

## What are default methods in java 8?

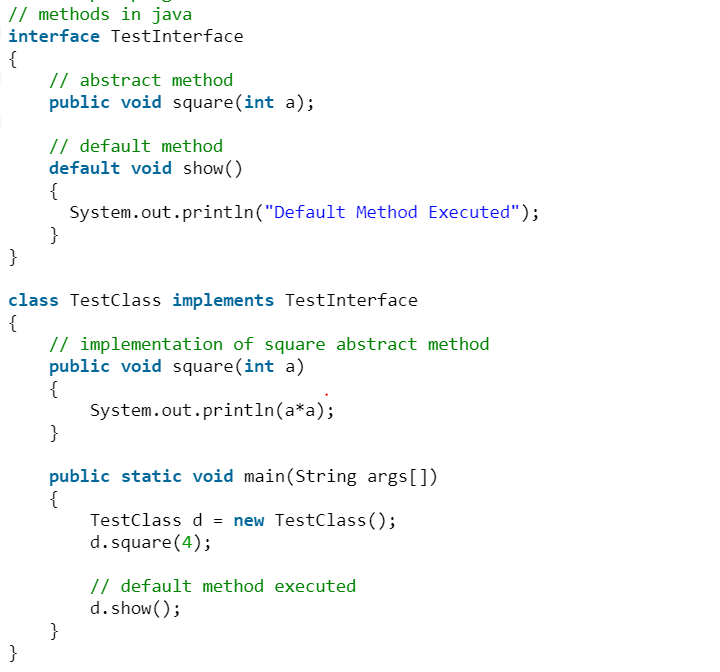
As name implies, default methods in java 8 are simply default. If you do not override them, they are the methods which will be invoked by caller classes. They are defined in interfaces.

Let’s understand with an example:

|  |
| --- |
| public interface Moveable {      default void move(){          System.out.println("I am moving");      }  } |

1. The method with the same signature in the “most specific default-providing interface” is selected. This means if class Animal implements two interfaces i.e. Moveable and Walkable such that Walkable extends Moveable. Then Walkable is here most specific interface and default method will be chosen from here if method signature is matched.
2. If Moveable and Walkable are independent interfaces then a serious conflict condition happen, and compiler will complain then it is unable to decide. The you have to help compiler by providing extra info that from which interface the default method should be called. e.g.

|  |
| --- |
| Walkable.super.move();  //or  Moveable.super.move(); |
|  |



## Java 8 – Functional Interfaces

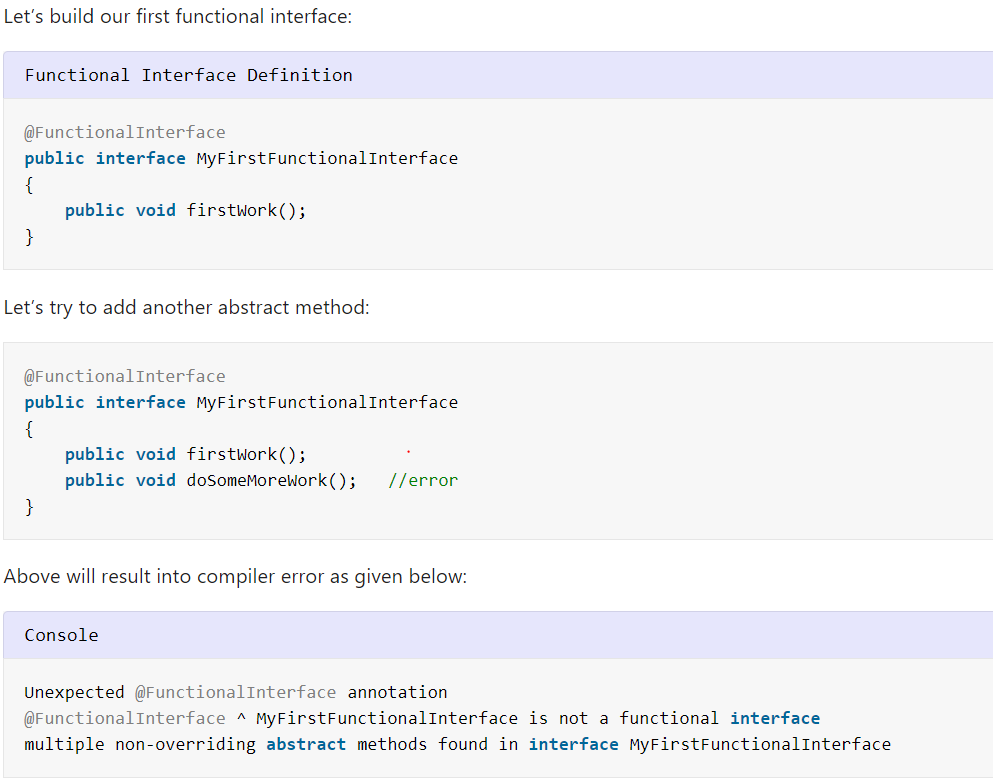
## 1. What is functional interface

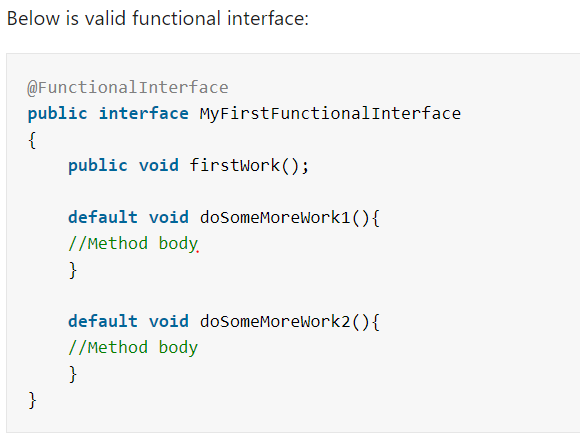
Functional interfaces are new additions in [**java 8**](https://howtodoinjava.com/category/java8/) which **permit exactly one abstract method inside them**. These interfaces are also called **Single Abstract Method interfaces (SAM Interfaces)**.

Functional interfaces are new concept introduced in Java 8. An interface with exactly one abstract method becomes Functional Interface. We don’t need to use @FunctionalInterface annotation to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. You can think of it like [@Override annotation](https://www.journaldev.com/817/java-override-method-overriding) and it’s best practice to use it. java.lang.Runnable with single abstract method run() is a great example of functional interface.

In Java 8, functional interfaces can be represented using lambda expressions, method reference and constructor references as well.

Java 8 introduces an annotation i.e. **@FunctionalInterface** too, which can be used for compiler level errors when the interface you have annotated violates the contracts of exactly one abstract method.

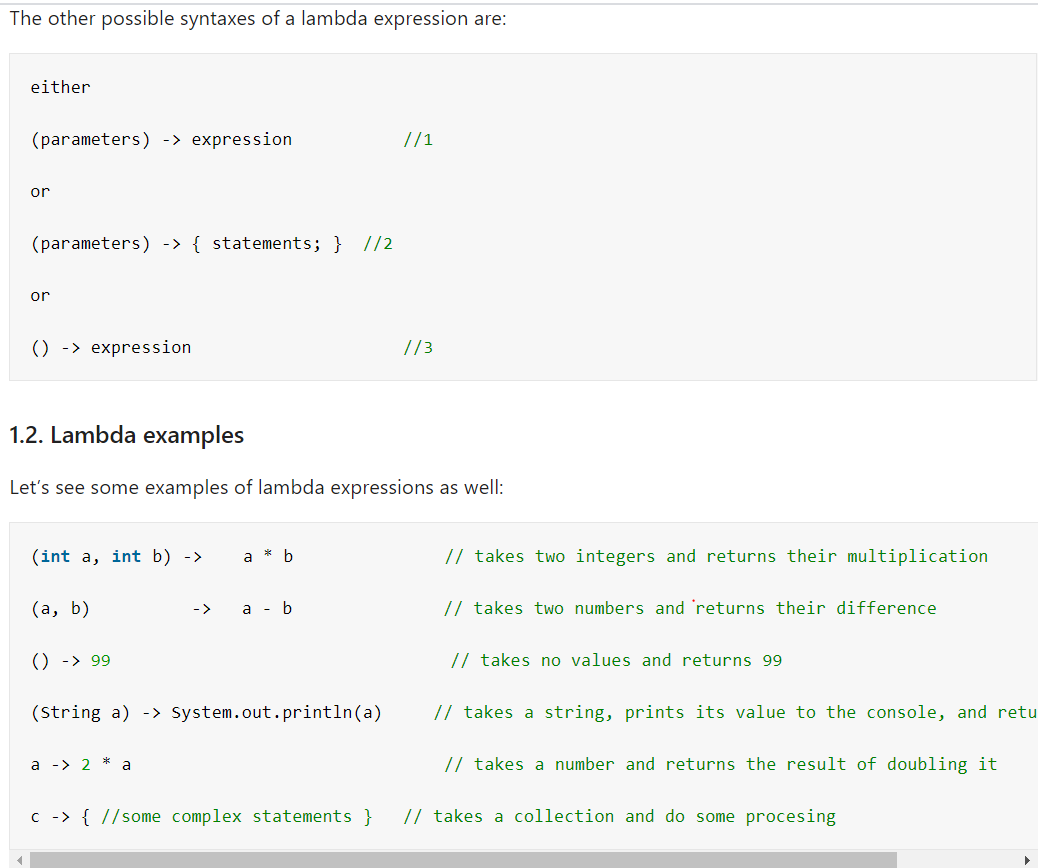




## 1. Lambda Expression

Lambda expressions are not unknown to many of us who have worked on other popular programming languages like Scala. In Java programming language, a Lambda expression (or function) is just an anonymous function, i.e., a function with no name and without being bounded to an identifier. They are written exactly in the place where it’s needed, typically as a parameter to some other function.

Yes, we can use **lambda** expression **only for functional interfaces**. Or you can say **interfaces** which has **only** single abstract method(SAM)

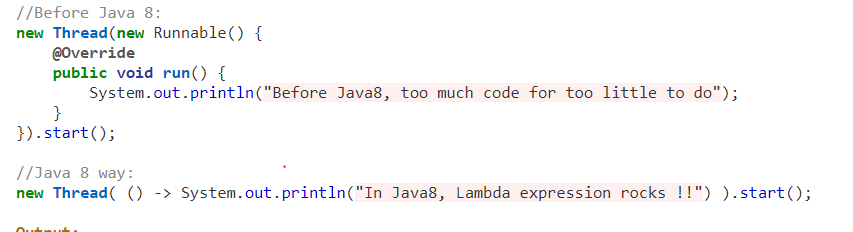


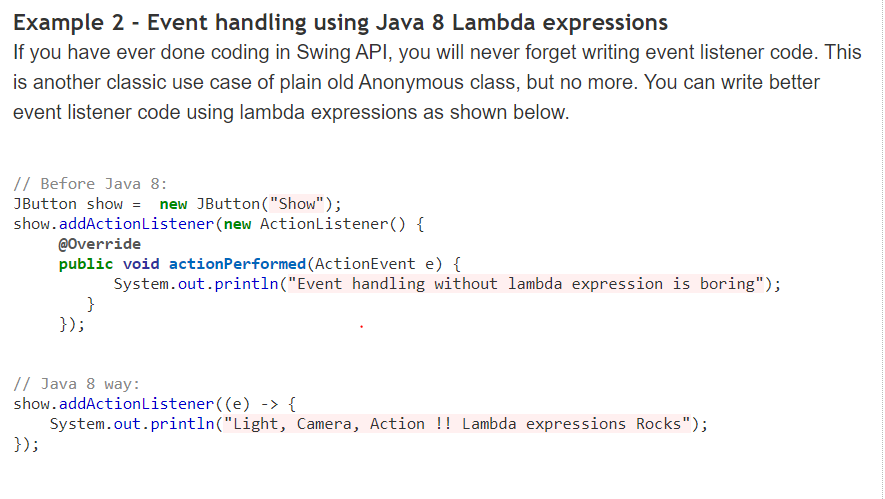
#### 1.3. Features of Lambda Expressions

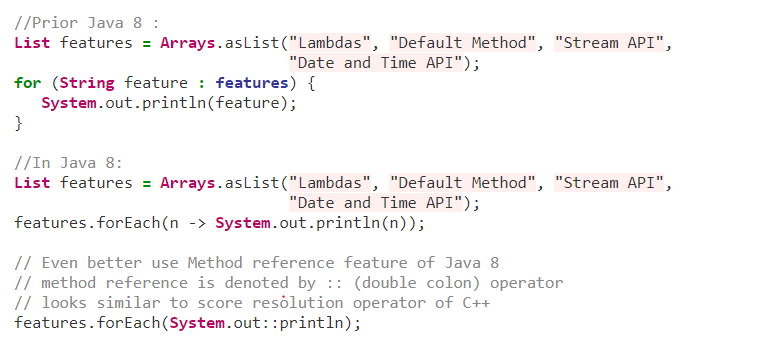
Let’s identify some **features of lambda expression**:

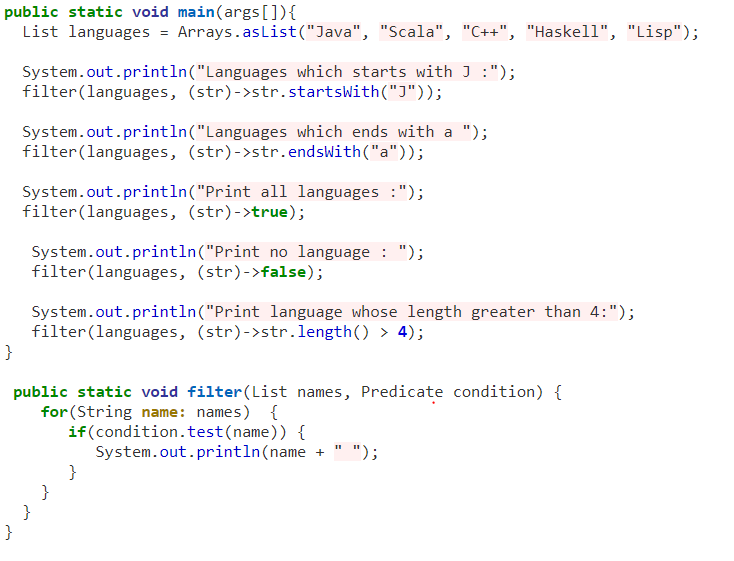
1. A lambda expression can have zero, one or more parameters.
2. The type of the parameters can be explicitly declared or it can be inferred from the context.
3. Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.
4. When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a\*a.
5. The body of the lambda expressions can contain zero, one or more statements.
6. If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.
7. In Java 8, Lambda expression is one of the biggest feature.
8. Lambdas are functions.
9. Lambdas can often replace an anonymous inner class.
10. A Lambda enables functions to be passed as an argument or return them as function return values.
11. It enables a new class of API's and programming styles (functional programming).
12. Comparator is a special kind of interface called a functional interface.
13. Functional interfaces are important because anywhere a functional interface is taken as an argument, it can be written as lambda expression.
14. Internal iteration is newly introduced in Java 8. For example list.forEach()
15. Method references are useful shorthand lambdas.
16. Lambda based code makes it easy to create fairly complicated structure by composition of smaller parts, resulting code is clear, concise and easy to modify.
17. Lambdas make code read more like the problem statement.

Example 1 - implementing Runnable using Lambda expression









## Java 8 method reference

In [Java 8](https://howtodoinjava.com/java-8-tutorial/), we can refer a method from class or object using **class::methodName** type syntax.

## Method reference to static method – Class::staticMethodName

## 2) Reference to an Instance Method

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

Syntax

containingObject::instanceMethodName

## 3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

ClassName::**new**

Look examples in code

## Stream In Java

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
   List number = Arrays.asList(2,3,4,5);  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

#### **Stream Creation**

Let’s first obtain a stream from an existing array:

private static Employee[] arrayOfEmps = {

new Employee(1, "Jeff Bezos", 100000.0),

new Employee(2, "Bill Gates", 200000.0),

new Employee(3, "Mark Zuckerberg", 300000.0)

};

Stream.of(arrayOfEmps);

We can also obtain a stream from an existing list:

private static List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream();

Note that **Java 8 added a new stream() method to the Collection interface.**

And we can create a stream from individual objects using Stream.of():

Stream.of(arrayOfEmps[0], arrayOfEmps[1], arrayOfEmps[2]);

Or simply using Stream.builder():

Stream.Builder<Employee> empStreamBuilder = Stream.builder();

empStreamBuilder.accept(arrayOfEmps[0]);

empStreamBuilder.accept(arrayOfEmps[1]);

empStreamBuilder.accept(arrayOfEmps[2]);

Stream<Employee> empStream = empStreamBuilder.build();

## Java forEach loop

Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get [ConcurrentModificationException](https://www.journaldev.com/378/java-util-concurrentmodificationexception) if iterator is not used properly.

Java 8 has introduced forEach method in java.lang.Iterable interface so that while writing code we focus on business logic only. forEach method takes java.util.function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse

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

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

