**Java 8 features**

1. [Java 8 – Lambda Expression](https://beginnersbook.com/2017/10/java-lambda-expressions-tutorial-with-examples/)  
   2. [Java 8 – Method references](https://beginnersbook.com/2017/10/method-references-in-java-8/)  
   3. [Java 8 – Functional interfaces](https://beginnersbook.com/2017/10/java-functional-interfaces/)  
   4. [Java 8 – Interface changes: Default and static methods](https://beginnersbook.com/2017/10/java-8-interface-changes-default-method-and-static-method/)  
   5. [Java 8 – Streams](https://beginnersbook.com/2017/10/java-8-stream-tutorial/)  
   6. [Java 8 – Stream filter](https://beginnersbook.com/2017/10/java-8-stream-filter/)  
   7. [Java 8 – forEach()](https://beginnersbook.com/2017/10/java-8-foreach/)  
   8. [Java 8 – Collectors class with example](https://beginnersbook.com/2017/10/java-8-stream-collectors-class-with-examples/)  
   9. [Java 8 – StringJoiner class with example](https://beginnersbook.com/2017/10/java-8-stringjoiner/)  
   10. [Java 8 – Optional class with example](https://beginnersbook.com/2017/10/java-8-optional-class/)  
   11. [Java 8 – Arrays Parallel Sort](https://beginnersbook.com/2017/10/java-8-arrays-parallel-sort-with-example/)

# 12. Predicate Filter

1. <https://beginnersbook.com/2017/10/java-8-features-with-examples/>
2. <https://howtodoinjava.com/java-8-tutorial/>
3. <https://www.java67.com/2014/09/top-10-java-8-tutorials-best-of-lot.html>
4. <https://stackoverflow.com/questions/14997202/creating-object-with-reference-to-interface>
5. <https://www.boraji.com/java-8-lambda-expression-examples>
6. <http://tutorials.jenkov.com/java/lambda-expressions.html>
7. [Java 8 – Lambda Expression](https://beginnersbook.com/2017/10/java-lambda-expressions-tutorial-with-examples/)

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.

## Java Lambda Expression Syntax

## Java lambda expressions are new in Java 8. Java lambda expressions are Java's first step into functional programming. A Java lambda expression is thus a function which can be created without belonging to any class. A Java lambda expression can be passed around as if it was an object and executed on demand.

## Java lambda expressions are commonly used to implement simple event listeners / callbacks, or in functional programming with the [Java Streams API](http://tutorials.jenkov.com/java-collections/streams.html).

## Java Lambdas and the Single Method Interface:

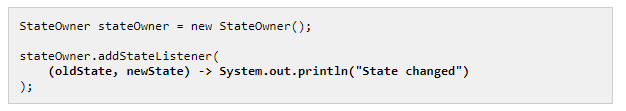
## Functional programming is very often used to implement event listeners. Event listeners in Java are often defined as Java interfaces with a single method. Here is a fictive single method interface example:

This Java interface defines a single method which is called whenever the state changes (in whatever is being observed).

In Java 7 you would have to implement this interface in order to listen for state changes. Imagine you have a class called StateOwner which can register state event listeners. Here is an example:

## In Java 7 you could add an event listener using an anonymous interface implementation, like this:

First a StateOwner instance is created. Then an anonymous implementation of the StateChangeListener interface is added as listener on the StateOwner instance.

In Java 8 you can add an event listener using a Java lambda expression, like this:

## The lambda expressions is this part:

The lambda expression is matched against the parameter type of the addStateListener() method's parameter. If the lambda expression matches the parameter type (in this case the StateChangeListener interface) , then the lambda expression is turned into a function that implements the same interface as that parameter.

Java lambda expressions can only be used where the type they are matched against is a single method interface. In the example above, a lambda expression is used as parameter where the parameter type was the StateChangeListener interface. This interface only has a single method. Thus, the lambda expression is matched successfully against that interface.

## Lambda Parameters

## Since Java lambda expressions are effectively just methods, lambda expressions can take parameters just like methods. The (oldState, newState) part of the lambda expression shown earlier specifies the parameters the lambda expression takes. These parameters have to match the parameters of the method on the single method interface. In this case, these parameters have to match the parameters of the onStateChange() method of the StateChangeListener interface:

As a minimum the number of parameters in the lambda expression and the method must match.

Second, if you have specified any parameter types in the lambda expression, these types must match too. I haven't shown you how to put types on lambda expression parameters yet (it is shown later in this text), but in many cases you don't need them.

### Zero Parameters:

If the method you are matching your lambda expression against takes no parameters, then you can write your lambda expression like this:

## Notice how the parentheses have no content in between. That is to signal that the lambda takes no parameters.

## One Parameter :

If the method you are matching your Java lambda expression against takes one parameter, you can write the lambda expression like this Notice the parameter is listed inside the parentheses.

When a lambda expression takes a single parameter, you can also omit the parentheses, like this:



### Multiple Parameters:

If the method you match your Java lambda expression against takes multiple parameters, the parameters need to be listed inside parentheses. Here is how that looks in Java code:



Only when the method takes a single parameter can the parentheses be omitted.

### Parameter Types:

Specifying parameter types for a lambda expression may sometimes be necessary if the compiler cannot infer the parameter types from the functional interface method the lambda is matching. Don't worry, the compiler will tell you when that is the case. Here is a Java lambda parameter type example:

(Car car) -> System.out.println("The car is: " + car.getName());

As you can see, the type (Car) of the car parameter is written in front of the parameter name itself, just like you would when declaring a parameter in a method elsewhere, or when making an anonymous implementation of an interface.

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.

//Syntax of lambda expression

(parameter\_list) -> {function\_body}

-> : This is arrow operator

The basic *syntax of a lambda expression* is:

|  |
| --- |
| either  (parameters) -> expression  or  (parameters) -> { statements; }  or  () -> expression |

**#** A typical lambda expression example will be like this:

|  |
| --- |
| (x, y) -> x + y  //This function takes two parameters and return their sum. |

#### ## Rules for writing lambda expressions

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.

## 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

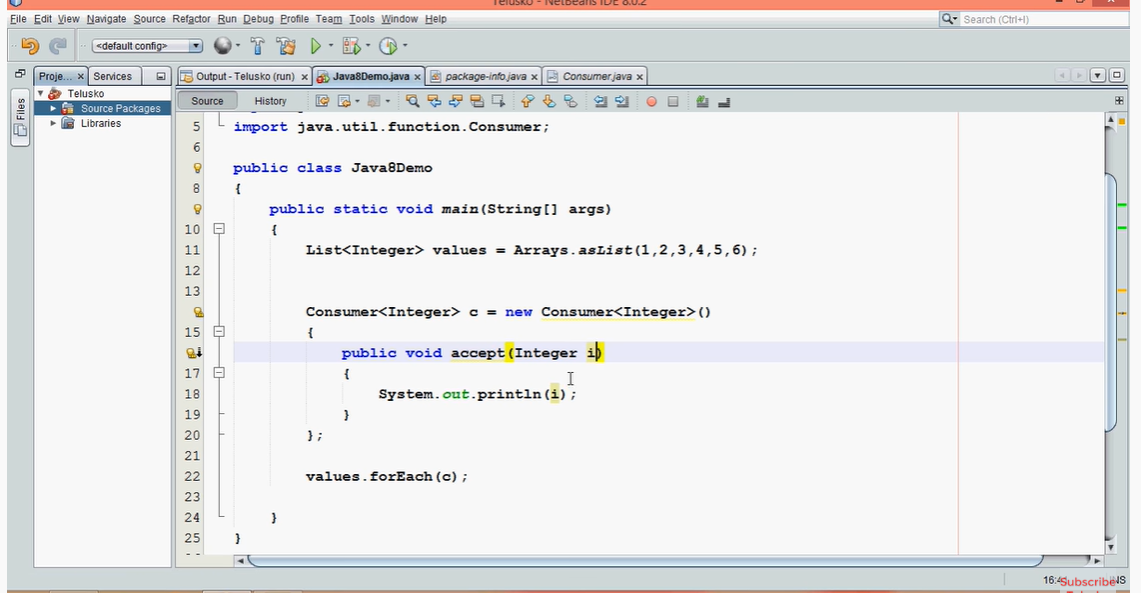
To use lambda expression, you need to either create your own functional interface or use the pre defined functional interface provided by Java. An interface with **only single abstract method** is called functional interface(or Single Abstract method interface), for example: Runnable, callable, ActionListener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

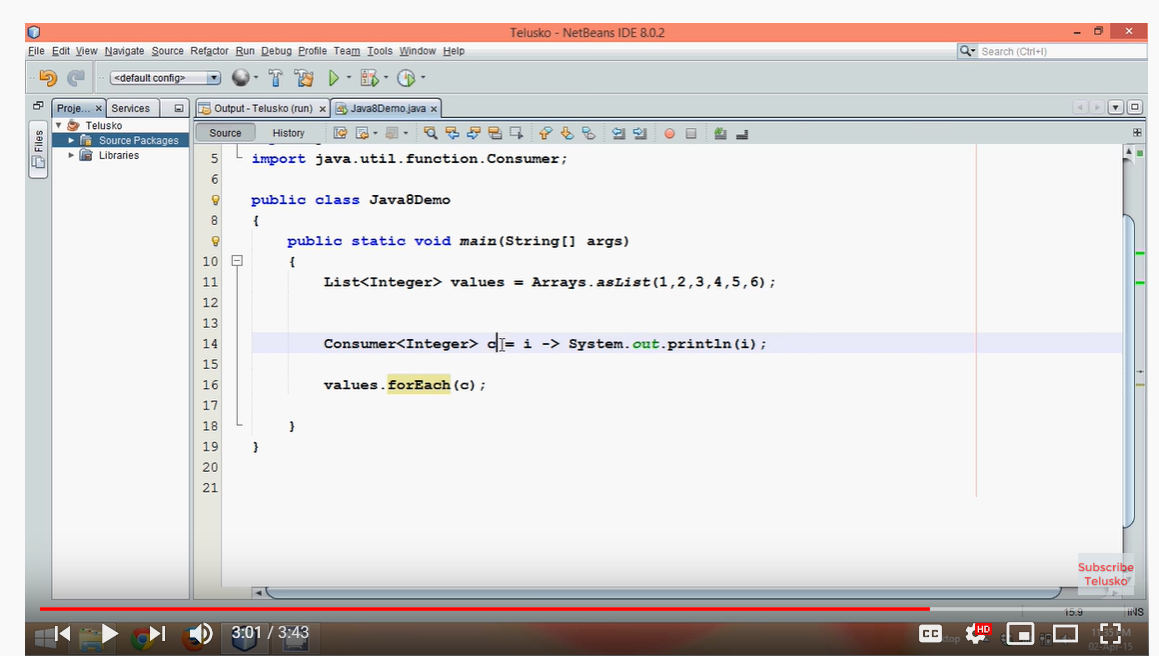
**To use function interface:**  
Pre Java 8: We create anonymous inner classes.  
Post Java 8: You can use lambda expression instead of anonymous inner classes.

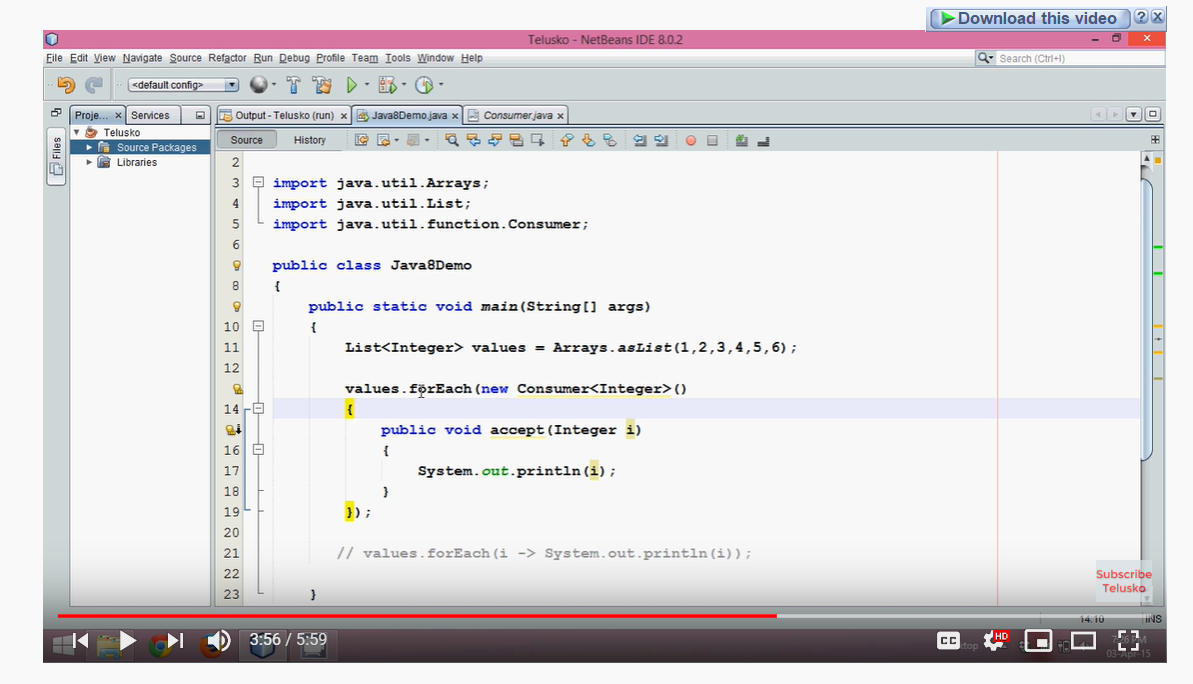
 a Lambda expression (or function) is just an anonymous function, i.e., a function with no name and without being bounded to an identifier. They are written exactly in the place where it’s needed, typically as a parameter to some other function.

We use lambda expression to reduce the boilerplate code.

StreamApi: it help to process our data.







## Functional Interface

functional interfaces are also called *Single Abstract Method interfaces (SAM Interfaces)*. As name suggest, they **permit exactly one abstract method** inside them. Java 8 introduces an annotation i.e. @FunctionalInterface which can be used for compiler level errors when the interface you have annotated violates the contracts of Functional Interface.

A typical functional interface example:

|  |
| --- |
| @FunctionalInterface  publicinterfaceMyFirstFunctionalInterface {      publicvoidfirstWork();  } |

Please note that a functional interface is valid even if the @FunctionalInterface annotation would be omitted. It is only for informing the compiler to enforce single abstract method inside interface.

Also, since default methods are not abstract you’re *free to add default methods* to your functional interface as many as you like.

Another important point to remember is that if an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere. for example, below is perfectly valid functional interface.

@FunctionalInterface

publicinterfaceMyFirstFunctionalInterface

{

    publicvoidfirstWork();

    @Override

    publicString toString();                //Overridden from Object class

    @Override

    publicbooleanequals(Object obj);        //Overridden from Object class

}

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)**.

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.

Let’s build our first functional interface:

|  |
| --- |
| Functional Interface Definition |
| @FunctionalInterface  publicinterfaceMyFirstFunctionalInterface  {      publicvoidfirstWork();  } |

Let’s try to add another abstract method:

|  |
| --- |
| @FunctionalInterface  publicinterfaceMyFirstFunctionalInterface  {      publicvoidfirstWork();      publicvoiddoSomeMoreWork();   //error  } |

Above will result into compiler error as given below:

|  |
| --- |
| Console |
| Unexpected @FunctionalInterfaceannotation  @FunctionalInterface^ MyFirstFunctionalInterface is not a functional interface  multiple non-overriding abstractmethods found in interfaceMyFirstFunctionalInterface  Below is list of things which are allowed and which are not in a functional interface.   * As discussed above, ***only one abstract method is allowed*** in any functional interface. Second abstract method is not permitted in a functional interface. If we remove **@FunctionInterface** annotation then we are allowed to add another abstract method, but it will make the interface non-functional interface. * A functional interface is ***valid even if the @FunctionalInterface annotation would be omitted***. It is only for informing the compiler to enforce single [abstract method](https://howtodoinjava.com/object-oriented/exploring-interfaces-and-abstract-classes-in-java/) inside interface. * Conceptually, a functional interface has exactly one abstract method. Since [**default methods**](https://howtodoinjava.com/java8/default-methods-in-java-8/) have an implementation, they are not abstract. Since default methods are not abstract you’re ***free to add default methods to your functional interface as many as you like***.   Below is valid functional interface:   |  | | --- | | @FunctionalInterface  publicinterfaceMyFirstFunctionalInterface  {      publicvoidfirstWork();      defaultvoiddoSomeMoreWork1(){      //Method body      }      defaultvoiddoSomeMoreWork2(){      //Method body      }  } |  * If an interface declares an ***abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count*** since any implementation of the interface will have an implementation from java.lang.Object or elsewhere. e.g. [**Comparator**](https://howtodoinjava.com/search-sort/when-to-use-comparable-and-comparator-interfaces-in-java/) is a functional interface even though it declared two abstract methods. Why? Because one of these abstract methods “equals()” which has signature equal to public method in Object class.   e.g. Below interface is a valid functional interface.   |  | | --- | | @FunctionalInterface  publicinterfaceMyFirstFunctionalInterface  {      publicvoidfirstWork();      @Override      publicString toString();                //Overridden from Object class      @Override      publicbooleanequals(Object obj);        //Overridden from Object class  } | |

# [**Creating object with reference to Interface**](https://stackoverflow.com/questions/14997202/creating-object-with-reference-to-interface)

# <https://stackoverflow.com/questions/14997202/creating-object-with-reference-to-interface>

<https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method>

<https://www.journaldev.com/12850/java-9-private-methods-interfaces>

A reference variable can be declared as a class type or an interface type.If the variable is declared as an interface type, it can reference any object of any class that implements the interface.

Based on the above statement I have made a code on understanding. As said above **declared as an interface type, it can reference any object of any class that implements the interface**.

But in my code is displaying displayName()method undefined at objParent.displayName();

publicclassOverridenClass

{

publicstaticvoid main(String[] args)

{

Pritable objParent = newParent();

objParent.sysout();

objParent.displayName();

}

}

interfacePritable

{

void sysout();

}

classParentimplementsPritable

{

publicvoid displayName()

{

System.out.println("This is Parent Name");

}

publicvoid sysout()

{

System.out.println("I am Printable Interfacein Parent Class");

}

}

Answer :

Right, because displayName is not defined in the Pritable interface. You can only access the methods defined on the interface through a variable declared as having that interface, even if the concrete class has additional methods. That's why you can call sysout, but not displayName.

classBar {

publicstaticvoid foo(Pritable p) {

p.sysout();

p.displayName();

}

}

classTest {

publicstaticfinalvoid main(String[] args) {

Bar.foo(newParent());

}

}

The code in foo **must not** rely on anything other than what is featured in the Pritable interface, as we have no idea at compile-time what the concrete class may be.

The point of interfaces is to define the characteristics that are available to the code using only an interface reference, without regard to the concrete class being used.

The displayName() method is displayed as undefined because objParent declared as type Pritable and the interface does not have such method. To be able to use method displayName(), you can declare it in interface Pritable:

interfacePritable

{

void sysout();

void displayName();

}

Or cast objParent to type Parent first before calling method displayName():

Pritable objParent = newParent();

objParent = (Parent)objParent;

objParent.displayName();

Interfaces are basically another way of - breaking the rules of single inheritance.

By using interfaces, a child class can, both inherit it's parents methods and be forced to implement it's interface methods. Resulting in an easy to extend and maintain inheritance tree etc.

The catch however is, when the child is referenced under the parent, you only have access to the parent methods. To access the interface methods, you will need to cast or create the child under the interface reference type.

Interfaces also allow the collection of multiple classes of different families to be collected under the interface type. To what benefit I am yet to discover.

In my opinion, it is pointless since I still cannot achieve fully blown polymorphism anyways - by just using the parent reference type and still have access to the interface implementations.

# Java 8 Interface Changes – default method and static method

Prior to java 8, [interface in java](https://beginnersbook.com/2013/05/java-interface/) can only have abstract methods. All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

## Why default method?

For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code.

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

## Java 8 Example: Default method in Interface

## Java 8 Example: Default method in Interface

The method newMethod() in MyInterface is a default method, which means we need not to implement this method in the implementation class Example. This way we can add the default methods to existing interfaces without bothering about the classes that implements these interfaces.

Interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

Default void newMethod(){

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

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

Public class Example implements MyInterface{

// implementing abstract method

publicvoid existingMethod(String str){

System.out.println("String is: "+str);

}

publicstaticvoid main(String[] args) {

Example obj = newExample();

//calling the default method of interface

obj.newMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

## Java 8 Example: Static method in Interface

As mentioned above, the static methods in interface are similar to default method so we need not to implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes.

Interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

Default void newMethod(){

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

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\* Similar to default methods, we need to implement these methods

\* in implementation classes so we can safely add them to the

\* existing interfaces.

\*/

Static void anotherNewMethod(){

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

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

publicclassExampleimplementsMyInterface{

// implementing abstract method

publicvoid existingMethod(String str){

System.out.println("String is: "+str);

}

publicstaticvoid main(String[] args) {

Example obj = newExample();

//calling the default method of interface

obj.newMethod();

//calling the static method of interface

MyInterface.anotherNewMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

Output:

Newly added default method

Newly added static method

Stringis: Java8is easy to learn

## Java 8 – Abstract classes vs interfaces

With the introduction of default methods in interfaces, it seems that the [abstract classes](https://beginnersbook.com/2013/05/java-abstract-class-method/) are same as interface in java 8. However this is not entirely true, even though we can now have concrete methods(methods with body) in interfaces just like abstract class, this doesn’t mean that they are same. There are still few differences between them, one of them is that abstract class can have constructor while in interfaces we can’t have constructors.

The purpose of interface is to provide full abstraction, while the purpose of abstract class is to provide partial abstraction. This still holds true. The interface is like a blueprint for your class, with the introduction of default methods you can simply say that we can add additional features in the interfaces without affecting the end user classes.

# Method References in Java 8.

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

Method reference is a shorthand notation of a lambda expression to call a method. For example:  
If your lambda expression is like this:

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

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

System.out::println

The :: operator is used in method reference to separate the class or object from the method name(we will learn this with the help of examples).

## 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

# **A Guide to Streams in Java 8: In-Depth Tutorial with Examples:**

### **Introduction**

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.

First of all, Java 8 Streams should not be confused with Java I/O streams (ex: *FileInputStream* etc); these have very little to do with each other.

Simply put, streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.

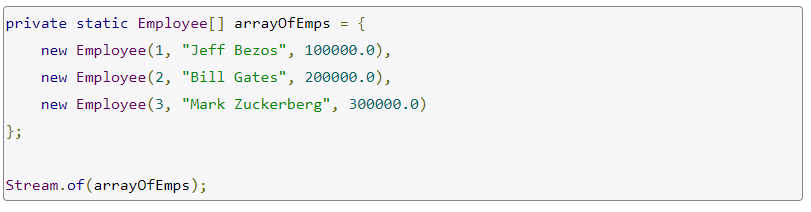
**A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.**

This new functionality – [*java.util.stream*](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html) – supports functional-style operations on streams of elements, such as map-reduce transformations on collections.

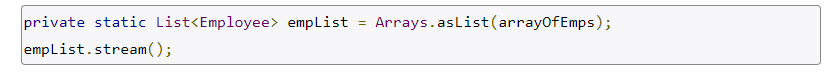
Let’s now dive into few simple examples of stream creation and usage – before getting into terminology and core concepts.

#### **Stream Creation**

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



We can also obtain a stream from an existing list:

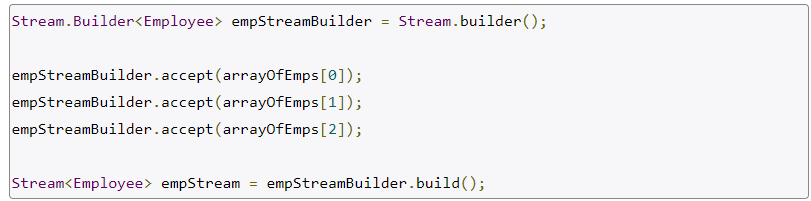


Note that **Java 8 added a new *stream()*method to the *Collection* interface.**

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



Or simply using Stream.builder():



### **Stream Operations**

Let’s now see some common usages and operations we can perform on and with the help of the new stream support in the language.

#### **forEach**

forEach() is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.

The method is so common that is has been introduced directly in Iterable, Map etc: