Table of Contents

[Java 8 Features 2](#_Toc65948967)

[INTERFACES IN JAVA 8 2](#_Toc65948968)

[DEFAULT METHODS IN MULTIPLE INHERITANCE 2](#_Toc65948969)

[WHY LAMBDAS? 3](#_Toc65948970)

[CONCEPTS OF LAMBDA 3](#_Toc65948971)

[LAMBDA EXPRESSIONS 4](#_Toc65948972)

[VALID AND INVALID LAMBDA EXPRESSIONS 4](#_Toc65948973)

[FUNCTIONAL INTERFACE 4](#_Toc65948974)

[FUNCTIONAL INTERFACES IN INHERITANCE 5](#_Toc65948975)

[USING LAMBDA EXPRESSION 5](#_Toc65948976)

[EXAMPLES 6](#_Toc65948977)

[STREAMS API 8](#_Toc65948978)

[ANALOGY OF STREAMS 8](#_Toc65948979)

[CONCEPTS OF STREAMS 8](#_Toc65948980)

# Java 8 Features

1. **LAMBDA EXPRESSIONS**
2. **FUNCTIONAL INTERFACE**
3. **METHOD REFERENCES (::) & CONSTUCTOR REFERENCES**
4. **COLLECTION API**
5. **STREAM API**
6. **DEFAULT & STATIC METHODS IN INTERFACE.**
7. **OPTIONAL CLASS**
8. **DATE AND TIME API (joda.org)**

## INTERFACES IN JAVA 8

Example

|  |  |  |
| --- | --- | --- |
| **INTERFACE**  public interface Greeting {  default void greet() {  System.out.println("Hello World !");  }  static void message() {  System.out.println("Welcome");  }  } | 1. Till Java 1.7 we can only create public abstract methods only 2. With Java 1.8 + we can declare concrete methods too – which can be either default or static method 3. The Interface having just one abstract method are called Functional interface. The functional interface can have any number of default or static methods 4. Default methods are also called Virtual Extension method or Defender methods | |
| **IMPLEMENTATION CLASSES** | | |
| public class PersonImpl implements Greeting{  public static void main(String[] args) {  PersonImpl person = new PersonImpl();  Greeting.message(); 🡨 Calling Static method  person.greet(); 🡨 **Calls the interface method**  }  } | | public class Employee implements Greeting {  public static void main(String[] args) {  Employee employee = new Employee();  Greeting.message();  employee.greet(); 🡨 **Calls the Employee method**  }  @Override  public void greet() {  System.out.println("This is from employee class");  }  } |
| * The default method of an interface can be accessed by its implementation using its object * The static method can be accessed by the implementation class using Interface name. * The default method can be overridden by implementation class, but static method cannot be overridden * The static method of interface cannot be accessed using implementation class name | | |

### DEFAULT METHODS IN MULTIPLE INHERITANCE

* If the interface has default method of same signature and name. Since multiple interface can be implemented by a class- which will leads to ambiguity condition while access the default method.
* The solution to resolve the ambiguity is
  + Override the default method
  + Call the respective default method using the Interface reference.

|  |  |
| --- | --- |
| **public** **interface** Left {  **default** **void** m1() {  System.***out***.println("From left");  }  } | **public** **interface** Right {  **default** **void** m1() {  System.***out***.println("From Right");  }  } |
| **public** **class** Ambiguity **implements** Left, Right {  **public** **static** **void** main(String[] args) {  Ambiguity ambiguity = **new** Ambiguity();  ambiguity.m1();  }  @Override  **public** **void** m1() {  Left.**super**.m1();  Right.**super**.m1();  }  } | |

## WHY LAMBDAS?

1. Enables functional programming in Java (like JavaScript)
2. Readable and concise code
3. Easier-to-use API and Libraries
4. Enable support for parallel processing

## CONCEPTS OF LAMBDA

***Lambda is just function that exists in isolation and can be passed as we do in functional programing Languages like JavaScript.***

|  |  |
| --- | --- |
| var printName = function(){  console.log("Hello World!");  }  var showName= function(fn){  fn();  }  showName(printName); | IN THE JS CODE (Functional Programming)   * “printName” is a variable which contains a value of function type. * As it exists in isolation it can be passed to another function as well as parameter. * Till Java 1.7 we cannot able to do this because all the method is always tied to a class. No method can exist in java in isolation. * Java1.8 enables this feature using Lambda expression where method can exist in isolation and can be treated as values * As function is can be values so it can be passed as parameter. |

**ANALOGY OF LAMBDAS WITH JS**

|  |  |
| --- | --- |
|  | * In Java 1.8 the variable which stores the function has be a functional interface * The RHS of the statement will be a Lambda expression. |

### LAMBDA EXPRESSIONS

**Lambda expressions has “no name”, no return types and no Modifiers.**

|  |  |
| --- | --- |
| **JAVA METHOD** | **EQUIVALENT LAMBDA EXPRESSION** |
| public void print(){ System.out.println(“Hello”); } | () -> System.out.println(“Hello”); |
| public void add(int a, int b){ System.out.println(a+b); } | (a,b) -> System.out.println(a+b); |
| public int square(int n){  return n\*n;  }  **EQUIVALENT LAMBDA EXPRESSION**  n->n\*n | Step 1 : (n) ->{return n\*n;}   * If we have only 1 parameter parenthesis are optional * If compiler can able to predict the return type we can remove the return type * If we have single statement brackets can be removed |

### VALID AND INVALID LAMBDA EXPRESSIONS

|  |  |
| --- | --- |
| **n->n\*n** | Valid |
| **(n)->n\*n** | **Valid** |
| **n->return n\*n** | Invalid. if we are using a return statement, we need to have curly braces “{}” |
| **n->{return n\*n;};** | Valid |
| **n->{return n\*n};** | Invalid |
| **n->{return n\*n;}** | Invalid |
| **n->{ n\*n;};** | Invalid . Because of we have curly braces we need to have return statement |

## FUNCTIONAL INTERFACE

|  |  |
| --- | --- |
| Prior to Java 1.8 – Interface can have only abstract method, but from Java 1.8 interfaces can have   1. **ABSTRACT METHODS** 2. **DEFAULT METHODS** 3. **STATIC METHODS**   All methods in interface are public methods | public interface Calculator {  public abstract int add(int num1, int num2);  default void print(){  System.out.println("Calculator");  }  public static void calculate(){  System.out.println("Calculator Numbers");  }} |
| * The Interface which has only one abstract method is called Functional Interface. It can have any number of default and static methods. * To make sure interface is adhered to the above rule of functional interface it can be annotated with an annotation @FunctionalInterface | **@FunctionalInterface**  public interface Calculator {  public abstract int add(int num1, int num2);  default void print(){ }  public static void calculate(){ }  } |

### FUNCTIONAL INTERFACES IN INHERITANCE

|  |  |
| --- | --- |
| If the parent interface is a functional interface the child interface will be valid functional interface if it does not contain any new abstract method | **@FunctionalInterface**  public interface Parent { void m1(); }  **@FunctionalInterface**  public interface Child extends Parent { } |
| If the parent interface is a functional interface the child interface will be valid functional interface if it defines the exact same of Parent interface | **@FunctionalInterface**  public interface Parent { void m1(); }  **@FunctionalInterface**  public interface Child extends Parent { void m1(); } |
| If the parent interface is a functional interface the child interface will be ***invalid*** *functional interface* if it defines different method as defined in Parent interface | **@FunctionalInterface**  public interface Parent { void m1(); }  **@FunctionalInterface**  public interface Child extends Parent {  void m2() ; 🡨 **Compile Time Error**  } |

## USING LAMBDA EXPRESSION

Lambda expression can be used in context of functional interface only

|  |  |  |
| --- | --- | --- |
| **TRADITIONAL WAY TO USE AN INTERFACE** | | |
| **INTERFACE**  public interface Calculator {  public abstract int add(int num1, int num2);  }  **IMPL CLASS**  public class CalculatorImpl implements Calculator {  @Override  **public** **int** add(**int** num1, **int** num2) {  **return** num1 + num2;  }  } | | **CALLING THE METHOD**  public class LamdaExpression {  public static void main(String[] args) {  Calculator calculator = new CalculatorImpl();  System.*out*.println(calculator.add(1, 2));  }  } |
| * Similar functionality can be achieved by using functional interface and Lambda expression without creating a “CalculatorImpl” class. * Lambda expression checks only the function signature in the functional interface in the LHS. * **Lambda expressions are an implementation of the method of functional interface.** | | |
| Example1  Step 1: Create a functional interface  **public** **interface** Calculator {  **public** **abstract** **int** add(**int** num1, **int** num2);  } | **Step 2: Call the method in the functional interface using Lambda expression**  **public** **class** LamdaExpression {  **public** **static** **void** main(String[] args) {  Calculator calculator = (a, b) -> a + b;  System.***out***.println(calculator.add(1, 2));  }} | |
| Example 2:  **public** **interface** Square {  **public** **abstract** **int** square(**int** num);  } | **public** **class** LamdaExpression {  **public** **static** **void** main(String[] args) {  Square square = n->n\*n;  System.***out***.println(square.square(2));  }} | |
| **PASSING THE LAMBDA EXPRESSION AS VALUES**  public class LamdaExpression {  public static void main(String[] args) {  Calculator calculator = (num1, num2) -> num1 + num2;  *calculateAdd*(calculator);  }  public static void calculateAdd(Calculator calculator) {  System.*out*.println(calculator.add(1, 2));  }  } | | |

### EXAMPLES

#### RUNNABLE INTERFACE IMPLEMENTATION USING LAMBDA EXPRESSION

Examples

|  |
| --- |
| **EXAMPLE 1**  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  Thread thread = **new** Thread(() -> {  **for** (**int** count = 0; count < 10; count++)  System.***out***.println(count);  });  thread.start();  }  } |
| **EXAMPLE 2**  **public** **class** LambdaThread {  **public** **static** **void** main(String[] args) {  Thread thread = **new** Thread(() -> {  Thread.*currentThread*().setName("Child Thread");  **for** (**int** childThreadCount = 0; childThreadCount < 10; childThreadCount++)  System.***out***.println(Thread.*currentThread*().getName() + "-" + childThreadCount);  });  thread.start();  **for** (**int** mainThreadCount = 0; mainThreadCount < 10; mainThreadCount++)  System.***out***.println(Thread.*currentThread*().getName() + "-" + mainThreadCount);  }  } |

#### EXAMPLE 2

|  |  |
| --- | --- |
| **EXAMPLE:**   1. Create a Person object 2. Sort the person based on Last Name 3. Print All person 4. Print person whose first name start with “J” | public class Person {  private String firstName;  private String lastName;  public Person(String firstName, String lastName) {  super();  this.firstName = firstName;  this.lastName = lastName;  }  // getter & setters  @Override toString() {}  } |
| **FUNCTIONAL INTERFACE**  public interface PersonFilter {  public abstract boolean test(Person person);  } | |
| **LAMBDA EXPRESSION**  public class Exercise {  public static void main(String[] args) {  List<Person> persons = Arrays.*asList*(new Person("Charles", "Dickens", 25),  new Person("John", "Doe", 28),  new Person("Samanta", "Beth", 17),  new Person("Dan", "Wilson", 60));  /\*Sort Person Object with Last Name\*/  Comparator<Person> comparator = (p1, p2) -> p1.getLastName().compareTo(p2.getLastName());  Collections.*sort*(persons, comparator);  /\*Print All\*/  *printPersons*(persons,p -> true);  /\*Filter Person's first name start with "J"\*/  *printPersons*(persons,p -> p.getFirstName().startsWith("J"));  /\*Filter Person's Last name start with "W"\*/  *printPersons*(persons,p -> p.getLastName().startsWith("W"));  }  public static void printPersons(List<Person> persons, PersonFilter personFilter) {  for (Person person : persons) {  if (personFilter.test(person)) {  System.*out*.println(person);  }  }  }  } | |

## LAMBDA IN COLLECTIONS

|  |  |
| --- | --- |
| * The compare() method can be considered as functional interface . So Lambda expression can be used for sorting * Overview of compare: |  |

### LIST INTERFACE

* List maintains the insertion order
* To sort a list, we use Collections.sort(<***list***>) . This will sort the list in a natural sorting order
* For custom sorting order (e.g reverse sorting) – we use Collections.sort(<***list***>, ***<Implementer\_of \_Comparator>***)

|  |  |
| --- | --- |
| **SORTING THE LIST WITHOUT USING LAMBDA EXPRESSION** | public class ListSorting {  public static void main(String[] args) {  List<Integer> numList = new ArrayList<Integer>();  numList.add(20);  numList.add(2);  numList.add(10);  numList.add(15);  System.out.println("Without Sorting = " + numList);  Collections.sort(numList); 🡨 Natural Sorting Order  System.out.println("After Natural Sorting = " + numList);  Collections.sort(numList, new ReverseSorting());🡨 **Reverse Sorting** System.out.println("After Custom Sorting(Reverse) = " + numList);  }  }  public class ReverseSorting implements Comparator<Integer> {  @Override  public int compare(Integer o1, Integer o2) {  **return** (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0;  } |
| **SORTING THE LIST USING LAMBDA EXPRESSION** | **public** **class** ListSorting {  **public** **static** **void** main(String[] args) {  List<Integer> numList = **new** ArrayList<Integer>();  numList.add(20);  numList.add(2);  numList.add(10);  numList.add(15);  Collections.sort(numList, (o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);  System.***out***.println("After Custom Sorting(Reverse) = " + numList);  }  } |

### SET INTERFACE

* The set interface does not allow duplicate elements

EXAMPLE: **Set<Integer> numSet = new TreeSet<Integer>();**

When we add the elements in the above **TreeSet** the elements will ordered in a natural sorting order

* If we need to do a custom sorting – we need to use a the another overloaded constructor

EXAMPLE: **Set<Integer> numSet = new TreeSet<Integer>(<Comparator\_Implementer>)**

#### SORTING SET USING LAMBDA EXPRESSION

**public** **class** SetSorting {

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

Set<Integer> numSet = **new** TreeSet<Integer>((o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);

numSet.add(20);

numSet.add(2);

numSet.add(10);

numSet.add(15);

System.***out***.println("After Custom Sorting(Reverse) = " + numSet);

}

}

### MAP INTERFACE

* The map store elements as key value pair

EXAMPLE: **Map<Integer> numMap = new TreeMapt<Integer>();**

When we add the elements in the above **TreeMap** the elements will ordered in a natural sorting order of the keys

* If we need to do a custom sorting – we need to use a the another overloaded constructor

EXAMPLE: **Map <Integer> numMap = new TreeMap<Integer>(<Comparator\_Implementer>)**

#### SORTING MAP USING LAMBDA EXPRESSION

**public** **class** MapSorting {

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

Map<Integer, String> treeMap = **new** TreeMap<Integer, String>((o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);

treeMap.put(10, "John");

treeMap.put(1, "Doe");

treeMap.put(3, "Oliver");

treeMap.put(2, "Wells");

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

}

}

## PRE-DEFINED FUCNTIONAL INTERFACES

Usually to use a lambda expression we need a functional interface. In Java 8 we have some predefined functional interface which can eb leveraged directly, so that we don’t have to create custom functional interface.

IMPORTANT PRE-DEFINED FUNCTIONAL INTERFACES (defined **in java.util.function** package)

* Predicate
* Function
* Consumer
* Supplier

### PREDICATE

* Predicate is a pre-defined functional Interface

|  |  |  |
| --- | --- | --- |
| And | Default Method | Used for joining the Predicate |
| Or | Default Method |
| negate() | Default Method | Negating the Predicate |
| Boolean test(T t) | Abstract method | Evaluating a Boolean value after evaluating a Predicate |
| isEqual() | Static Method | To check equality of Object |

#### EXAMPLES – PREDICATE EXAMPLE

**public** **class** PredicateExamples {

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

PredicateExamples predicateExamples = **new** PredicateExamples();

String[] names = { "John", "Clark", "Stephen", "Ram" };

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

nameList.add("Krish");

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

/\* Predicate to check number is greater than 10 \*/

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

/\* Predicate to check String length greater than 5 \*/

Predicate<String> predicateStringLength = s -> s.length() > 5;

predicateExamples.testPredicate(names, predicateStringLength);

/\* Predicate to check String Start with K \*/

Predicate<String> predicateStartWithK = s -> s.startsWith("S");

predicateExamples.testPredicate(names, predicateStartWithK);

/\* Predicate to check collection is empty of not \*/

Predicate<Collection<String>> predicateEmptyCollection = c -> c.isEmpty();

System.***out***.println(predicateGreaterThanTen.test(10));

System.***out***.println(predicateGreaterThanTen.test(100));

System.***out***.println(predicateEmptyCollection.test(nameList) + " " + predicateEmptyCollection.test(emptyList));

}

**private** **void** testPredicate(String[] names, Predicate<String> predicate) {

**for** (String name : names) {

**if** (predicate.test(name))

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

}

}

}

#### EXAMPLES – PREDICATE JOIN

* join() and and() are default methods of Predicate Interface. This is used to join multiple predicates

**public** **class** PredicateJoinExample {

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

Integer[] testNumbers = { 10, 20, 45, 11, 87, 90, 1 };

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

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

PredicateJoinExample predicateJoinExample = **new** PredicateJoinExample();

System.***out***.println("Even number greater than 10");

predicateJoinExample.testJoinPredicate(testNumbers,

predicateGreaterThanTen.and(predicateEvenNumber)); 🡨 JOINING PREDICATE USING AND

System.***out***.println("Even number or greater than 10");

predicateJoinExample.testJoinPredicate(testNumbers,

predicateGreaterThanTen.or(predicateEvenNumber)); 🡨 JOINING PREDICATE USING OR

System.***out***.println("Not Even number and less than 10");

predicateJoinExample.testJoinPredicate(testNumbers,

predicateGreaterThanTen.or(predicateEvenNumber.negate()));🡨 NEGATE PREDICATE

}

**public** **void** testJoinPredicate(Integer[] testNumbers, Predicate<Integer> joinPredicate) {

**for** (Integer number : testNumbers) {

System.***out***.println(number + " " + joinPredicate.test(number));

}

}

}

#### EXAMPLES – PREDICATE EQUALS

* IsEquals method is used to check the equality of the of ob

**public** **class** StudentEqualsPredicate {

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

List<Student> studentList = **new** ArrayList<Student>();

studentList.add(**new** Student(1, "John"));

studentList.add(**new** Student(2, "Hopkin"));

studentList.add(**new** Student(3, "John"));

Predicate<Student> predicateToTest = Predicate.*isEqual*(**new** Student(1, "John"));

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

**if**(predicateToTest.test(student))

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

}

}

}

### FUNCTION

|  |  |  |
| --- | --- | --- |
| andThen() | Default Method |  |
| apply() | Abstract method |  |
| compose() | Default Method |  |
| Identity() | Static Method |  |

## METHOD AND CONSTRUCTOR REFERENCES

## STREAMS API

### ANALOGY OF STREAMS

### CONCEPTS OF STREAMS

1. A Stream pipeline is consisting of source, followed by zero or more intermediate operations and a terminal operation



* **SOURCE**: Streams can be created from Collections, List, Set , ints, long, double , arrays and lines of file

**Streams operations can be intermediate or terminal operation**

* **INTERMEDIATE OPERATIONS**
  + Multiple intermediate operation can be chained. Each intermediate operation return a stream itself that why they can be chained.
  + Example of intermediate operations are filter, map, sort.
* **TERMINAL OPERATIONS**
  + Examples of terminal operations foreach , collect or reduce – this returns either void or non -stream result.

#### INTERMEDIATE OPERATIONS

* Zero of more intermediate operations are allowed
* The order of intermediate operations matters especially in large dataset. **It better to filter first and then sort or map.**
* **Intermediate Operation includes**
  + anyMatch()
  + distinct()
  + filter()
  + findFirst()
  + flatmap()
  + map()
  + skip()
  + sorted()

#### TERMINAL OPERATIONS

* One terminal operation is allowed

|  |  |
| --- | --- |
| forEach() | Applies the same function to each element |
| Collect() | Saves the |
| Other terminal operation reduce the stream to a single summary element like count | |