<https://shekhargulati.com/7-days-with-java-8/>

<https://www.javatpoint.com/java-8-features>

## Java 8 Programming Language Enhancements

Java 8 provides following features for Java Programming:

* Lambda expressions,
* Method references,
* Functional interfaces,
* Stream API,
* Default methods,
* Base64 Encode Decode,
* Static methods in interface,
* Optional class,
* Collectors class,
* ForEach() method,
* Parallel array sorting,
* Nashorn JavaScript Engine,
* Parallel Array Sorting,
* Type and Repating Annotations,
* IO Enhancements,
* Concurrency Enhancements,
* JDBC Enhancements etc.

<https://www.javatpoint.com/java-lambda-expressions>

## Lambda Expressions

Lambda expression provides a clear and concise way to implement **SAM interface(Single Abstract Method) by using an expression(one method interface using an expression.)** It is very useful in collection library in **which it helps to iterate, filter and extract data.**

The Lambda expression **is used to provide the implementation of an interface which has functional interface.** It saves a lot of code. In case of lambda expression, **we don't need to define the method again for providing the implementation. Here, we just write the implementation code.**

Java lambda expression **is treated as a function, so compiler does not create .class file.**

## Functional Interface

**Lambda expression provides implementation of *functional interface*.** An **interface which has only one abstract method is called functional interface.** Java **provides an anotation @*FunctionalInterface*, which is used to declare an interface as functional interface.**

## Java Lambda Expression Syntax

1. (argument-list) -> {body}

Java lambda expression is consisted of three components.

1) Argument-list: It can b**e empty or non-empty as well.**

2) Arrow-token: It is used to **link arguments-list and body of expression.**

3) Body: It contains **expressions and statements for lambda expression.**

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

<https://www.javatpoint.com/java-8-method-reference>

Method reference is used to refer method of functional interface.Each time **when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.**

## Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

All samples of static would be applied to instance method so those sample are not included

## 1) Reference to a Static Method

You can refer to static method defined in the class.

## 3) Reference to a Constructor

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

**Syntax**

1. **ClassName::new**

**https://www.javatpoint.com/java-8-functional-interfaces**

An Interface that contains exactly one abstract method is known as functional interface. It c**an have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.**

Functional Interface is also **known as Single Abstract Method Interfaces or SAM Interfaces.**

### Invalid Functional Interface

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

interface sayable{   
 void say(String msg); // abstract method   
}   
@FunctionalInterface   
interface Doable extends sayable{   
 // Invalid '@FunctionalInterface' annotation; Doable is not a functional interface   
 void doIt();   
}

### 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.

Only important ones listed

|  |  |
| --- | --- |
| **Interface** | **Description** |
| [BiConsumer<T,U>](https://www.javatpoint.com/java-biconsumer-interface) | It represents an operation that accepts two input arguments and returns no result. |
| [Consumer<T>](https://www.javatpoint.com/java-consumer-interface) | It represents an operation that accepts a single argument and returns no result. |
| [Function<T,R>](https://www.javatpoint.com/java-function-interface) | It represents a function that accepts one argument and returns a result. |
| [Predicate<T>](https://www.javatpoint.com/java-predicate-interface) | It represents a predicate (boolean-valued function) of one argument. |

<https://www.javatpoint.com/java-8-stream>

Java provides a new **additional package in Java 8 called java.util.stream.** This package consists of **classes, interfaces and enum to allows functional-style operations on the elements.** You can use stream by importing **java.util.stream package.**

Stream provides following features:

* Stream **does not store elements**. It simply **conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.**
* Stream is **functional in nature. Operations performed on a stream does not modify it's source.** For example, **filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.**
* Stream is **lazy and evaluates code only when required.**
* The elements of a **stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.**

You can use **stream to filter, collect, print, and convert from one data structure to other etc.** In the following examples, **we have apply various operations with the help of stream.**

Important Methods

https://www.javatpoint.com/java-predicate-interface

It is a **functional interface** which represents a **predicate (boolean-valued function) of one argument.** It is defined in the java.util.function package **and contains test() a functional method.**

### Java Predicate Interface Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| boolean test(T t) | It evaluates **this predicate on the given argument.** |
| default Predicate<T> and(Predicate<? super T> other) | It returns a **composed predicate that represents a short-circuiting logical AND of this predicate and another**. When evaluating the composed predicate, **if this predicate is false, then the other predicate is not evaluated.** |
| default Predicate<T> negate() | It returns a predicate that represents the logical negation of this predicate. |
| default Predicate<T> or(Predicate<? super T> other) | It returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. When evaluating the composed predicate, if this predicate is true, then the other predicate is not evaluated. |
| static <T> Predicate<T> isEqual(Object targetRef) | It returns a predicate **that tests if two arguments are equal according to Objects.equals(Object, Object).** |

https://www.javatpoint.com/java-consumer-interface

# Java Consumer Interface

It is a functional interface defined in java.util.function package. It contains an **abstract accept() and a default andThen() method.** It can be **used as the assignment target for a lambda expression or method reference.**

The Consumer Interface **accepts a single argument and does not return any result.**

|  |  |
| --- | --- |
| void accept(T t) | It performs this operation on the given argument.  //The only abstract method |
| default Consumer<T> andThen(Consumer<? super T> after) | It returns a **composed Consumer that performs, in sequence, this operation followed by the after operation**. If performing **either operation throws an exception, it is relayed to the caller of the composed operation.** If performing this **operation throws an exception, the after operation will not be performed.** |

<https://www.javatpoint.com/java-function-interface>

It is a functional interface. It is used to **refer method by specifying type of parameter. It returns a result back to the referred function.**

Functions **accept one argument and produce a result**. **Default methods can be used to chain multiple functions together (compose, andThen).**

|  |  |
| --- | --- |
| **Method** | **Description** |
| default <V> Function<T,V> **andThen**(Function<? super R,? extends V> after) | It returns a composed function that first applies this function to its input, and then applies the after function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function. |
| static <T> Function<T,T> identity() | It returns a **function that always returns its input argument.** |
| R **apply**(T t) | It applies this **function to the given argument.** |
| default <V> Function<V,R> **compose**(Function<? super V,? extends T> before) | It Returns a c**omposed function that first applies the before function to its input, and then applies this function to the result.** If **evaluation of either function throws an exception, it is relayed to the caller of the composed function.** |

<http://www.deadcoderising.com/2015-09-07-java-8-functional-composition-using-compose-and-andthen/>

As you can see, the difference between **compose and andThen is the order they execute the functions.** While the **compose function executes the caller last and the parameter first,** the **andThen executes the caller first and the parameter last.**

[**https://www.javatpoint.com/java-biconsumer-interface**](https://www.javatpoint.com/java-biconsumer-interface)

# Java BiConsumer Interface

BiConsumer Interface **accepts two input arguments and does not return any result.** This is the **two-arity specialization of Consumer interface.** It provides a functional method **accept(Object, Object) to perform custom operations.**

### Java BiConsumer Interface Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| void **accept**(T t, U u) | It performs **this operation on the given arguments.** |
| default BiConsumer<T,U> **andThen**(BiConsumer<? super T,? super U> after) | It returns a composed BiConsumer **that performs, in sequence, this operation followed by the after operation.** If performing **either operation throws an exception, it is relayed to the caller of the composed operation.** If performing this operation throws an exception, the after operation will not be performed. |

**https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/**

**Supplier can be used in all contexts where there is no input but an output is expected.**

**Only one method get() is defined**

**Supplier’s Function Descriptor is** () -> T **. This means that there is no input in the lambda definition and the return output is an object of type T.**

[**https://www.javatpoint.com/java-8-optional**](https://www.javatpoint.com/java-8-optional)

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

## Java Optional Class Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public **static** <T> Optional<T> **empty**() | It returns an **empty Optional object. No value is present for this Optional.** |
| public **static** <T> Optional<T> **of**(T value) | It returns an **Optional with the specified present non-null value.** |
| public **static** <T> Optional<T> **ofNullable**(T value) | It returns an **Optional describing the specified value, if non-null, otherwise returns an empty Optional.** |
| **public T get()** | **If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.** |
| public boolean **isPresent**() | **It returns true if there is a value present, otherwise false.** |
| public void **ifPresent**(**Consumer**<? super T> consumer) | **If a value is present, invoke the specified consumer with the value, otherwise do nothing.** |

There are other functions like **filter,map,flatMap,**

|  |  |
| --- | --- |
| public T **orElse**(T **other**) | It returns the **value if present, otherwise returns other.** |
| public T orElseGet(Supplier<? extends T> other) | It **returns the value if present, otherwise invoke other** and **return the result of that invocation.** |
| public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X extends Throwable | It **returns the contained value, if present, otherwise throw an exception to be created by the provided supplier**. |
| public boolean equals(Object obj) | Indicates whether **some other object is "equal to" this Optional or not**. The other object is considered equal if:   * It is **also an Optional** and; * Both **instances have no value present or**; * the **present values are "equal to" each other via equals().** |

Also has **hashcode and toString**

<http://winterbe.com/posts/2014/03/16/java-8-tutorial/>

Default Methods for Interfaces

Java 8 enables us to **add non-abstract method implementations to interfaces by utilizing the default keyword**. This feature is also known as **Extension Methods**

interface Formula {

double calculate(int a);

default double sqrt(int a) {

return Math.sqrt(a);

}

}

Besides the **abstract** method calculate the interface Formula also defines the **default** method sqrt. **Concrete classes only have to implement the abstract method calculate**. The default method sqrt can be used out of the box.

Formula formula = new Formula() {

@Override

public double calculate(int a) {

return sqrt(a \* 100);

}

};

formula.calculate(100); // 100.0

formula.sqrt(16); // 4.0

The formula **is implemented as an anonymous object.** The code is **quite verbose: 6 lines of** code for such a simple calucation of sqrt(a \* 100)

Lambda expressions

Let's start with a simple example of how to sort a list of strings in prior versions of Java:

List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {

@Override

public int compare(String a, String b) {

return b.compareTo(a);

}

});

The static utility method **Collections.sort accepts a list and a comparator in order to sort the elements of the given list.**

Instead of creating anonymous objects all day long, Java 8 comes with a much shorter syntax, **lambda expressions**:

Collections.sort(names, (String a, String b) -> {

return b.compareTo(a);

});

As you can see the code is much shorter and easier to read. But it gets even shorter:

Collections.sort(names, (String a, String b) -> b.compareTo(a));

For **one line method bodies you can skip both the braces {} and the return keyword**. But it gets even more shorter:

Collections.sort(names, (a, b) -> b.compareTo(a));

**The java compiler is aware of the parameter types so you can skip them as well.** Let's dive deeper into how lambda expressions can be used in the wild.

### Functional Interfaces

How does lambda expressions fit into Javas type system? Each lambda corresponds to a given type, specified by an **interface**. A so **called *functional interface* must contain**exactly **one abstract method declaration**. Each **lambda expression of that type will be matched to this abstract method**. Since **default methods are not abstract you're free to add default methods to your functional interface.**

We can use arbitrary interfaces as lambda expressions **as long as the interface only contains one abstract method.** To ensure that your interface meet the requirements, you should add the **@FunctionalInterface annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract method declaration to the interface**.

Example:

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

Converter<String, Integer> converter = (from) -> Integer.valueOf(from);

Integer converted = converter.convert("123");

System.out.println(converted); // 123

Keep in mind that **the code is also valid if the @FunctionalInterface annotation would be ommited.**

### Method and Constructor References

The above example code can be further simplified by **utilizing static method references:**

Converter<String, Integer> converter = Integer::valueOf;

Integer converted = converter.convert("123");

System.out.println(converted); // 123

Java **8 enables you to pass references of methods or constructors via the :: keyword**. The above example shows **how to reference a static method. But we can also reference object methods:**

class Something {

String startsWith(String s) {

return String.valueOf(s.charAt(0));

}

}

Something something = new Something();

Converter<String, String> converter = something::startsWith;

String converted = converter.convert("Java");

System.out.println(converted); // "J"

Let's see how the :: keyword works for constructors. First we define an example bean with different constructors:

class Person {

String firstName;

String lastName;

Person() {}

Person(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

}

Next we specify a person factory interface to be used for creating new persons:

interface PersonFactory<P extends Person> {

P create(String firstName, String lastName);

}

Instead of implementing the factory manually, we glue everything together via constructor references:

PersonFactory<Person> personFactory = Person::new;

Person person = personFactory.create("Peter", "Parker");

We create a reference to the Person constructor via Person::new. The Java compiler automatically chooses the right constructor by matching the signature of PersonFactory.create.

### Lambda Scopes

Accessing outer scope variables from lambda expressions is very similar to anonymous objects. You can access **final variables from the local outer scope as well as instance fields and static variables.**

#### **Accessing local variables**

We can read **final local variables from the outer scope of lambda expressions:**

final int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3

But different to anonymous objects the variable num does not have to be declared final. This code is also valid:

int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3

However num must be **implicitly final** for the code to compile. The following code does **not** compile:

int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

num = 3;

**Writing to num from within the lambda expression is also prohibited.**

#### **Accessing fields and static variables**

In constrast **to local variables we have both read and write access to instance fields and static variables from within lambda expressions**. This behaviour is well known from anonymous objects.

class Lambda4 {

static int outerStaticNum;

int outerNum;

void testScopes() {

Converter<Integer, String> stringConverter1 = (from) -> {

outerNum = 23;

return String.valueOf(from);

};

Converter<Integer, String> stringConverter2 = (from) -> {

outerStaticNum = 72;

return String.valueOf(from);

};

}

}

#### **Accessing Default Interface Methods**

Remember the formula example from the first section? Interface Formula defines a default method sqrt which can be accessed from each formula instance including anonymous objects. This does not work with lambda expressions.

**Default methods**cannot**be accessed from within lambda expressions**. The following code does not compile:

Formula formula = (a) -> sqrt( a \* 100);

### Built-in Functional Interfaces

The JDK 1.8 API contains many built-in functional interfaces. Some of them are well known from older versions of **Java like Comparator or Runnable**. Those existing interfaces are **extended to enable Lambda support via the @FunctionalInterface annotation.**

#### **Predicates**

**Predicates are boolean-valued functions of one argument.** The interface contains various **default methods for composing predicates to complex logical terms (and, or, negate)**

Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo"); // true

predicate.negate().test("foo"); // false

Predicate<Boolean> nonNull = Objects::nonNull;

Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;

Predicate<String> isNotEmpty = isEmpty.negate();

#### **Functions**

Functions **accept one argument** and produce a result. **Default methods can be used to chain multiple functions together (compose, andThen).**

Function<String, Integer> toInteger = Integer::valueOf;

Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"

#### **Suppliers**

Suppliers **produce a result of a given generic type**. Unlike **Functions, Suppliers don't accept arguments.**

Supplier<Person> personSupplier = Person::new;

personSupplier.get(); // new Person

Consumers

Consumers represents **operations to be performed on a single input argument.**

Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);

greeter.accept(new Person("Luke", "Skywalker"));

Comparators

Comparators are well known from older versions of Java. Java 8 adds various default methods to the interface.

Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");

Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2); // > 0

comparator.reversed().compare(p1, p2); // < 0

**Optionals**

Optionals are **not functional interfaces**, instead it's a nifty utility to **prevent NullPointerException** Optional is a simple **container for a value which may be null or non-null**. Think of a method **which may return a non-null result but sometimes return nothing**. Instead of **returning null you return an Optional in Java 8.**

Optional<String> optional = Optional.of("bam");

optional.isPresent(); // true

optional.get(); // "bam"

optional.orElse("fallback"); // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0))); // "b"

Streams

A java.util.Stream **represents a sequence of elements on which one or more operations can be performed**. Stream operations are **either *intermediate* or *terminal***. While **terminal operations return a result of a certain type, intermediate operations return the stream itself so you can chain multiple method calls in a row. Streams are created on a source**, e.g. a java.util.Collection like lists or sets (**maps are not supported**). Stream operations **can either be executed sequential or parallel.**

Let's first look how sequential streams work. First we create a sample source in form of a list of strings:

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

stringCollection.add("ddd2");

stringCollection.add("aaa2");

stringCollection.add("bbb1");

stringCollection.add("aaa1");

stringCollection.add("bbb3");

stringCollection.add("ccc");

stringCollection.add("bbb2");

stringCollection.add("ddd1");

Collections in Java 8 are extended so you can simply **create streams either by calling Collection.stream() or Collection.parallelStream().** The following sections explain the most common stream operations.

Filter

Filter accepts a predicate to filter all elements of the stream. This operation is ***intermediate* which enables us to call another stream operation (forEach) on the result**. ForEach accepts a consumer to be executed for each element in the filtered stream**. ForEach is a terminal operation**. **It's void, so we cannot call another stream operation.**

stringCollection

.stream()

.filter((s) -> s.startsWith("a"))

.forEach(System.out::println);

// "aaa2", "aaa1"

Sorted

Sorted is an *intermediate* operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom Comparator.

stringCollection

.stream()

.sorted()

.filter((s) -> s.startsWith("a"))

.forEach(System.out::println);

// "aaa1", "aaa2"

Keep in mind that sorted **does only create a sorted view of the stream** **without manipulating the ordering of the backed collection. The ordering of stringCollection is untouched:**

System.out.println(stringCollection);

// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

Map

The *intermediate* **operation map converts each element into another object via the given function,** you can also **use mapto transform each object into another type. The generic type of the resulting stream depends on the generic type of the function you pass to map.**

stringCollection

.stream()

.map(String::toUpperCase)

.sorted((a, b) -> b.compareTo(a))

.forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"

Match

Various matching operations can be used **to check whether a certain predicate matches the stream**. All of those operations are *terminal* and return a boolean result.

boolean anyStartsWithA =

stringCollection

.stream()

.anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA); // true

boolean allStartsWithA =

stringCollection

.stream()

.allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA); // false

boolean noneStartsWithZ =

stringCollection

.stream()

.noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ); // true

Count

Count is a *terminal* operation returning the number of elements in the stream as a **long**.

long startsWithB =

stringCollection

.stream()

.filter((s) -> s.startsWith("b"))

.count();

System.out.println(startsWithB); // 3

Reduce

This ***terminal* operation performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.**

Optional<String> reduced =

stringCollection

.stream()

.sorted()

.reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);

// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"

Parallel Streams

As mentioned **above streams can be either sequential or parallel**. Operations on **sequential streams are performed on a single thread while operations on parallel streams are performed concurrent on multiple threads**.

The following example demonstrates **how easy it is to increase the performance by using parallel streams**.

First we create a large list of unique elements:

int max = 1000000;

List<String> values = new ArrayList<>(max);

for (int i = 0; i < max; i++) {

UUID uuid = UUID.randomUUID();

values.add(uuid.toString());

}

Now we measure the time it takes to sort a stream of this collection.

Sequential Sort

long t0 = System.nanoTime();

long count = values.stream().sorted().count();

System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);

System.out.println(String.format("sequential sort took: %d ms", millis));

// sequential sort took: 899 ms

Parallel Sort

long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();

System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);

System.out.println(String.format("parallel sort took: %d ms", millis));

// parallel sort took: 472 ms

As you can see both code snippets are almost identical but the parallel sort is roughly 50% faster. All you have to do is change stream() to parallelStream().

Map

As already **mentioned maps don't support streams. Instead maps now support various new and useful methods for doing common tasks.**

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

for (int i = 0; i < 10; i++) {

map.putIfAbsent(i, "val" + i);

}

map.forEach((id, val) -> System.out.println(val));

The above code should be self-explaining: putIfAbsent prevents us from writing additional if null checks; forEach accepts a consumer to perform operations for each value of the map.

This example shows how to compute code on the map by utilizing functions:

map.computeIfPresent(3, (num, val) -> val + num);

map.get(3); // val33

map.computeIfPresent(9, (num, val) -> null);

map.containsKey(9); // false

map.computeIfAbsent(23, num -> "val" + num);

map.containsKey(23); // true

map.computeIfAbsent(3, num -> "bam");

map.get(3); // val33

Next, we learn how to remove entries for a a given key, only if it's currently mapped to a given value:

map.remove(3, "val3");

map.get(3); // val33

map.remove(3, "val33");

map.get(3); // null

Another helpful method:

map.getOrDefault(42, "not found"); // not found

Merging entries of a map is quite easy:

map.merge(9, "val9", (value, newValue) -> value.concat(newValue));

map.get(9); // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));

map.get(9); // val9concat

Merge either put the key/value into the map if no entry for the key exists, or the merging function will be called to change the existing value.

***UPDATE****- I'm currently working on a JavaScript implementation of the Java 8 Streams API for the browser. If I've drawn your interest check out*[*Stream.js on GitHub*](https://github.com/winterbe/streamjs)*. Your Feedback is highly appreciated.*