name.equals(other.name))

return false;

if (ssn == null) {

if (other.ssn != null)

return false;

} else if (!ssn.equals(other.ssn))

return false;

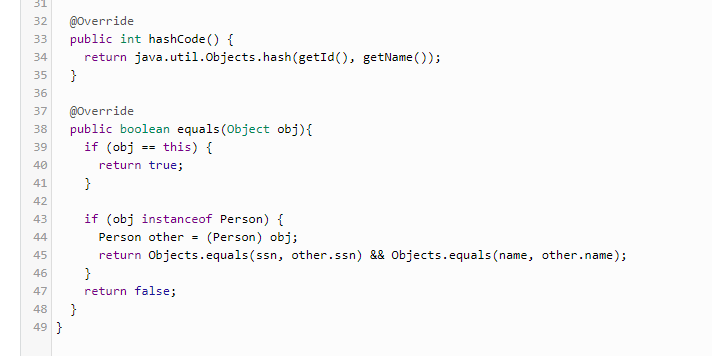
return true;

}

}

After Java 7





import java.util.Objects;

public class Person {

String ssn;

String name;

public Person(){}

public Person(String ssn, String name){

this.ssn = ssn;

this.name = name;

}

public String getId() {

return ssn;

}

public void setId(String id) {

this.ssn = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public int hashCode() {

return java.util.Objects.hash(getId(), getName());

}

@Override

public boolean equals(Object obj){

if (obj == this) {

return true;

}

if (obj instanceof Person) {

Person other = (Person) obj;

return Objects.equals(ssn, other.ssn) && Objects.equals(name, other.name);

}

return false;

}

}

HashingTest.java

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.junit.Test;

import junit.framework.Assert;

public class HashingTest {

public static final Log LOG = LogFactory.getLog(HashingTest.class);

@Test

public void testHashing() {

Person person1 = new Person("xxxx123", "Person1");

Person person2 = new Person("xxxx234", "Person2");

LOG.info("person1 HashCode ==>> " +person1.hashCode());

LOG.info("person2 HashCode ==>> " +person2.hashCode());

Assert.assertFalse(person1.hashCode() == person2.hashCode());

Assert.assertFalse("Both are same employee...", person1.equals(person2));

Person person3 = new Person("xxxx567", "Person3");

Person person4 = new Person("xxxx567", "Person4");

LOG.info("person3 HashCode ==>> " +person3.hashCode());

LOG.info("person4 HashCode ==>> " +person4.hashCode());

Assert.assertEquals(person3.hashCode(), person4.hashCode());

Assert.assertTrue("Both are not same employee...",person3.equals(person4));

}

}

**Java 8**

**Lambda expressions**

Lambda expression is a new feature which is introduced in Java 8. A lambda expression is an **anonymous function**. A function that doesn’t have a name and doesn’t belong to any class. The concept of lambda expression was first introduced in LISP programming language.

**Java Lambda Expression Syntax**

To create a lambda expression, we specify input parameters (if there are any) on the left side of the lambda operator ->, and place the expression or block of statements on the right side of lambda operator.

For example, the lambda expression

(x, y) -> x + y

specifies that lambda expression takes two arguments x and y and returns the sum of these.



**Lambda expression vs method in Java**

A method (or function) in Java has these main parts:

1. Name  
2. Parameter list  
3. Body  
4. return type.

A lambda expression in Java has these main parts:

Lambda expression **only has body and parameter list**.

1. **No name** – function is anonymous so we don’t care about the name  
   2. Parameter list  
   3. Body – This is the main part of the function.  
   4. **No return type** – The java 8 compiler is able to infer the return type by checking the code. you need not to mention it explicitly.

**Where to use the Lambdas in Java**

To use lambda expression, you need to either create your own functional interface or use the pre defined functional interface provided by Java. An interface **with only single abstract method** is called functional interface(or Single Abstract method interface), for example: Runnable, callable, ActionListener etc.

**To use function interface:**

**Pre Java 8:** We create anonymous inner classes.  
**Post Java 8:** You can use lambda expression instead of anonymous inner classes.

**Java Lambda expression Example**

**Without using Lambda expression**: Prior to java 8 we used the anonymous inner class to implement the only abstract method of functional interface.



import java.awt.\*;

import java.awt.event.\*;

public class ButtonListenerOldWay {

public static void main(String[] args) {

Frame frame=new Frame("ActionListener Before Java8");

Button b=new Button("Click Here");

b.setBounds(50,100,80,50);

b.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

System.out.println("Hello World!");

}

});

frame.add(b);

frame.setSize(200,200);

frame.setLayout(null);

frame.setVisible(true);

}

}

**By using Lambda expression**: Instead of creating anonymous inner class, we can create a lambda expression like this:



import java.awt.\*;

public class ButtonListenerNewWay {

public static void main(String[] args) {

Frame frame=new Frame("ActionListener java8");

Button b=new Button("Click Here");

b.setBounds(50,100,80,50);

b.addActionListener(e -> System.out.println("Hello World!"));

frame.add(b);

frame.setSize(200,200);

frame.setLayout(null);

frame.setVisible(true);

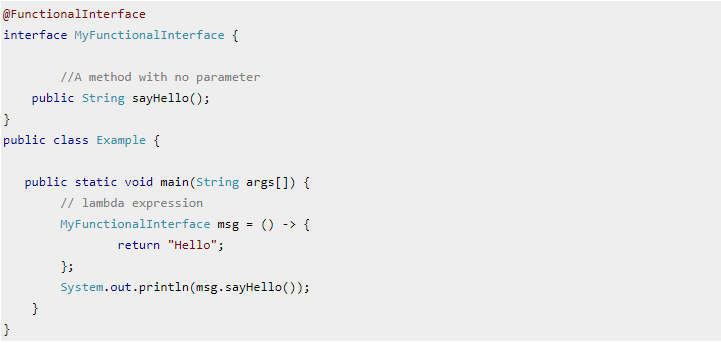
}

}

**Note:** As you can see that we used less code with lambda expression.

Lets see few more examples of Lambda expressions.

**Example 1: Java Lambda Expression with no parameter**



@FunctionalInterface

interface MyFunctionalInterface {

//A method with no parameter

public String sayHello();

}

public class Example {

public static void main(String args[]) {

// lambda expression

MyFunctionalInterface msg = () -> {

return "Hello";

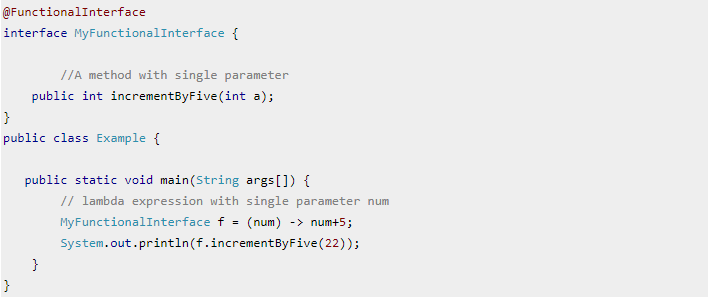
};

System.out.println(msg.sayHello());

}

}

**Example 2: Java Lambda Expression with single parameter**



@FunctionalInterface

interface MyFunctionalInterface {

//A method with single parameter

public int incrementByFive(int a);

}

public class Example {

public static void main(String args[]) {

// lambda expression with single parameter num

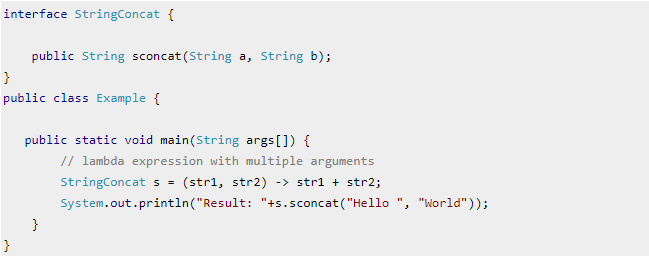
MyFunctionalInterface f = (num) -> num+5;

System.out.println(f.incrementByFive(22));

}

}

**Example 3: Java Lambda Expression with Multiple Parameters**



interface StringConcat {

public String sconcat(String a, String b);

}

public class Example {

public static void main(String args[]) {

// lambda expression with multiple arguments

StringConcat s = (str1, str2) -> str1 + str2;

System.out.println("Result: "+s.sconcat("Hello ", "World"));

}

}

**Example 4: Iterating collections using foreach loop**



import java.util.\*;

public class Example{

public static void main(String[] args) {

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

list.add("Rick");

list.add("Negan");

list.add("Daryl");

list.add("Glenn");

list.add("Carl");

list.forEach(

// lambda expression

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

);

}

}

**Example 5: Iterating Map in Java 8 using Lambda expression**



import java.util.HashMap;

import java.util.Map;

public class IterateMapUsingLambda {

public static void main(String[] args) {

Map<String, Integer> prices = new HashMap<>();

prices.put("Apple", 50);

prices.put("Orange", 20);

prices.put("Banana", 10);

prices.put("Grapes", 40);

prices.put("Papaya", 50);

/\* Iterate without using Lambda

for (Map.Entry<String, Integer> entry : prices.entrySet()) {

System.out.println("Fruit: " + entry.getKey() + ", Price: " + entry.getValue());

}

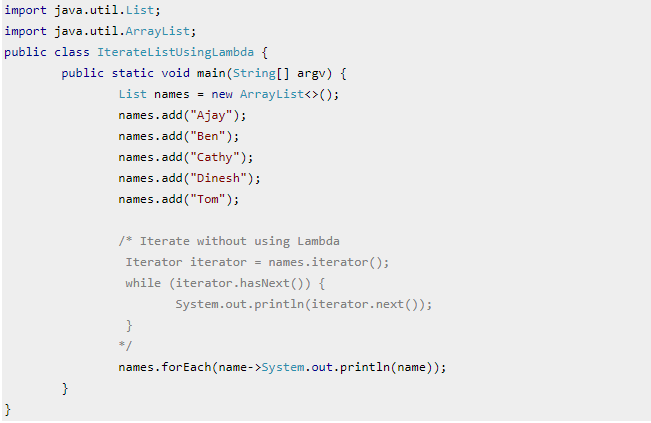
\*/

prices.forEach((k,v)->System.out.println("Fruit: " + k + ", Price: " + v));

}

}

**Example 6: Iterating List in Java 8 using Lambda expression**



import java.util.List;

import java.util.ArrayList;

public class IterateListUsingLambda {

public static void main(String[] argv) {

List names = new ArrayList<>();

names.add("Ajay");

names.add("Ben");

names.add("Cathy");

names.add("Dinesh");

names.add("Tom");

/\* Iterate without using Lambda

Iterator iterator = names.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

\*/

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

}

}

**Method reference**

Method reference is a shorthand notation of a lambda expression to call a method.

For example:

If your lambda expression is like this:



then you can replace it with a method reference like this:



The :: operator is used in method reference to separate the class or object from the method name

**Four types of method references**

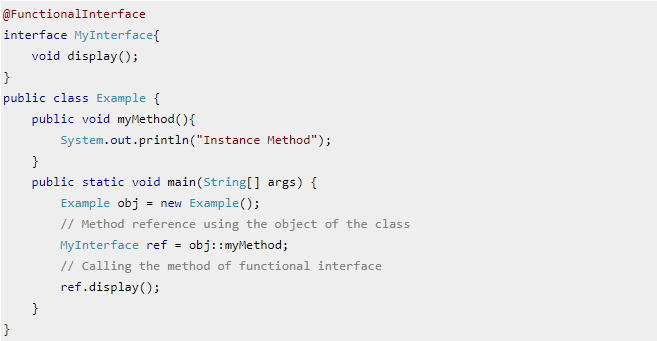
1. Method reference to an instance method of an object – object::instanceMethod

2. Method reference to a static method of a class – Class::staticMethod

3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod

4. Method reference to a constructor – Class::new

1. **Method reference to an instance method of an object**



@FunctionalInterface

interface MyInterface{

void display();

}

public class Example {

public void myMethod(){

System.out.println("Instance Method");

}

public static void main(String[] args) {

Example obj = new Example();

// Method reference using the object of the class

MyInterface ref = obj::myMethod;

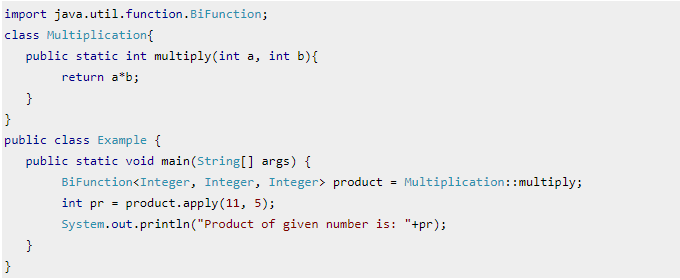
// Calling the method of functional interface

ref.display();

}

}

1. **Method reference to a static method of a class**



import java.util.function.BiFunction;

class Multiplication{

public static int multiply(int a, int b){

return a\*b;

}

}

public class Example {

public static void main(String[] args) {

BiFunction<Integer, Integer, Integer> product = Multiplication::multiply;

int pr = product.apply(11, 5);

System.out.println("Product of given number is: "+pr);

}

}

1. **Method reference to an instance method of an arbitrary object of a particular type**



import java.util.Arrays;

public class Example {

public static void main(String[] args) {

String[] stringArray = { "Steve", "Rick", "Aditya", "Negan", "Lucy", "Sansa", "Jon"};

/\* Method reference to an instance method of an arbitrary

\* object of a particular type

\*/

Arrays.sort(stringArray, String::compareToIgnoreCase);

for(String str: stringArray){

System.out.println(str);

}

}

}

1. **Method reference to a constructor**



@FunctionalInterface

interface MyInterface{

Hello display(String say);

}

class Hello{

public Hello(String say){

System.out.print(say);

}

}

public class Example {

public static void main(String[] args) {

//Method reference to a constructor

MyInterface ref = Hello::new;

ref.display("Hello World!");

}

}

**Functional Interface**

An interface with only single abstract method is called functional interface. You can either use the predefined functional interface provided by Java or create your own functional interface and use it. **Functional interfaces all have only one abstract method**. That is the reason,they are also known as Single Abstract Method interfaces (SAM Interfaces).

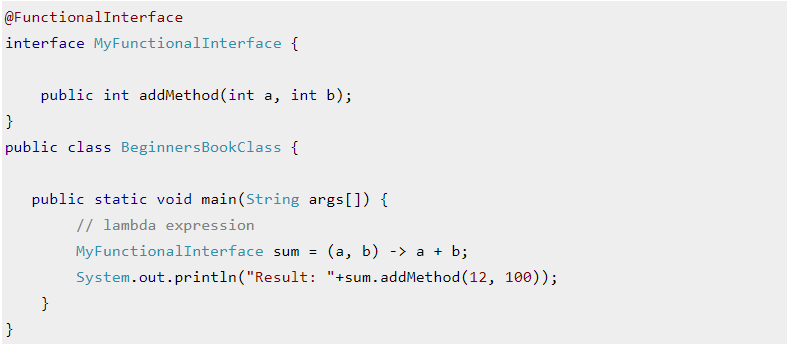
**To use lambda expression in Java, you need to either create your own functional interface or use the pre defined functional interface provided by Java**.

While creating your own functional interface, mark it with @FunctionalInterface annotation, this annotation is introduced in Java 8. Although its optional, you should use it so that you get a compilation error if the interface you marked with this annotation is not following the rules of functional interfaces.

**What are the rules of defining a functional interface?**

The functional interface should have **Only one abstract method**. Along with the one abstract method, they can have any number of default and static methods.

**Example 1: Creating your own functional interface**



@FunctionalInterface

interface MyFunctionalInterface {

public int addMethod(int a, int b);

}

public class BeginnersBookClass {

public static void main(String args[]) {

// lambda expression

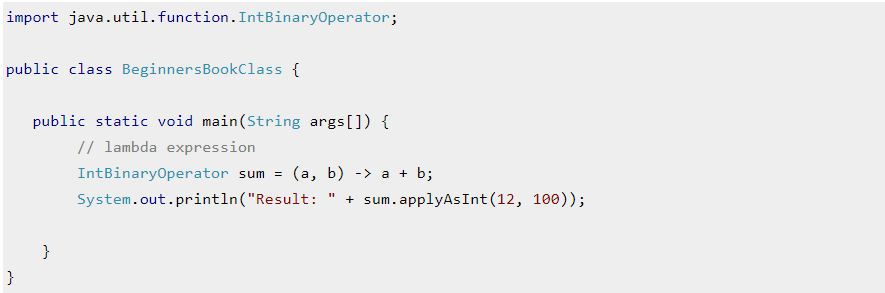
MyFunctionalInterface sum = (a, b) -> a + b;

System.out.println("Result: "+sum.addMethod(12, 100));

}

}

**Example 2: Using predefined functional interface**



import java.util.function.IntBinaryOperator;

public class BeginnersBookClass {

public static void main(String args[]) {

// lambda expression

IntBinaryOperator sum = (a, b) -> a + b;

System.out.println("Result: " + sum.applyAsInt(12, 100));

}

}

**Functional interface example: using anonymous inner class vs using lambda expression**

We have been using functional interfaces even prior to java8, they were used by creating anonymous inner classes using these interfaces. You must have seen functional interfaces such as Runnable, ActionListener, Comparator etc. They all have single abstract method.

Lets see an example of ActionListener to see how it was used with Anonymous inner class and how it can be implemented using lambda expression.

**ActionListener Example: Before Java 8: Using anonymous inner class**



import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

class Example extends JFrame

{

JButton button;

public Example()

{

setTitle("Button Action Example without Lambda Expression");

setSize(400,300);

setVisible(true);

setLayout(new FlowLayout());

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

button = new JButton("Button");

button.setBounds(100,100,90,40);

button.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

System.out.println("You clicked the button.");

}

});

add(button);

}

public static void main(String args[])

{

new Example();

}

}

**ActionListener Example: Lambda Expression**



import javax.swing.\*;

import java.awt.\*;

class Example extends JFrame

{

JButton button;

public Example()

{

setTitle("Button Action Example using Lambda Expression");

setSize(400,300);

setVisible(true);

setLayout(new FlowLayout());

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

button = new JButton("Button");

button.setBounds(100,100,90,40);

//Lambda expression

button.addActionListener(e->

System.out.println("You clicked the button."));

add(button);

}

public static void main(String args[])

{

new Example();

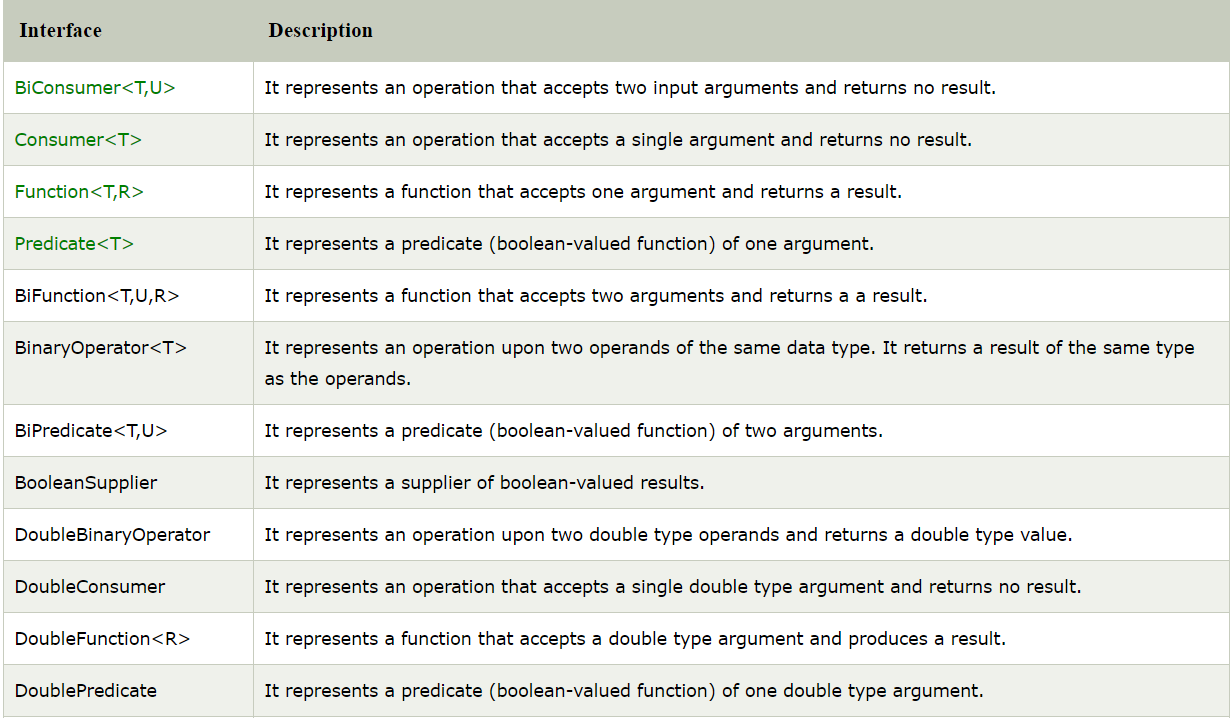
}

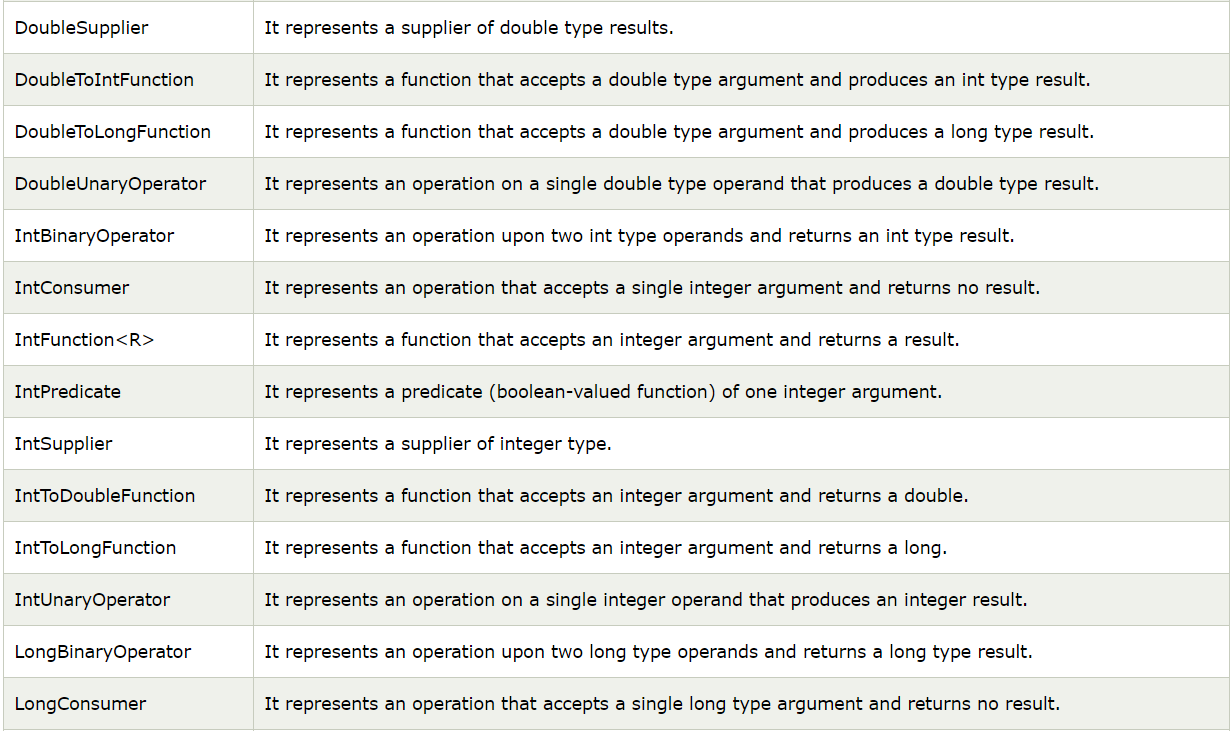
}

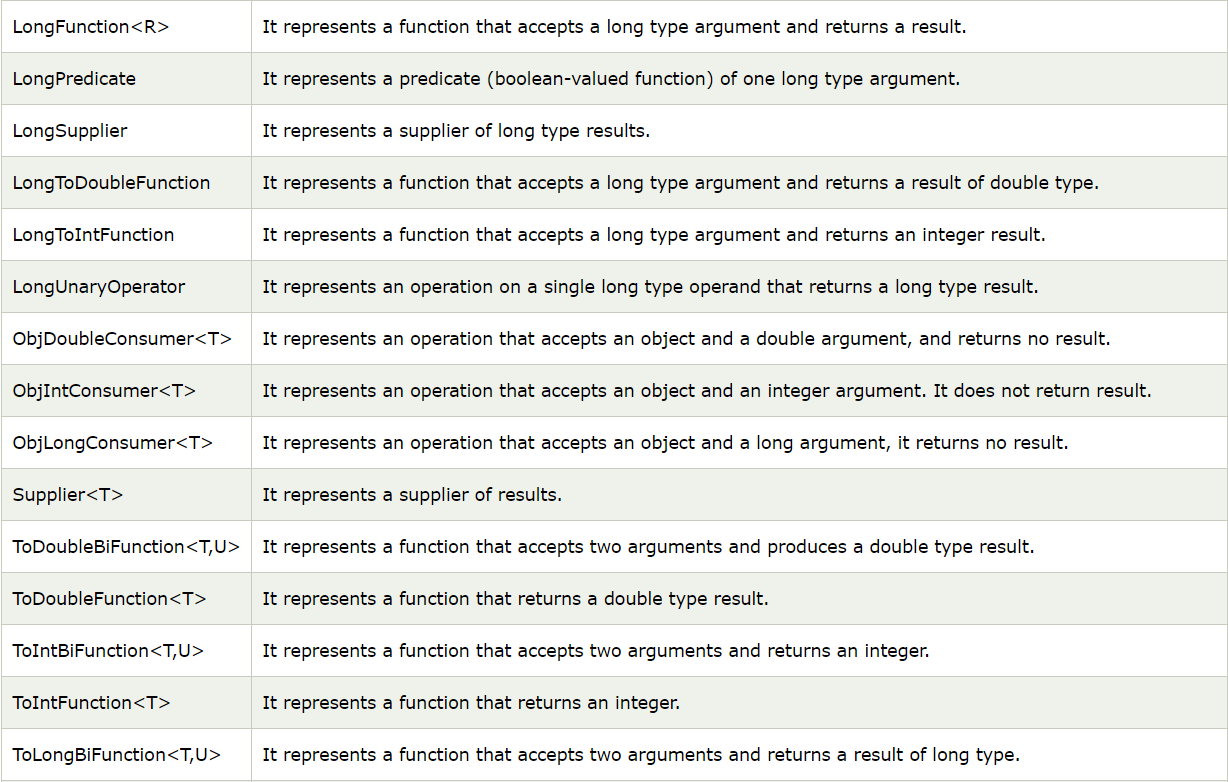
**Java Predefined-Functional Interfaces**

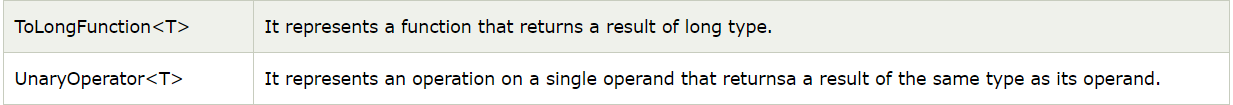
Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.

You can also define your own custom functional interface. Following is the list of functional interface which are placed in java.util.function package.









**Stream API**

Stream API is another new feature of java 8. All the classes and interfaces of this API is in the **java.util.stream** package. By using streams we can perform various aggregate operations on the data returned from collections, arrays, Input/Output operations.

Before we see how stream API can be used in Java, let’s see an example to understand the use of streams.

**Java Stream Example**

To understand how stream works, lets take an example without using stream and then we will see the same example with streams.

**Finding certain strings without using Stream**



import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

int count = 0;

for (String str : names) {

if (str.length() < 6)

count++;

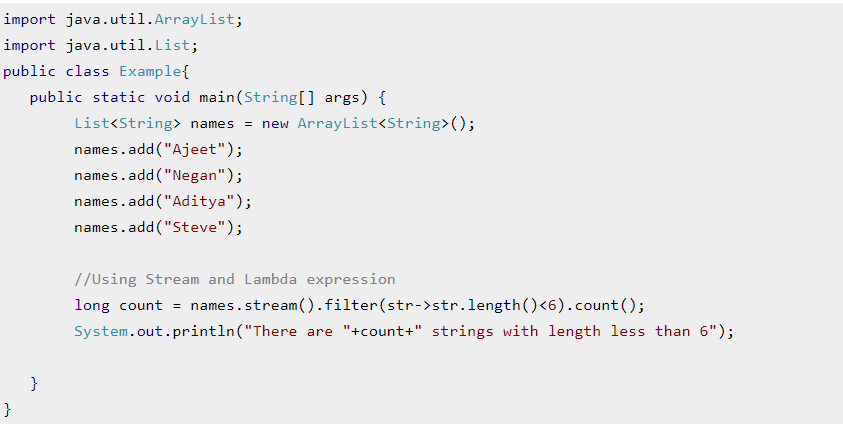
}

System.out.println("There are "+count+" strings with length less than 6");

}

}

**Same example using Stream**



import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

//Using Stream and Lambda expression

long count = names.stream().filter(str->str.length()<6).count();

System.out.println("There are "+count+" strings with length less than 6");

}

}

**How to work with Stream in Java**

Working of stream can be explained in three stages:

1. Create a stream

2. Perform **intermediate operations** on the initial stream to transform it into another stream and so on on further intermediate operations. In the above example, the filter() operation is intermediate operation, there can be more than one intermediate operations.

3. Perform **terminal operation** on the final stream to get the result. In the above example, the count() operation is terminal operation.

**Java Stream Features**

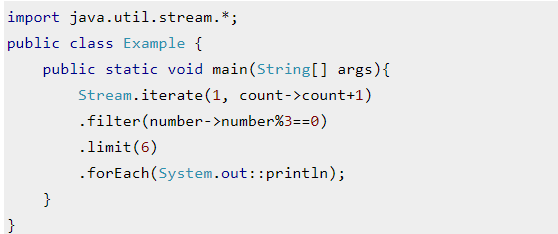
1. Stream **does not store** the elements. it simply performs the aggregate operations(such as filter() and count() that we have seen in the above example) to get the desired stream of data.

2. The aggregate operations that we perform on the collection, array or any other data source **do not change** the data of the source, they simply return a new stream. For example the code we have seen above is filtering the strings with length less than 6 using the stream operations but it didn’t change the elements of the list.

3. All the stream operations are **lazy** in nature which means they are not executed until they are needed. For example, if we want to display only the first 2 elements of a list using stream, the stream operation would stop at the end of second iteration after displaying the second element of list.

Let’s see few examples of Java Stream:

Java Stream Example 1: Iterating and displaying selected integers



import java.util.stream.\*;

public class Example {

public static void main(String[] args){

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

.filter(number->number%3==0)

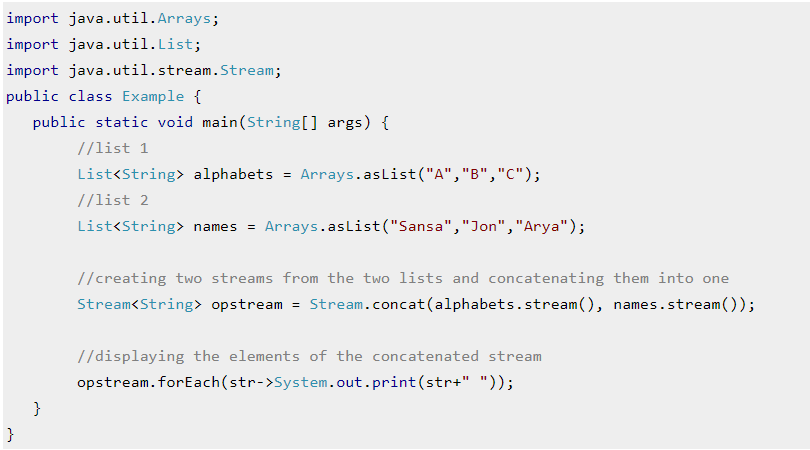
.limit(6)

.forEach(System.out::println);

}

}

**Java Stream Example 2: Concatenating two streams**



import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class Example {

public static void main(String[] args) {

//list 1

List<String> alphabets = Arrays.asList("A","B","C");

//list 2

List<String> names = Arrays.asList("Sansa","Jon","Arya");

//creating two streams from the two lists and concatenating them into one

Stream<String> opstream = Stream.concat(alphabets.stream(), names.stream());

//displaying the elements of the concatenated stream

opstream.forEach(str->System.out.print(str+" "));

}

}

**Stream Filter**

The filter() is an intermediate operation that reads the data from a stream and returns a new stream after transforming the data based on the given condition.

**A Simple Example of Java Stream Filter()**

In this example we are creating a stream from the list of names using stream() method and then we are creating another stream of long names using stream filter(). As I mentioned above, the stream filter transforms the data of one stream into another stream.



import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class Example {

public static void main(String[] args) {

List<String> names = Arrays.asList("Melisandre","Sansa","Jon","Daenerys","Joffery");

//Creating the stream of all names

Stream<String> allNames = names.stream();

//Creating another stream by filtering long names using filter()

Stream<String> longNames = allNames.filter(str -> str.length() > 6);

//displaying the long names

longNames.forEach(str->System.out.print(str+" "));

}

}

**Example 1: Stream filter() and collect()**

We can create a stream and apply a filter in a one line as shown in the example below. The collect() method here collects the final stream and converts it into a list.



import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<String> names = Arrays.asList("Melisandre","Sansa","Jon","Daenerys","Joffery");

List<String> longnames = names.stream() // converting the list to stream

.filter(str -> str.length() > 6) // filter the stream to create a new stream

.collect(Collectors.toList()); // collect the final stream and convert it to a List

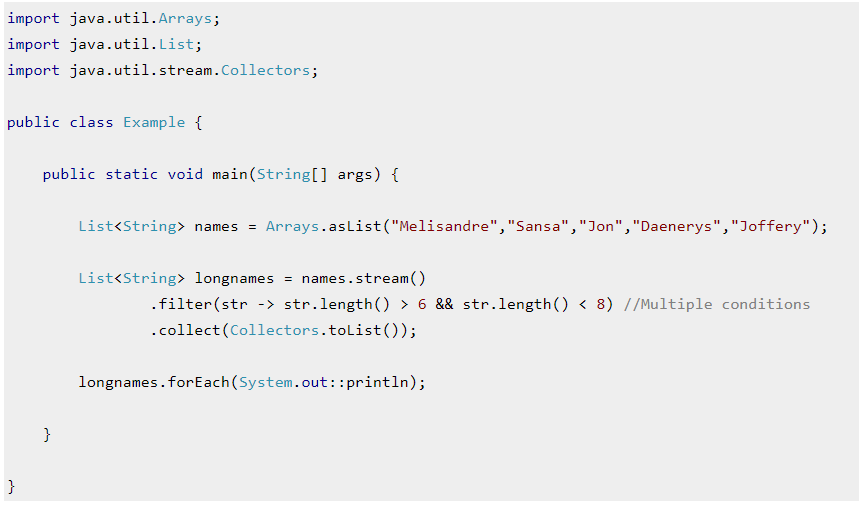
longnames.forEach(System.out::println);

}

}

**Example 2: Stream filter() with multiple conditions**

We can have more than one conditions in the filter() method joined using the logical operators in java. In the following example, we have two conditions in the filter method joined using and (&&) logical operator.



import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<String> names = Arrays.asList("Melisandre","Sansa","Jon","Daenerys","Joffery");

List<String> longnames = names.stream()

.filter(str -> str.length() > 6 && str.length() < 8) //Multiple conditions

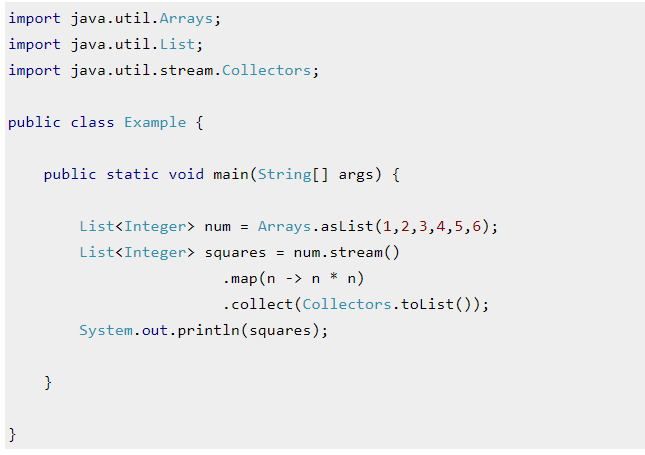
.collect(Collectors.toList());

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

}

}

**Example 3: Stream filter() and map() method in Java**



import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<Integer> num = Arrays.asList(1,2,3,4,5,6);

List<Integer> squares = num.stream()

.map(n -> n \* n)

.collect(Collectors.toList());

System.out.println(squares);

}

}

**Example 4: Filter Map by Keys**



import java.util.Map;

import java.util.HashMap;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

Map<Integer, String> hmap = new HashMap<Integer, String>();

hmap.put(11, "Apple");

hmap.put(22, "Orange");

hmap.put(33, "Kiwi");

hmap.put(44, "Banana");

Map<Integer, String> result = hmap.entrySet()

.stream()

.filter(map -> map.getKey().intValue() <= 22)

.collect(Collectors.toMap(map -> map.getKey(), map -> map.getValue()));

System.out.println("Result: " + result);

}

}

**Example 5: Filter Map by Values**



import java.util.Map;

import java.util.HashMap;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

Map<Integer, String> hmap = new HashMap<Integer, String>();

hmap.put(11, "Apple");

hmap.put(22, "Orange");

hmap.put(33, "Kiwi");

hmap.put(44, "Banana");

Map<Integer, String> result = hmap.entrySet()

.stream()

.filter(map -> "Orange".equals(map.getValue()))

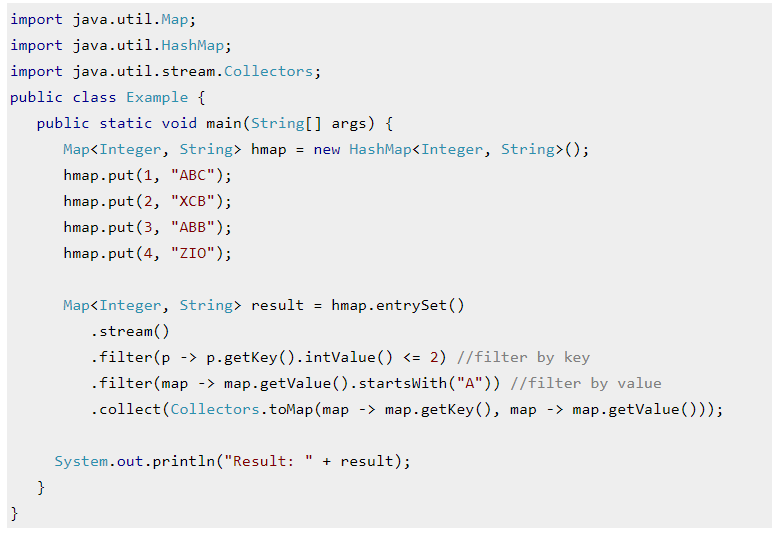
.collect(Collectors.toMap(map -> map.getKey(), map -> map.getValue()));

System.out.println("Result: " + result);

}

}

**Example 6:Filter Map by both Keys and Values**



import java.util.Map;

import java.util.HashMap;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

Map<Integer, String> hmap = new HashMap<Integer, String>();

hmap.put(1, "ABC");

hmap.put(2, "XCB");

hmap.put(3, "ABB");

hmap.put(4, "ZIO");

Map<Integer, String> result = hmap.entrySet()

.stream()

.filter(p -> p.getKey().intValue() <= 2) //filter by key

.filter(map -> map.getValue().startsWith("A")) //filter by value

.collect(Collectors.toMap(map -> map.getKey(), map -> map.getValue()));

System.out.println("Result: " + result);

}

}

**Example 7: A stream with null values**



import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<String> list = Arrays.asList("Java", "Stream", null, "Filter", null);

List<String> result = list.stream().collect(Collectors.toList());

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

}

}

**Example 8: Filter null values from a stream**



import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<String> list = Arrays.asList("Java", "Stream", null, "Filter", null);

List<String> result = list.stream()

.filter(str -> str!=null)

.collect(Collectors.toList());

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

}

}

**Base64(Encode-Decode)**

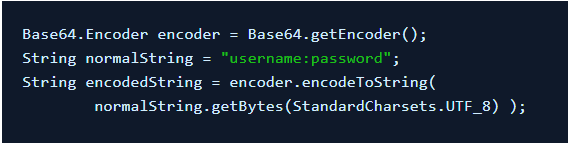
**Base64 support before Java 8**

For many years, java has provided support for base 64 via a non-public class (therefore non-usable) java.util.prefs.Base64 an undocumented class sun.misc.BASE64Encoder. This class has also very limited information in public domain.

**Base64 support from Java 8**

Java 8 has added a class for Base 64 encoding and decoding purpose i.e. java.util.Base64. We will the code examples below for using it.

1. **Encoding a string to base 64**



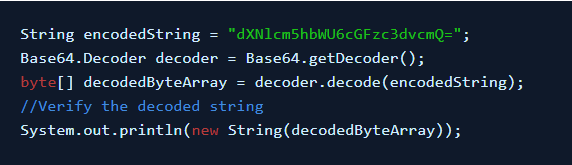
Base64.Encoder encoder = Base64.getEncoder();

String normalString = "username:password";

String encodedString = encoder.encodeToString(

normalString.getBytes(StandardCharsets.UTF\_8) );

1. **Decoding a base 64 encoded string**



String encodedString = "dXNlcm5hbWU6cGFzc3dvcmQ=";

Base64.Decoder decoder = Base64.getDecoder();

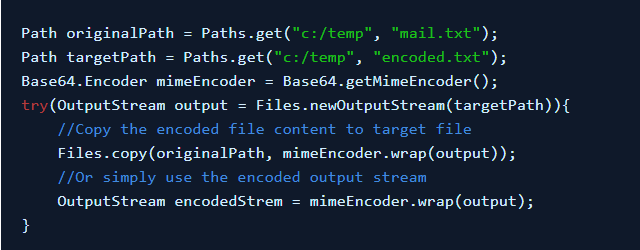
byte[] decodedByteArray = decoder.decode(encodedString);

//Verify the decoded string

System.out.println(new String(decodedByteArray));

1. **Wrap to a base 64 encoded output stream**

If you don’t want to directly work with data and rather prefer to work with streams, you can wrap the output stream such that all data written to this output stream will be automatically base 64 encoded.



Path originalPath = Paths.get("c:/temp", "mail.txt");

Path targetPath = Paths.get("c:/temp", "encoded.txt");

Base64.Encoder mimeEncoder = Base64.getMimeEncoder();

try(OutputStream output = Files.newOutputStream(targetPath)){

//Copy the encoded file content to target file

Files.copy(originalPath, mimeEncoder.wrap(output));

//Or simply use the encoded output stream

OutputStream encodedStrem = mimeEncoder.wrap(output);

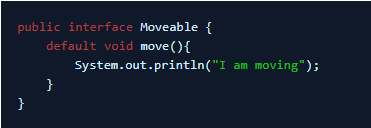
}

**Default Method in Interface**

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

As name implies, default methods in java 8 are simply default. If you do not override them, they are the methods which will be invoked by caller classes. They are defined in interfaces.

**An example**



public interface Moveable {

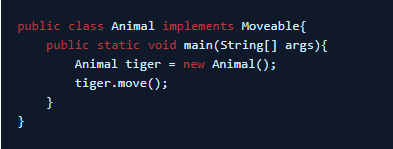
    default void move(){

        System.out.println("I am moving");

    }

}

Moveable interface defines a method move(); and provided a default implementation as well. If any class implements this interface then it need not to implement it’s own version of move() method. It can directly call instance.move();



public class Animal implements Moveable{

    public static void main(String[] args){

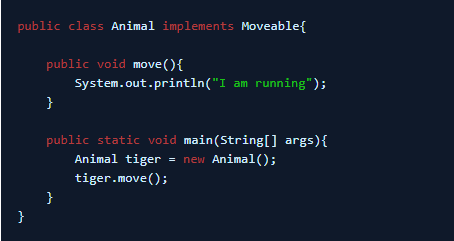
        Animal tiger = new Animal();

        tiger.move();

    }

}

And if class willingly wants to customize the behavior then it can provide it’s own custom implementation and override the method. Now it’s own custom method will be called.



public class Animal implements Moveable{

    public void move(){

        System.out.println("I am running");

    }

    public static void main(String[] args){

        Animal tiger = new Animal();

        tiger.move();

    }

}

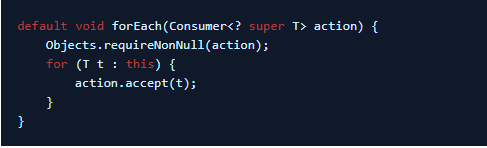
Best part comes as following benefits:

1. Static default methods: You can define static default methods in interface which will be available to all instances of class which implement this interface. This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class. This enables you to define methods out of your class and yet share with all child classes.
2. They provide you an highly desired capability of adding a capability to number of classes without even touching their code. Simply add a default method in interface which they all implement.

**Why default methods were needed in java 8?**

Simplest answer is to enable the functionality of lambda expression in java. Lambda expression are essentially of type of functional interface. To support lambda expressions seamlessly, all core classes have to be modified. But these core classes like java.util.List are implemented not only in JDK classes, but also in thousands of client code as well. Any incompatible change in core classes will back fire for sure and will not be accepted at all.

Default methods break this deadlock and allow adding support for functional interface in core classes. Let’s see an example. Below is a method which has been added to java.lang.Iterable.



default void forEach(Consumer<? super T> action) {

    Objects.requireNonNull(action);

    for (T t : this) {

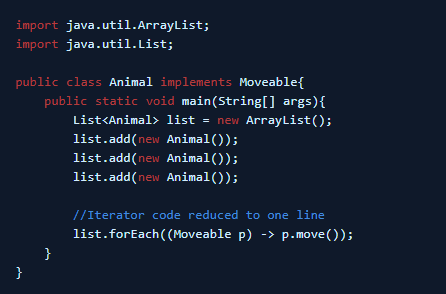
        action.accept(t);

    }

}

Before java 8, if you had to iterate on a java collection then your would get an iterator instance and call it’s next method until hasNext() returns false. This is common code and have been used thousands of time in day to day programming by us. Syntax is also always same. So can we make it compact so that it takes only single line of code and still do the job for us as before. Above function does that.

Now to iterate and perform some simple operation on every item in list, all you need to do is:



import java.util.ArrayList;

import java.util.List;

public class Animal implements Moveable{

    public static void main(String[] args){

        List<Animal> list = new ArrayList();

        list.add(new Animal());

        list.add(new Animal());

        list.add(new Animal());

        //Iterator code reduced to one line

        list.forEach((Moveable p) -> p.move());

    }

}

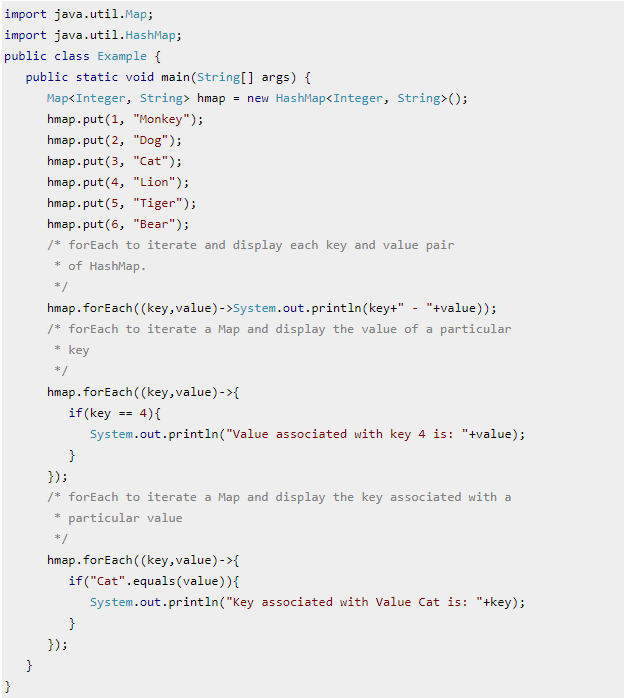
So here, an additional method has been added to List without breaking any custom implementations of it. It has been very desired feature in java since long.

**forEach**

In Java 8, we have a newly introduced forEach method to iterate over collections and Streams in Java.

We will learn how to use forEach() and forEachOrdered() methods to loop a particular collection and stream.

**Example 1: forEach to iterate a Map**



import java.util.Map;

import java.util.HashMap;

public class Example {

public static void main(String[] args) {

Map<Integer, String> hmap = new HashMap<Integer, String>();

hmap.put(1, "Monkey");

hmap.put(2, "Dog");

hmap.put(3, "Cat");

hmap.put(4, "Lion");

hmap.put(5, "Tiger");

hmap.put(6, "Bear");

/\* forEach to iterate and display each key and value pair

\* of HashMap.

\*/

hmap.forEach((key,value)->System.out.println(key+" - "+value));

/\* forEach to iterate a Map and display the value of a particular

\* key

\*/

hmap.forEach((key,value)->{

if(key == 4){

System.out.println("Value associated with key 4 is: "+value);

}

});

/\* forEach to iterate a Map and display the key associated with a

\* particular value

\*/

hmap.forEach((key,value)->{

if("Cat".equals(value)){

System.out.println("Key associated with Value Cat is: "+key);

}

});

}

}

**Example 2: forEach to iterate a List**



import java.util.List;

import java.util.ArrayList;

public class Example {

public static void main(String[] args) {

List<String> fruits = new ArrayList<String>();

fruits.add("Apple");

fruits.add("Orange");

fruits.add("Banana");

fruits.add("Pear");

fruits.add("Mango");

//lambda expression in forEach Method

fruits.forEach(str->System.out.println(str));

}

}

**Example 3: forEach method to iterate a Stream**



import java.util.List;

import java.util.ArrayList;

public class Example {

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Maggie");

names.add("Michonne");

names.add("Rick");

names.add("Merle");

names.add("Governor");

names.stream() //creating stream

.filter(f->f.startsWith("M")) //filtering names that starts with M

.forEach(System.out::println); //displaying the stream using forEach

}

}

**Example 4: Stream forEachOrdered() Method Example**

For sequential streams the order of elements is same as the order in the source, so the output would be same whether you use forEach or forEachOrdered.

However when working with parallel streams, you would always want to use the forEachOrdered() method when the order matters to you, as this method guarantees that the order of elements would be same as the source.

Lets take an example to understand the difference between forEach() and forEachOrdered().



import java.util.List;

import java.util.ArrayList;

public class Example {

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Maggie");

names.add("Michonne");

names.add("Rick");

names.add("Merle");

names.add("Governor");

//forEach - the output would be in any order

System.out.println("Print using forEach");

names.stream()

.filter(f->f.startsWith("M"))

.parallel()

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

/\* forEachOrdered - the output would always be in this order:

\* Maggie, Michonne, Merle

\*/

System.out.println("Print using forEachOrdered");

names.stream()

.filter(f->f.startsWith("M"))

.parallel()

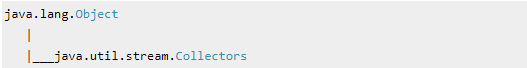
.forEachOrdered(n->System.out.println(n));

}

}

**Collectors Class**

Collectors is a final class that extends the Object class.



**Example: Stream Collectors groupingBy and counting**

In this example, we are grouping the elements of a list using groupingBy() method of Collectors class and printing the occurrences of each element in the list.



import java.util.Arrays;

import java.util.List;

import java.util.Map;

import java.util.function.Function;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<String> names =

Arrays.asList("Jon", "Ajeet", "Steve",

"Ajeet", "Jon", "Ajeet");

Map<String, Long> map =

names.stream().collect(

Collectors.groupingBy(

Function.identity(), Collectors.counting()

)

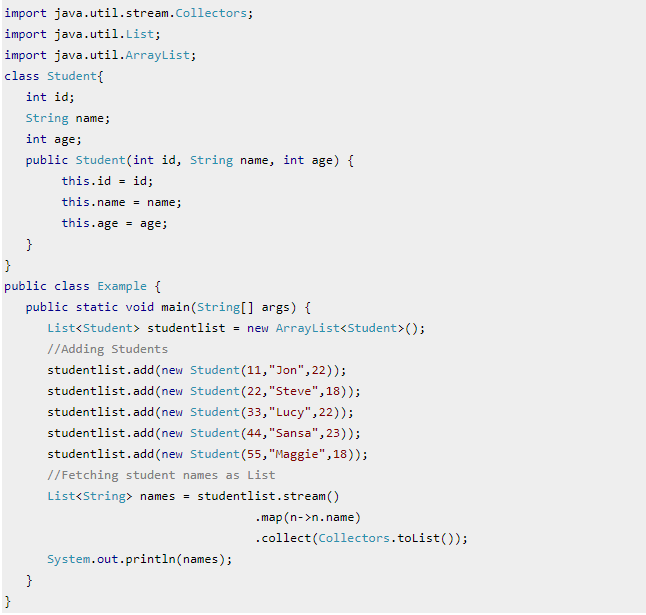
);

System.out.println(map);

}

}

**Example: Stream Collectors example of fetching data as List**



import java.util.stream.Collectors;

import java.util.List;

import java.util.ArrayList;

class Student{

int id;

String name;

int age;

public Student(int id, String name, int age) {

this.id = id;

this.name = name;

this.age = age;

}

}

public class Example {

public static void main(String[] args) {

List<Student> studentlist = new ArrayList<Student>();

//Adding Students

studentlist.add(new Student(11,"Jon",22));

studentlist.add(new Student(22,"Steve",18));

studentlist.add(new Student(33,"Lucy",22));

studentlist.add(new Student(44,"Sansa",23));

studentlist.add(new Student(55,"Maggie",18));

//Fetching student names as List

List<String> names = studentlist.stream()

.map(n->n.name)

.collect(Collectors.toList());

System.out.println(names);

}

}

**Example: Collecting Data as Set**

In this example we are converting the list of students to the stream and then we are applying the Java Stream filter to get the selected records from the stream, after that we are converting that stream to set using Collectors.toSet() method.



import java.util.stream.Collectors;

import java.util.List;

import java.util.Set;

import java.util.ArrayList;

class Student{

int id;

String name;

int age;

public Student(int id, String name, int age) {

this.id = id;

this.name = name;

this.age = age;

}

}

public class Example {

public static void main(String[] args) {

List<Student> studentlist = new ArrayList<Student>();

//Adding Students

studentlist.add(new Student(11,"Jon",22));

studentlist.add(new Student(22,"Steve",18));

studentlist.add(new Student(33,"Lucy",22));

studentlist.add(new Student(44,"Sansa",23));

studentlist.add(new Student(55,"Maggie",18));

//Fetching student data as a Set

Set<Student> students = studentlist.stream()

.filter(n-> n.id>22)

.collect(Collectors.toSet());

//Iterating Set

for(Student stu : students) {

System.out.println(stu.id+" "+stu.name+" "+stu.age);

}

}

}

**Example: Getting the average age of students using averagingInt() method**



import java.util.stream.Collectors;

import java.util.List;

import java.util.ArrayList;

class Student{

int id;

String name;

int age;

public Student(int id, String name, int age) {

this.id = id;

this.name = name;

this.age = age;

}

}

public class Example {

public static void main(String[] args) {

List<Student> studentlist = new ArrayList<Student>();

//Adding Students

studentlist.add(new Student(11,"Jon",22));

studentlist.add(new Student(22,"Steve",18));

studentlist.add(new Student(33,"Lucy",22));

studentlist.add(new Student(44,"Sansa",23));

studentlist.add(new Student(55,"Maggie",18));

//Getting the average Age

Double avgAge = studentlist.stream()

.collect(Collectors.averagingInt(s->s.age));

System.out.println("Average Age of Students is: "+avgAge);

}

}

**Optional Class**

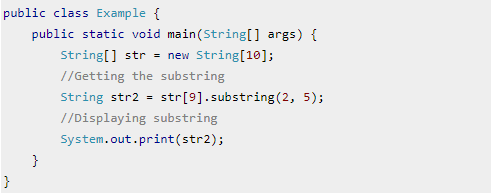
In Java 8, we have a newly introduced Optional class in java.util package.

This class is introduced to avoid NullPointerException that we frequently encounter if we do not perform null checks in our code. Using this class we can easily check whether a variable has null value or not and by doing this we can avoid the NullPointerException.

Before we see the example of Optional class, lets see what happens when we don’t use Optional class and do not perform null check.

**Example: Without using Optional class**

In this example, we didn’t assign the value to the String str and we are trying to get the substring out of it. Since there is no value present in the str, the program is throwing NullPointerException.



**Solution: Using Optional Class**

Optional.ofNullable() method of the Optional class, returns a Non-empty Optional if the given object has a value, otherwise it returns an empty Optional.

We can check whether the returned Optional value is empty or non-empty using the isPresent() method.



//Importing Optional class

import java.util.Optional;

public class Example {

public static void main(String[] args) {

String[] str = new String[10];

Optional<String> isNull = Optional.ofNullable(str[9]);

if(isNull.isPresent()){

//Getting the substring

String str2 = str[9].substring(2, 5);

//Displaying substring

System.out.print("Substring is: "+ str2);

}

else{

System.out.println("Cannot get the substring from an empty string");

}

str[9] = "AgraIsCool";

Optional<String> isNull2 = Optional.ofNullable(str[9]);

if(isNull2.isPresent()){

//Getting the substring

String str2 = str[9].substring(2, 5);

//Displaying substring

System.out.print("Substring is: "+ str2);

}

else{

System.out.println("Cannot get the substring from an empty string");

}

}

}

**Example: Optional isPresent() vs ifPresent() methods**

In the above example, we have seen that by using isPresent() method we can check whether the particular Optional object(or instance) is empty or no-empty.

There is another method present in the Optional class, which only executes if the given Optional object is non-empty, the method is ifPresent(). Lets see an example to understand the difference.



//Importing Optional class

import java.util.Optional;

public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

//Optional.empty() creates an empty Optional object

Optional<String> nothing = Optional.empty();

/\* isPresent() method: Checks whether the given Optional

\* Object is empty or not.

\*/

if (GOT.isPresent()) {

System.out.println("Watching Game of Thrones");

}

else {

System.out.println("I am getting Bored");

}

/\* ifPresent() method: It executes only if the given Optional

\* object is non-empty.

\*/

//This will print as the GOT is non-empty

GOT.ifPresent(s -> System.out.println("Watching GOT is fun!"));

//This will not print as the nothing is empty

nothing.ifPresent(s -> System.out.println("I prefer getting bored"));

}

}

JavaScript Nashorn

**Parallel Array Sort**

Java 8 introduced a new method parallelSort() in the Arrays class of java.util package. This method is introduced to support the parallel sorting of array elements.

Algorithm of parallel sorting:

1. The given array is divided into the sub arrays and the sub arrays are further divided into the their sub arrays, this happens until the sub array reaches a minimum granularity.

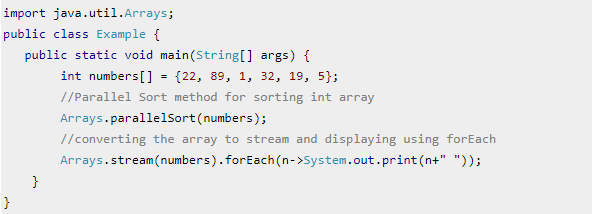
2. The sub arrays are sorted individually by multiple threads. The parallel sort uses Fork/Join Framework for sorting sub arrays parallelly.

3. The sorted sub arrays are merged.

**Advantage of Parallel Sort Over Simple Sort:**

The parallelSort() method uses the concept of multithreading which makes it much faster compared to the normal sort when there are lot of elements.

**Example: Sorting Primitive Data types with Parallel Sort**



import java.util.Arrays;

public class Example {

public static void main(String[] args) {

int numbers[] = {22, 89, 1, 32, 19, 5};

//Parallel Sort method for sorting int array

Arrays.parallelSort(numbers);

//converting the array to stream and displaying using forEach

Arrays.stream(numbers).forEach(n->System.out.print(n+" "));

}

}

**Type Inference Improvement**

Before Java 7:

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

After Java 7:

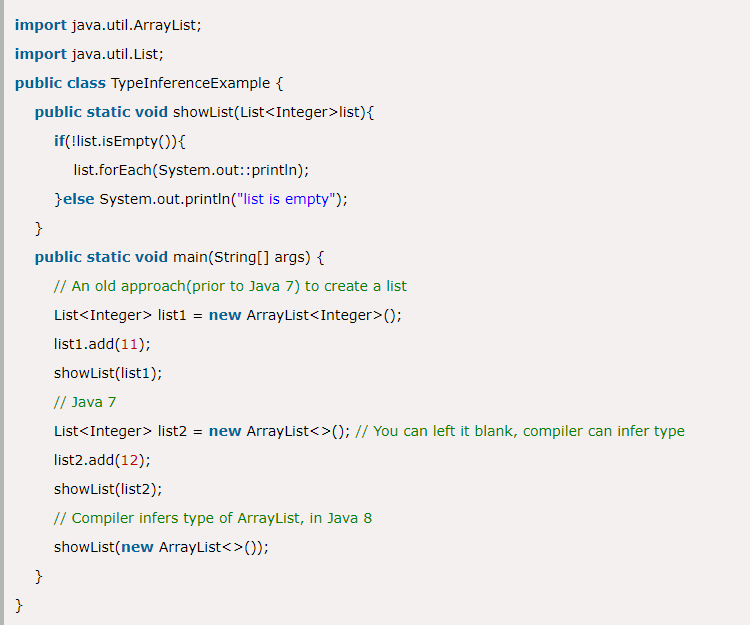
List<Integer> list2 = **new** ArrayList<>();

After Java 8:

In Java 8, you can call specialized method without explicitly mentioning of type of arguments.

showList(**new** ArrayList<>());

Example:



**Predicate**

In Java 8, Predicate is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

**Using Predicate on a collection**

public class Employee {

public Employee(Integer id, Integer age, String gender, String fName, String lName){

this.id = id;

this.age = age;

this.gender = gender;

this.firstName = fName;

this.lastName = lName;

}

private Integer id;

private Integer age;

private String gender;

private String firstName;

private String lastName;

//Please generate Getter and Setters

//To change body of generated methods, choose Tools | Templates.

@Override

public String toString() {

return this.id.toString()+" - "+this.age.toString();

}

}

**1. All Employees who are male and age more than 21**

public static Predicate<Employee> isAdultMale()

{

return p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase("M");

}

**2. All Employees who are female and age more than 18**

public static Predicate<Employee> isAdultFemale()

{

return p -> p.getAge() > 18 && p.getGender().equalsIgnoreCase("F");

}

**3. All Employees whose age is more than a given age**

public static Predicate<Employee> isAgeMoreThan(Integer age)

{

return p -> p.getAge() > age;

}

**EmployeePredicates.java**

import java.util.List;

import java.util.function.Predicate;

import java.util.stream.Collectors;

public class EmployeePredicates

{

public static Predicate<Employee> isAdultMale() {

return p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase("M");

}

public static Predicate<Employee> isAdultFemale() {

return p -> p.getAge() > 18 && p.getGender().equalsIgnoreCase("F");

}

public static Predicate<Employee> isAgeMoreThan(Integer age) {

return p -> p.getAge() > age;

}

public static List<Employee> filterEmployees (List<Employee> employees,

Predicate<Employee> predicate)

{

return employees.stream()

.filter( predicate )

.collect(Collectors.<Employee>toList());

}

}

**TestEmployeePredicates.java**

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import static predicateExample.EmployeePredicates.\*;

public class TestEmployeePredicates

{

public static void main(String[] args)

{

Employee e1 = new Employee(1,23,"M","Rick","Beethovan");

Employee e2 = new Employee(2,13,"F","Martina","Hengis");

Employee e3 = new Employee(3,43,"M","Ricky","Martin");

Employee e4 = new Employee(4,26,"M","Jon","Lowman");

Employee e5 = new Employee(5,19,"F","Cristine","Maria");

Employee e6 = new Employee(6,15,"M","David","Feezor");

Employee e7 = new Employee(7,68,"F","Melissa","Roy");

Employee e8 = new Employee(8,79,"M","Alex","Gussin");

Employee e9 = new Employee(9,15,"F","Neetu","Singh");

Employee e10 = new Employee(10,45,"M","Naveen","Jain");

List<Employee> employees = new ArrayList<Employee>();

employees.addAll(Arrays.asList(new Employee[]{e1,e2,e3,e4,e5,e6,e7,e8,e9,e10}));

System.out.println( filterEmployees(employees, isAdultMale()) );

System.out.println( filterEmployees(employees, isAdultFemale()) );

System.out.println( filterEmployees(employees, isAgeMoreThan(35)) );

//Employees other than above collection of "isAgeMoreThan(35)"

//can be get using negate()

System.out.println(filterEmployees(employees, isAgeMoreThan(35).negate()));

}

}

**Read File**

We will learn to read a file line by line using stream api. Also learn to iterate through lines and filter the file content based on some conditions.

**Java 7 – Read file using FileReader**

private static void readLinesUsingFileReader() throws IOException

{

    File file = new File("c:/temp/data.txt");

    FileReader fr = new FileReader(file);

    BufferedReader br = new BufferedReader(fr);

    String line;

    while((line = br.readLine()) != null)

    {

        if(line.contains("password")){

            System.out.println(line);

        }

    }

    br.close();

    fr.close();

}

**Java 8 read file – line by line**

Path filePath = Paths.get("c:/temp", "data.txt");

//try-with-resources

try (Stream<String> lines = Files.lines( filePath ))

{

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

}

catch (IOException e)

{

    e.printStackTrace();

}

**Java 8 read file – filtering stream of lines**

Path filePath = Paths.get("c:/temp", "data.txt");

try (Stream<String> lines = Files.lines(filePath))

{

     List<String> filteredLines = lines

                    .filter(s -> s.contains("password"))

                    .collect(Collectors.toList());

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

}

catch (IOException e) {

    e.printStackTrace();

}

**Write to File**

**Java 8 write to file using BufferedWriter**

BufferedWriter is used to write text to a character or byte stream. Before printing the characters, it stores the characters in buffer and print in bunches. Without buffering, each invocation of a print() method would cause characters to be converted into bytes that would then be written immediately to the file, which can be very inefficient.

//Get the file reference

Path path = Paths.get("c:/output.txt");

//Use try-with-resource to get auto-closeable writer instance

try (BufferedWriter writer = Files.newBufferedWriter(path))

{

    writer.write("Hello World !!");

}

**Java 8 Write to file using Files.write()**

String content = "Hello World !!";

Files.write(Paths.get("c:/output.txt"), content.getBytes());

**Java – Date and Time APIs**

1. **Legacy Date Time API**

**Classes**

The primary classes before Java 8 release were :

* [System.currentTimeMillis()](https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#currentTimeMillis()) : represents the current date and time as milliseconds since January 1st 1970.
* [java.util.Date](https://docs.oracle.com/javase/7/docs/api/java/util/Date.html) : represents a specific instant in time, with millisecond precision.
* [java.util.Calendar](https://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html) : an abstract class that provides methods for converting between instances and manipulating the calendar fields in difefrent ways.
* [java.text.SimpleDateFormat](https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html) : a concrete class for formatting and parsing dates in a locale-sensitive manner and any predefined as well as any user-defined pattern.
* [java.util.TimeZone](https://docs.oracle.com/javase/7/docs/api/java/util/TimeZone.html) : represents a time zone offset, and also figures out daylight savings.

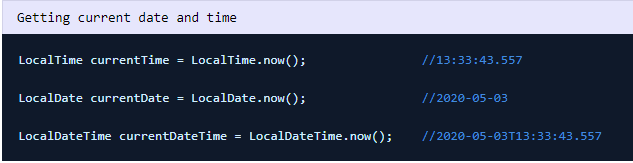
1. **New Date Time API in Java 8**

* [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 : 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.

**Examples**

1. **Get Current Date and Time**

All date-time classes have a factory method now() which is the preferred way to get the current date and time in Java 8.



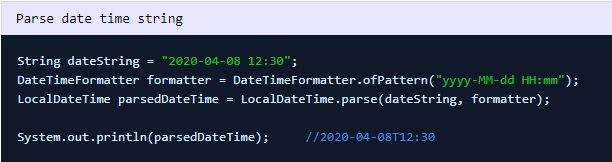
LocalTime currentTime = LocalTime.now(); //13:33:43.557

LocalDate currentDate = LocalDate.now(); //2020-05-03

LocalDateTime currentDateTime = LocalDateTime.now(); //2020-05-03T13:33:43.557

1. **Parse Date and Time**

Date parsing is done with the help of DateTimeFormatter class and parse() methods in date time classes.



String dateString = "2020-04-08 12:30";

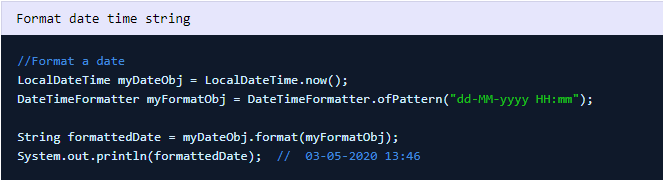
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");

LocalDateTime parsedDateTime = LocalDateTime.parse(dateString, formatter);

System.out.println(parsedDateTime); //2020-04-08T12:30

1. **Format Date and Time**

Date formatting is done with the help of DateTimeFormatter class and format() methods in date time classes.



//Format a date

LocalDateTime myDateObj = LocalDateTime.now();

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

String formattedDate = myDateObj.format(myFormatObj);

System.out.println(formattedDate);  //  03-05-2020 13:46

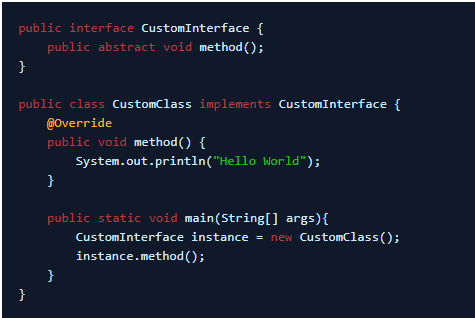
**Java 9**

**Private Methods in Interface – Java 9**

With Java 9, we are allowed to include private methods in interfaces. Using private methods, now encapsulation is possible in interfaces as well.

**Interfaces until Java 7**

In Java 7 and all earlier versions, interfaces were very simple. They could only contain public abstract methods. These interface methods MUST be implemented by classes which choose to implement the interface.



public interface CustomInterface {

    public abstract void method();

}

public class CustomClass implements CustomInterface {

    @Override

    public void method() {

        System.out.println("Hello World");

    }

    public static void main(String[] args){

        CustomInterface instance = new CustomClass();

        instance.method();

    }

}

**Static and defaults methods since Java 8**

From Java 8, in addition to public abstract methods, you can have public static methods and public default methods.



public interface CustomInterface {

    public abstract void method1();

    public default void method2() {

        System.out.println("default method");

    }

    public static void method3() {

        System.out.println("static method");

    }

}

public class CustomClass implements CustomInterface {

    @Override

    public void method1() {

        System.out.println("abstract method");

    }

    public static void main(String[] args){

        CustomInterface instance = new CustomClass();

        instance.method1();

        instance.method2();

        CustomInterface.method3();

    }

}

**Private methods since java 9**

Since java 9, you will be able to add private methods and private static method in interfaces.

These private methods will improve code re-usability inside interfaces. Foe example, if two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to it’s implementing classes.

Using private methods in interfaces have four rules :

1. Private interface method cannot be abstract.
2. Private method can be used only inside interface.
3. Private static method can be used inside other static and non-static interface methods.
4. Private non-static methods cannot be used inside private static methods.



public interface CustomInterface {

    public abstract void method1();

    public default void method2() {

        method4();  //private method inside default method

        method5();  //static method inside other non-static method

        System.out.println("default method");

    }

    public static void method3() {

        method5(); //static method inside other static method

        System.out.println("static method");

    }

    private void method4(){

        System.out.println("private method");

    }

    private static void method5(){

        System.out.println("private static method");

    }

}

public class CustomClass implements CustomInterface {

    @Override

    public void method1() {

        System.out.println("abstract method");

    }

    public static void main(String[] args){

        CustomInterface instance = new CustomClass();

        instance.method1();

        instance.method2();

        CustomInterface.method3();

    }

}

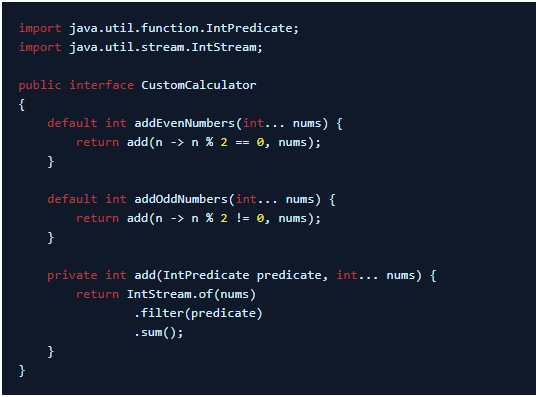
**Java 9 Private Interface Method Example**

Java 9 private interface methods can be static or instance. In both cases, the private method is not inherited by sub-interfaces or implementations. They are mainly there to improve code re-usability within interface only – thus improving encapsulation.

Let’s see a demo to understand the private interface method’s usage.

I am creating a calculator class with two functions. First function will accept some integers and add all even numbers in it. Second function will accept some integers and add all odd numbers in it.

**CustomCalculator.java – Interface**



import java.util.function.IntPredicate;

import java.util.stream.IntStream;

public interface CustomCalculator

{

    default int addEvenNumbers(int... nums) {

        return add(n -> n % 2 == 0, nums);

    }

    default int addOddNumbers(int... nums) {

        return add(n -> n % 2 != 0, nums);

    }

    private int add(IntPredicate predicate, int... nums) {

        return IntStream.of(nums)

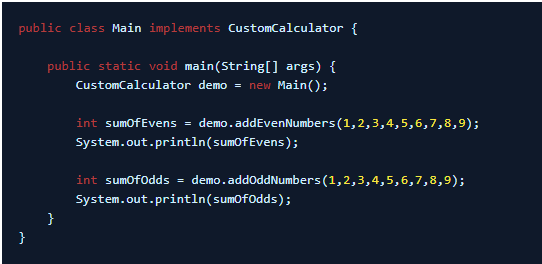
                .filter(predicate)

                .sum();

    }

}

**Main.java – Class**



public class Main implements CustomCalculator {

    public static void main(String[] args) {

        CustomCalculator demo = new Main();

        int sumOfEvens = demo.addEvenNumbers(1,2,3,4,5,6,7,8,9);

        System.out.println(sumOfEvens);

        int sumOfOdds = demo.addOddNumbers(1,2,3,4,5,6,7,8,9);

        System.out.println(sumOfOdds);

    }

}

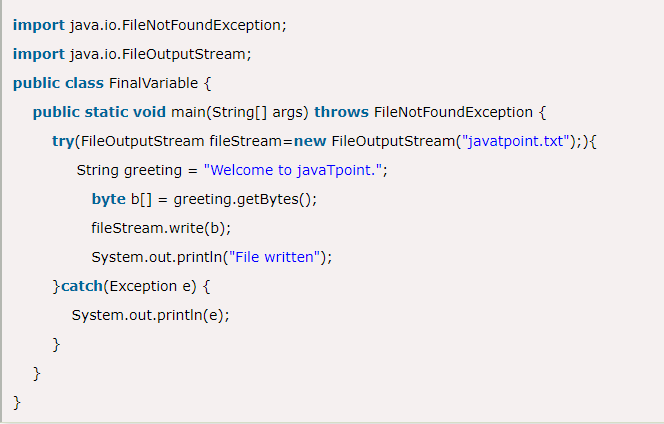
**Java 9 Try With Resource Enhancement**

Java introduced try-with-resource feature in Java 7 that helps to close resource automatically after being used.

In other words, we can say that we don't need to close resources (file, connection, network etc) explicitly, try-with-resource close that automatically by using AutoClosable interface.

In Java 7, try-with-resources has a limitation that requires resource to declare locally within its block.

Example Java 7 Resource Declared within resource block



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);

}

}

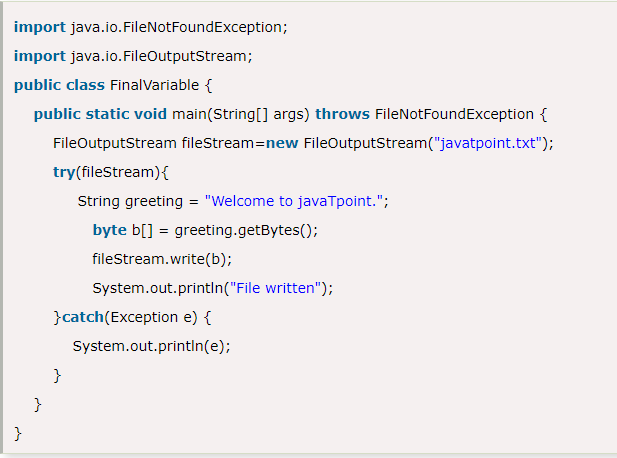
}

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.**

**Java 7 Resource declared outside the resource block**

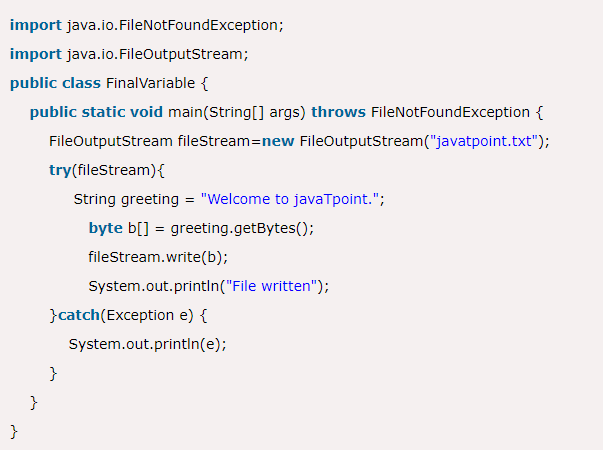
If we do like the following code in Java 7, compiler generates an error message.



To deal with this error, try-with-resource is improved in Java 9 and now we can use reference of the resource that is not declared locally.

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

**Java 9 try-with-resource Example**



**Java 9 - Collection Factory Methods**

Java 9 Collection library includes static factory methods for List, Set and Map interface. These methods are useful to create small number of collection.

Suppose, if we want to create a list of 5 elements, we need to write the following code.

**Java List Example**



import java.util.ArrayList;

import java.util.List;

public class FactoryMethodsExample {

public static void main(String[] args) {

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

list.add("Java");

list.add("JavaFX");

list.add("Spring");

list.add("Hibernate");

list.add("JSP");

for(String l : list){

System.out.println(l);

}

}

}

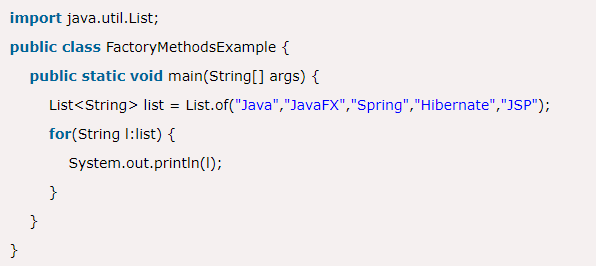
**Factory Methods for Collection**

Factory methods are special type of static methods that are used to create **unmodifiable** instances of collections. It means we can use these methods to create list, set and map of small number of elements.

It is unmodifiable, so adding new element will throw **java.lang.UnsupportedOperationException**

**Java 9 List Factory Method Example**

In Java 9, we can write this code in vary simple manner with the help of **List.of()** factory method.



import java.util.List;

public class FactoryMethodsExample {

public static void main(String[] args) {

List<String> list = List.of("Java","JavaFX","Spring","Hibernate","JSP");

for(String l:list) {

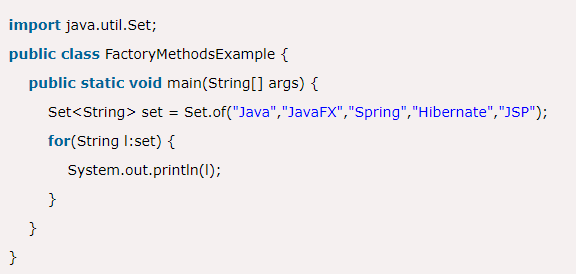
System.out.println(l);

}

}

}

**Java 9 Set Interface Factory Methods Example**



import java.util.Set;

public class FactoryMethodsExample {

public static void main(String[] args) {

Set<String> set = Set.of("Java","JavaFX","Spring","Hibernate","JSP");

for(String l:set) {

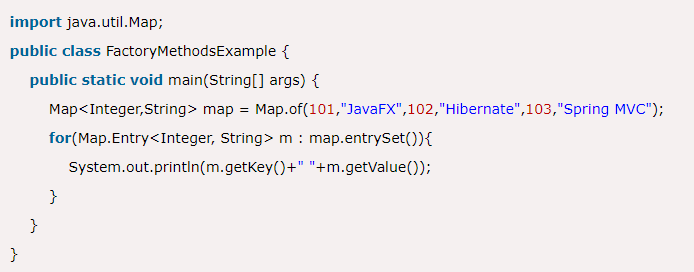
System.out.println(l);

}

}

}

**Java 9 Map Interface Factory Methods Example**



import java.util.Map;

public class FactoryMethodsExample {

public static void main(String[] args) {

Map<Integer,String> map = Map.of(101,"JavaFX",102,"Hibernate",103,"Spring MVC");

for(Map.Entry<Integer, String> m : map.entrySet()){

System.out.println(m.getKey()+" "+m.getValue());

}

}

}

**Java 9 Stream API Improvement**

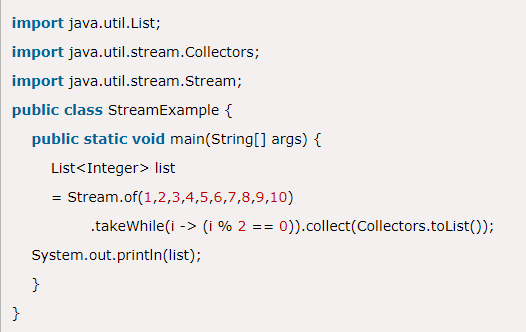
In Java 9, Stream API has improved and new methods are added to the Stream interface.

**Java Stream takeWhile() Method**

Stream takeWhile method takes each element that matches its predicate. It stops when it get unmatched element. It returns a subset of elements that contains all matched elements, other part of stream is discarded.

Java Stream takeWhile() Method Example 1

In this example, we have a list of integers and picks up even values by using takewhile method.



import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class StreamExample {

public static void main(String[] args) {

List<Integer> list

= Stream.of(1,2,3,4,5,6,7,8,9,10)

.takeWhile(i -> (i % 2 == 0)).collect(Collectors.toList());

System.out.println(list);

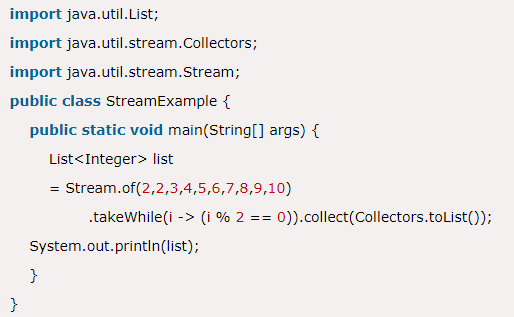
}

}

This example returns an empty list because it fails at first list element, and takewhile stops here.

**Java Stream takeWhile() Method Example 2**

In this example, we are getting first two elements because these are even and stops at third element.



import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class StreamExample {

public static void main(String[] args) {

List<Integer> list

= Stream.of(2,2,3,4,5,6,7,8,9,10)

.takeWhile(i -> (i % 2 == 0)).collect(Collectors.toList());

System.out.println(list);

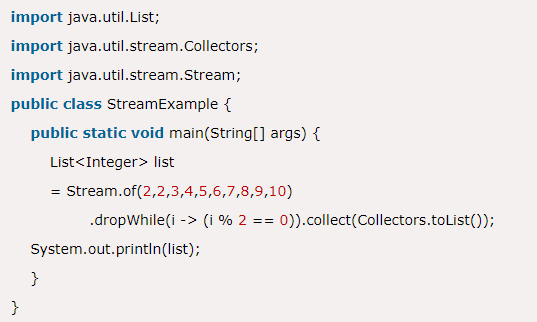
}

}

**Java Stream dropWhile() Method**

It returns a stream that contains elements after dropping the elements that match the given predicate.

Java Stream dropWhile() Method Example



import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class StreamExample {

public static void main(String[] args) {

List<Integer> list

= Stream.of(2,2,3,4,5,6,7,8,9,10)

.dropWhile(i -> (i % 2 == 0)).collect(Collectors.toList());

System.out.println(list);

}

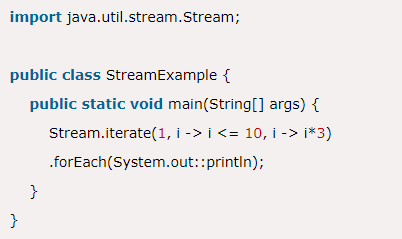
}

**Java Stream Iterate Method**

A new overloaded method iterate is added to the Java 9 stream interface. This method allows us to iterate stream elements till the specified condition.

It takes three arguments, seed, hasNext and next.

**Java Stream Iterate Method Example**



import java.util.stream.Stream;

public class StreamExample {

public static void main(String[] args) {

Stream.iterate(1, i -> i <= 10, i -> i\*3)

.forEach(System.out::println);

}

}

**Java 11**

**String Handling Enhancements**

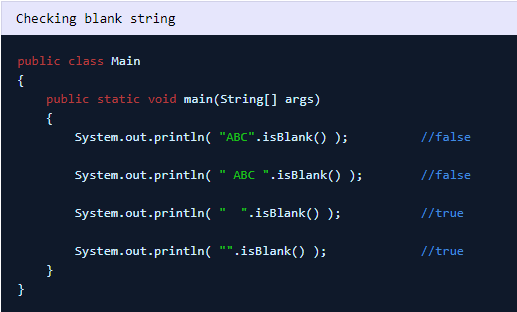
**String isBlank() – Check blank or empty string in Java**

**String.isBlank()** method to determine if a given string is blank or empty or contains only white spaces. isBlank() method has been added in Java 11.

To check is given string does not have even blank spaces, use **String.isEmpty()** method.

**String isBlank() example**

Java program to check if given string is blank or not.



public class Main

{

    public static void main(String[] args)

    {

        System.out.println( "ABC".isBlank() );          //false

        System.out.println( " ABC ".isBlank() );        //false

        System.out.println( "  ".isBlank() );           //true

        System.out.println( "".isBlank() );             //true

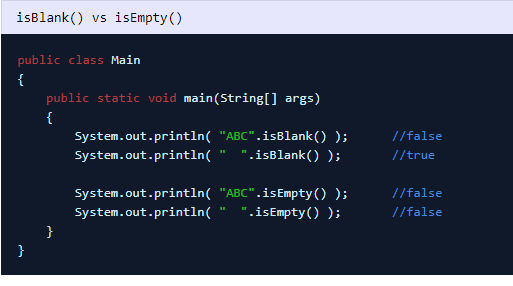
    }

}

**isBlank() vs isEmpty()**

Both methods are used to check for blank or empty strings in java. The difference between both methods is that isEmpty() method returns true if, and only if, string length is 0.

isBlank() method only checks for non-whitespace characters. It does not check the string length.



public class Main

{

    public static void main(String[] args)

    {

        System.out.println( "ABC".isBlank() );      //false

        System.out.println( "  ".isBlank() );       //true

        System.out.println( "ABC".isEmpty() );      //false

        System.out.println( "  ".isEmpty() );       //false

    }

}

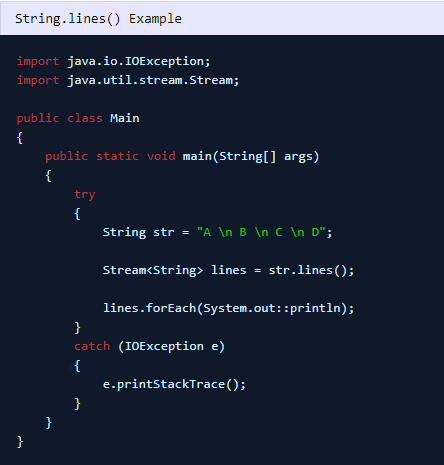
**String lines() – Get stream of lines – Java 11**

Convert multi-line string into stream of lines using String.lines() method in Java 11.

This method is useful when we want to read content from a file and process each string separately.

**Java program to get stream of lines**

Java program to read a file and get the content as stream of lines.



import java.io.IOException;

import java.util.stream.Stream;

public class Main

{

    public static void main(String[] args)

    {

        try

        {

            String str = "A \n B \n C \n D";

            Stream<String> lines = str.lines();

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

        }

        catch (IOException e)

        {

            e.printStackTrace();

        }

    }

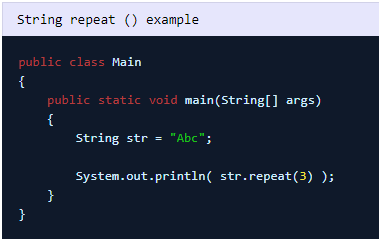
}

**String repeat() – Repeat string N times in Java**

Repeat a given string N times, to produce a new string which contains all the repetitions, though a simple Java program.

**Example**

Java program to repeat string ‘Abc’ to 3 times.



public class Main

{

    public static void main(String[] args)

    {

        String str = "Abc";

        System.out.println( str.repeat(3) );

    }

}

**String strip() – Remove leading and trailing white spaces**

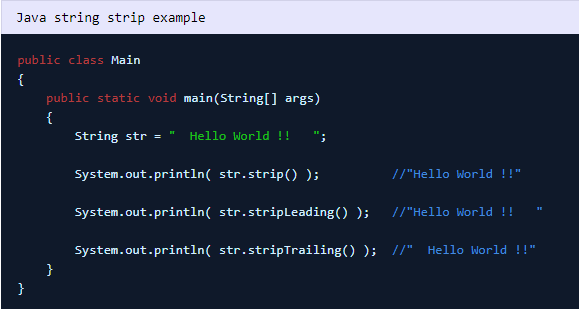
String class’s strip(), stripLeading() and stripTrailing() methods to remove unwanted white spaces from a given string in Java 11.

**String strip() APIs**

Since Java 11, String class includes 3 more methods which help in removing extra white-spaces.

* **String strip()** – returns a string whose value is given string, with all leading and trailing white space removed. Please note that String.trim() method also produces the same result.
* **String stripLeading()** – returns a string whose value is given string, with all leading white space removed.
* **String stripTrailing()** – returns a string whose value is given string, with all trailing white space removed.

**Example**



public class Main

{

    public static void main(String[] args)

    {

        String str = "  Hello World !!   ";

        System.out.println( str.strip() );          //"Hello World !!"

        System.out.println( str.stripLeading() );   //"Hello World !!   "

        System.out.println( str.stripTrailing() );  //"  Hello World !!"

    }

}

**Files.readString()**

**Java 6 and below:**

Java 6 and below uses [BufferedReader](https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html) class.

It’s readLine() method **reads the file one line at a time** and return the content.



import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class ReadFileToString

{

    public static void main(String[] args)

    {

        String filePath = "c:/temp/data.txt";

        System.out.println( usingBufferedReader( filePath ) );

    }

    //Read file content into the string with - using BufferedReader and FileReader

    //You can use this if you are still not using Java 8

    private static String usingBufferedReader(String filePath)

    {

        StringBuilder contentBuilder = new StringBuilder();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath)))

        {

            String sCurrentLine;

            while ((sCurrentLine = br.readLine()) != null)

            {

                contentBuilder.append(sCurrentLine).append("\n");

            }

        }

        catch (IOException e)

        {

            e.printStackTrace();

        }

        return contentBuilder.toString();

    }

}

**Java 7:**

**Files.readAllBytes() – Read the entire File to String – Java 7**

[readAllBytes()](https://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#readAllBytes(java.nio.file.Path)) method reads all the bytes from a file. The method ensures that the file is closed when all bytes have been read or an I/O error, or other runtime exception, is thrown.

After reading all bytes, we pass those bytes to String class constructor to create a string.



import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Paths;

public class ReadFileToString

{

    public static void main(String[] args)

    {

        String filePath = "c:/temp/data.txt";

        System.out.println( readAllBytesJava7( filePath ) );

    }

    //Read file content into string with - Files.readAllBytes(Path path)

    private static String readAllBytesJava7(String filePath)

    {

        String content = "";

        try

        {

            content = new String ( Files.readAllBytes( Paths.get(filePath) ) );

        }

        catch (IOException e)

        {

            e.printStackTrace();

        }

        return content;

    }

}

**Java 8:**

**Files.lines() – Java 8**

[lines()](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#lines-java.nio.file.Path-java.nio.charset.Charset-) method **read all lines from a file to stream** and populates lazily as the [stream](https://howtodoinjava.com/java8/java-streams-by-examples/) is consumed. Bytes from the file are decoded into characters using the specified charset.



import java.io.IOException;

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.stream.Stream;

public class ReadFileToString

{

    public static void main(String[] args)

    {

        String filePath = "c:/temp/data.txt";

        System.out.println( readLineByLineJava8( filePath ) );

    }

    //Read file content into the string with - Files.lines(Path path, Charset cs)

    private static String readLineByLineJava8(String filePath)

    {

        StringBuilder contentBuilder = new StringBuilder();

        try (Stream<String> stream = Files.lines( Paths.get(filePath), StandardCharsets.UTF\_8))

        {

            stream.forEach(s -> contentBuilder.append(s).append("\n"));

        }

        catch (IOException e)

        {

            e.printStackTrace();

        }

        return contentBuilder.toString();

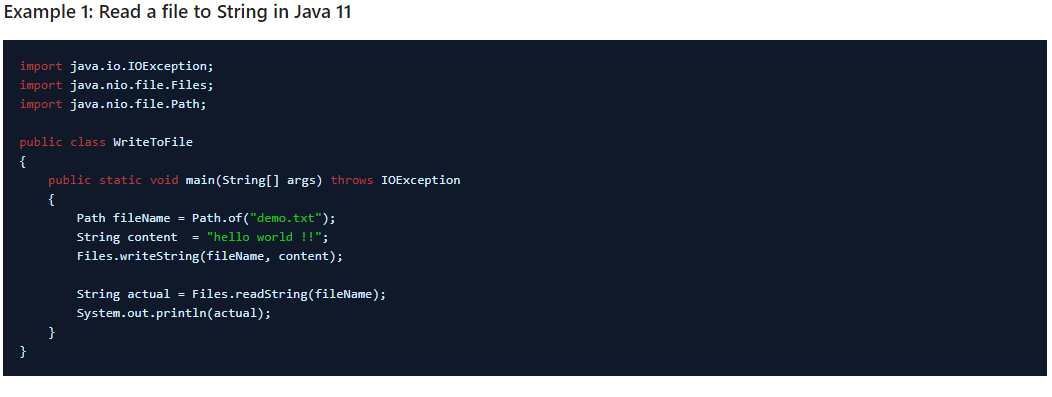
    }

}

**Java 11:**

**Files.readString() – Java 11**

With the new method [readString()](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html#readString(java.nio.file.Path)) introduced in [Java 11](https://howtodoinjava.com/java11/features-enhancements/), it takes only a single line to read a file’s content in to String.



import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

public class WriteToFile

{

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

    {

        Path fileName = Path.of("demo.txt");

        String content  = "hello world !!";

        Files.writeString(fileName, content);

        String actual = Files.readString(fileName);

        System.out.println(actual);

    }

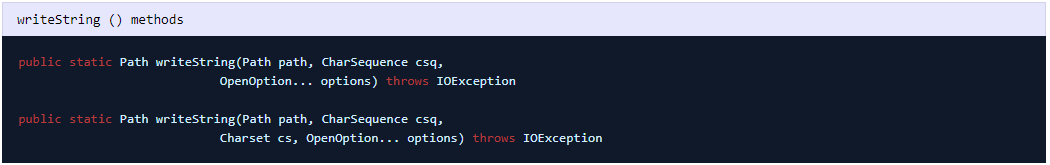
}

**Files.writeString()**

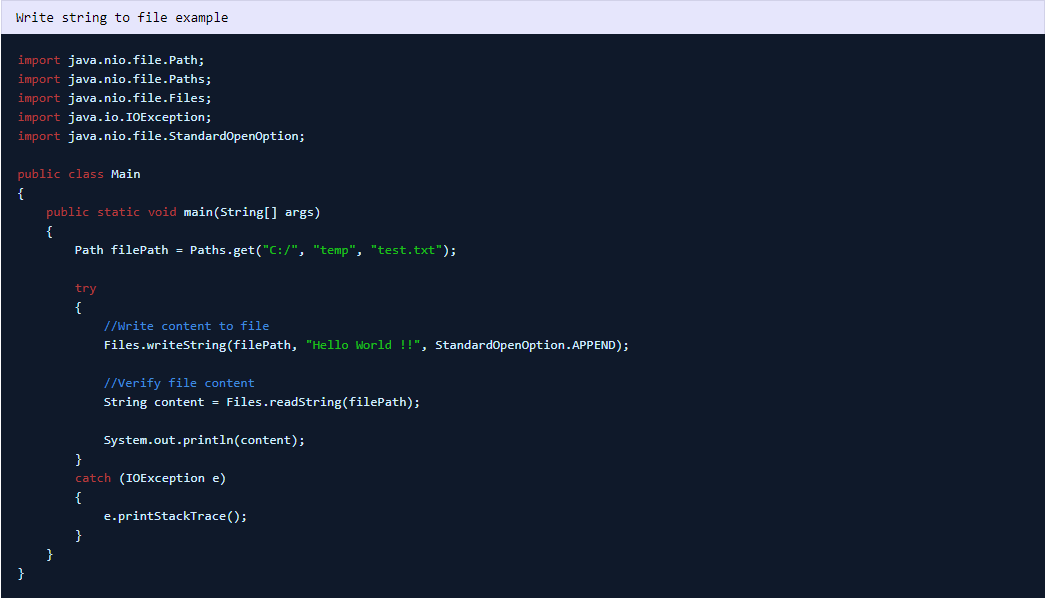
**Java 11:**

**Files writeString() – Java11**

java.nio.file.Files class has two overloaded [static](https://howtodoinjava.com/java/keywords/java-static-keyword/) methods to write content to file.



* First method writes all content to a file, using the **UTF-8** charset.
* First method is equivalent to writeString(path, string, StandardCharsets.UTF\_8, options).
* Second method does the same with with only using the specified charset.
* options specifies how the file is opened.



import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.Files;

import java.io.IOException;

import java.nio.file.StandardOpenOption;

public class Main

{

    public static void main(String[] args)

    {

        Path filePath = Paths.get("C:/", "temp", "test.txt");

        try

        {

            //Write content to file

            Files.writeString(filePath, "Hello World !!", StandardOpenOption.APPEND);

            //Verify file content

            String content = Files.readString(filePath);

            System.out.println(content);

        }

        catch (IOException e)

        {

            e.printStackTrace();

        }

    }

}