

Java SE 8 New Features

Study Notes



December 1, 2015

Vladimir Virlan

Contents

[Introduction 2](#_Toc436921856)

[Java Programming Language 2](#_Toc436921857)

[Lambda Expressions 2](#_Toc436921858)

[Functional Interfaces 3](#_Toc436921859)

[Predicate<T > predicate 6](#_Toc436921860)

[Consumer<T> block 6](#_Toc436921861)

[Function<T,R> 7](#_Toc436921862)

[Supplier<T> 8](#_Toc436921863)

[Custom Functional Interfaces 8](#_Toc436921864)

[Method references 9](#_Toc436921865)

[Reference to a Static Method 9](#_Toc436921866)

[Reference to an Instance Method of a Particular Object 9](#_Toc436921867)

[Reference to an Instance Method of an Arbitrary Object of a Particular Type 10](#_Toc436921868)

[Reference to a Constructor 10](#_Toc436921869)

[Default methods and statics in interfaces 10](#_Toc436921870)

[Repeating Annotations 13](#_Toc436921871)

[Type Annotations 13](#_Toc436921872)

[Improved type inference 13](#_Toc436921873)

[Method parameter reflection 13](#_Toc436921874)

[java.lang and java.util Packages 13](#_Toc436921875)

[Standard Encoding and Decoding Base64 13](#_Toc436921876)

[Unsigned Arithmetic Support 18](#_Toc436921877)

[Stream API 18](#_Toc436921878)

[Aggregate Operations 18](#_Toc436921879)

[Reduction 18](#_Toc436921880)

[Parallelism 18](#_Toc436921881)

[Date Time API 19](#_Toc436921882)

[Optional 19](#_Toc436921883)

[Concurrency 19](#_Toc436921884)

[Tools 19](#_Toc436921885)

[jdeps 19](#_Toc436921886)

[Nashorn, JavaScript Engine 19](#_Toc436921887)

# Introduction

This document contains my study notes for the new features in Java SE 8.

# Java Programming Language

## Lambda Expressions

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

Here is an example of some lambda expressions:

ArithmeticOperation addition = (**int** a, **int** b) -> a + b;

ArithmeticOperation subtraction = (**int** a, **int** b) -> a - b;

ArithmeticOperation multiplication = (**int** a, **int** b) -> a \* b;

ArithmeticOperation division = (**int** a, **int** b) -> a / b;

System.***out***.println("10 + 10 = "+operate(10, 10, addition));

System.***out***.println("10 - 10 = "+operate(10, 10, subtraction));

System.***out***.println("10 \* 10 = "+operate(10, 10, multiplication));

System.***out***.println("10 / 10 = "+operate(10, 10, division));

This is the operate method:

**private** **int** operate(**int** a, **int** b, ArithmeticOperation operation) {

**return** operation.operation(a, b);

}

And here is the ArithmeticOperation:

**public** **interface** ArithmeticOperation {

**int** operation(**int** a, **int** b);

}

### Functional Interfaces

#### Package java.util.function

*Functional interfaces* provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the *functional method* for that functional interface, to which the lambda expression's parameter and return types are matched or adapted. Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context:

// Assignment context

Predicate<String> p = String::isEmpty;

// Method invocation context

stream.filter(e -> e.getSize() > 10)...

// Cast context

stream.map((ToIntFunction) e -> e.getSize())...

The interfaces in this package are general purpose functional interfaces used by the JDK, and are available to be used by user code as well. While they do not identify a complete set of function shapes to which lambda expressions might be adapted, they provide enough to cover common requirements. Other functional interfaces provided for specific purposes, such as [FileFilter](https://docs.oracle.com/javase/8/docs/api/java/io/FileFilter.html), are defined in the packages where they are used.

The interfaces in this package are annotated with [FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html). This annotation is not a requirement for the compiler to recognize an interface as a functional interface, but merely an aid to capture design intent and enlist the help of the compiler in identifying accidental violations of design intent.

Functional interfaces often represent abstract concepts like functions, actions, or predicates. In documenting functional interfaces, or referring to variables typed as functional interfaces, it is common to refer directly to those abstract concepts, for example using "this function" instead of "the function represented by this object". When an API method is said to accept or return a functional interface in this manner, such as "applies the provided function to...", this is understood to mean a *non-null* reference to an object implementing the appropriate functional interface, unless potential nullity is explicitly specified.

The functional interfaces in this package follow an extensible naming convention, as follows:

* There are several basic function shapes, including [Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) (unary function from T to R), [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) (unary function from T to void), [Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) (unary function from T to boolean), and [Supplier](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) (nilary function to R).
* Function shapes have a natural arity based on how they are most commonly used. The basic shapes can be modified by an arity prefix to indicate a different arity, such as [BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html) (binary function from T and U to R).
* There are additional derived function shapes which extend the basic function shapes, including [UnaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html) (extends Function) and [BinaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html) (extends BiFunction).
* Type parameters of functional interfaces can be specialized to primitives with additional type prefixes. To specialize the return type for a type that has both generic return type and generic arguments, we prefix ToXxx, as in [ToIntFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntFunction.html). Otherwise, type arguments are specialized left-to-right, as in [DoubleConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleConsumer.html) or [ObjIntConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html). (The type prefix Obj is used to indicate that we don't want to specialize this parameter, but want to move on to the next parameter, as in [ObjIntConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html).) These schemes can be combined, as in IntToDoubleFunction.
* If there are specialization prefixes for all arguments, the arity prefix may be left out (as in [ObjIntConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html)).

|  |  |
| --- | --- |
| **Interface Summary** | |
| **Interface** | **Description** |
| [**BiConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html)<T,U> | Represents an operation that accepts two input arguments and returns no result. |
| [**BiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<T,U,R> | Represents a function that accepts two arguments and produces a result. |
| [**BinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html)<T> | Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| [**BiPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html)<T,U> | Represents a predicate (boolean-valued function) of two arguments. |
| [**BooleanSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BooleanSupplier.html) | Represents a supplier of boolean-valued results. |
| [**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<T> | Represents an operation that accepts a single input argument and returns no result. |
| [**DoubleBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleBinaryOperator.html) | Represents an operation upon two double-valued operands and producing a double-valued result. |
| [**DoubleConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleConsumer.html) | Represents an operation that accepts a single double-valued argument and returns no result. |
| [**DoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleFunction.html)<R> | Represents a function that accepts a double-valued argument and produces a result. |
| [**DoublePredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoublePredicate.html) | Represents a predicate (boolean-valued function) of one double-valued argument. |
| [**DoubleSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleSupplier.html) | Represents a supplier of double-valued results. |
| [**DoubleToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleToIntFunction.html) | Represents a function that accepts a double-valued argument and produces an int-valued result. |
| [**DoubleToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleToLongFunction.html) | Represents a function that accepts a double-valued argument and produces a long-valued result. |
| [**DoubleUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleUnaryOperator.html) | Represents an operation on a single double-valued operand that produces a double-valued result. |
| [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<T,R> | Represents a function that accepts one argument and produces a result. |
| [**IntBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntBinaryOperator.html) | Represents an operation upon two int-valued operands and producing an int-valued result. |
| [**IntConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntConsumer.html) | Represents an operation that accepts a single int-valued argument and returns no result. |
| [**IntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntFunction.html)<R> | Represents a function that accepts an int-valued argument and produces a result. |
| [**IntPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntPredicate.html) | Represents a predicate (boolean-valued function) of one int-valued argument. |
| [**IntSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntSupplier.html) | Represents a supplier of int-valued results. |
| [**IntToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntToDoubleFunction.html) | Represents a function that accepts an int-valued argument and produces a double-valued result. |
| [**IntToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntToLongFunction.html) | Represents a function that accepts an int-valued argument and produces a long-valued result. |
| [**IntUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntUnaryOperator.html) | Represents an operation on a single int-valued operand that produces an int-valued result. |
| [**LongBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongBinaryOperator.html) | Represents an operation upon two long-valued operands and producing a long-valued result. |
| [**LongConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongConsumer.html) | Represents an operation that accepts a single long-valued argument and returns no result. |
| [**LongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongFunction.html)<R> | Represents a function that accepts a long-valued argument and produces a result. |
| [**LongPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongPredicate.html) | Represents a predicate (boolean-valued function) of one long-valued argument. |
| [**LongSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongSupplier.html) | Represents a supplier of long-valued results. |
| [**LongToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongToDoubleFunction.html) | Represents a function that accepts a long-valued argument and produces a double-valued result. |
| [**LongToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongToIntFunction.html) | Represents a function that accepts a long-valued argument and produces an int-valued result. |
| [**LongUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongUnaryOperator.html) | Represents an operation on a single long-valued operand that produces a long-valued result. |
| [**ObjDoubleConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjDoubleConsumer.html)<T> | Represents an operation that accepts an object-valued and a double-valued argument, and returns no result. |
| [**ObjIntConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html)<T> | Represents an operation that accepts an object-valued and a int-valued argument, and returns no result. |
| [**ObjLongConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjLongConsumer.html)<T> | Represents an operation that accepts an object-valued and a long-valued argument, and returns no result. |
| [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<T> | Represents a predicate (boolean-valued function) of one argument. |
| [**Supplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html)<T> | Represents a supplier of results. |
| [**ToDoubleBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces a double-valued result. |
| [**ToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleFunction.html)<T> | Represents a function that produces a double-valued result. |
| [**ToIntBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces an int-valued result. |
| [**ToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntFunction.html)<T> | Represents a function that produces an int-valued result. |
| [**ToLongBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToLongBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces a long-valued result. |
| [**ToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToLongFunction.html)<T> | Represents a function that produces a long-valued result. |
| [**UnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html)<T> | Represents an operation on a single operand that produces a result of the same type as its operand. |

### Predicate<T > predicate

Represents a predicate (boolean-valued function) of one argument.This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [test(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**and**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#and-java.util.function.Predicate-)([**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> other)  Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. |
| static <T> [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<T> | [**isEqual**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#isEqual-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) targetRef)  Returns a predicate that tests if two arguments are equal according to [**Objects.equals(Object, Object)**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#equals-java.lang.Object-java.lang.Object-). |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**negate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#negate--)()  Returns a predicate that represents the logical negation of this predicate. |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**or**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#or-java.util.function.Predicate-)([**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> other)  Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. |
| boolean | [**test**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) t)  Evaluates this predicate on the given argument. |

Example:

Predicate<String> p = String::isEmpty;

System.***out***.println(p.test("aha"));

Will print: false

### Consumer<T> block

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects. This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [accept(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| void | [**accept**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t)  Performs this operation on the given argument. |
| default [**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> | [**andThen**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#andThen-java.util.function.Consumer-)([**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> after)  Returns a composed Consumer that performs, in sequence, this operation followed by the after operation. |

Example:

Consumer<String> consumer = (x) -> System.***out***.println(x.toUpperCase());

consumer.accept("Hello World!");

Will print: HELLO WORLD!

### Function<T,R>

Represents a function that accepts one argument and produces a result. This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [apply(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#apply-T-).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| default <V> [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html),V> | [**andThen**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#andThen-java.util.function.Function-)([**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html),? extends V> after)  Returns a composed function that first applies this function to its input, and then applies the after function to the result. |
| [**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) | [**apply**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#apply-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) t)  Applies this function to the given argument. |
| default <V> [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<V,[**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)> | [**compose**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#compose-java.util.function.Function-)([**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super V,? extends [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)> before)  Returns a composed function that first applies the before function to its input, and then applies this function to the result. |
| static <T> [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<T,T> | [**identity**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#identity--)()  Returns a function that always returns its input argument. |

Example:

Function<String,Integer> function = (f) -> f.length();

System.***out***.println(function.apply("Hello There!"));

Will print: 12

### Supplier<T>

Represents a supplier of results. There is no requirement that a new or distinct result be returned each time the supplier is invoked. This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [get()](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html#get--).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) | [**get**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html#get--)()  Gets a result. |

Example 1:

Random r = **new** Random();

Supplier<Integer> intSupplier = () -> r.nextInt();

System.***out***.println(intSupplier.get());

Result: 1643357758 or -17532822 or any other random int

Example 2:

**private** **void** supplier() {

Random r = **new** Random();

Supplier<Integer> intSupplier = () -> r.nextInt(25);

**int** len = 10;

Supplier<String> randomStrSupplier = () -> {

String alpha = "abcdefghijklmnopqrstuvwxyz";

StringBuilder sb = **new** StringBuilder();

**for** (**int** i = 0; i < len; i++) {

sb.append(alpha.toCharArray()[intSupplier.get()]);

}

**return** sb.toString();

};

System.***out***.println(randomStrSupplier.get());

}

Result: nljwotquta or uhypnprjxv or any other 10 sized string

### Custom Functional Interfaces

It is possible to create your own functional interfaces. You just have to annotate them with:

Annotation Type FunctionalInterface

[@Documented](https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Documented.html)

[@Retention](https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Retention.html)([value](https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Retention.html#value--)=[RUNTIME](https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/RetentionPolicy.html#RUNTIME))

[@Target](https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Target.html)([value](https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Target.html#value--)=[TYPE](https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/ElementType.html#TYPE))

public @interface FunctionalInterface

An informative annotation type used to indicate that an interface type declaration is intended to be a *functional interface* as defined by the Java Language Specification. Conceptually, a functional interface has exactly one abstract method. Since [default methods](https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Method.html#isDefault--) have an implementation, they are not abstract. 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.

Note that instances of functional interfaces can be created with lambda expressions, method references, or constructor references.

If a type is annotated with this annotation type, compilers are required to generate an error message unless:

* The type is an interface type and not an annotation type, enum, or class.
* The annotated type satisfies the requirements of a functional interface.

However, the compiler will treat any interface meeting the definition of a functional interface as a functional interface regardless of whether or not a FunctionalInterface annotation is present on the interface declaration.

Example:

MyPersonalFunctionalInterface<String> mine = (x) -> System.***out***.println("Hello "+x);

mine.say("Vlad");

where MyPersonalFunctionalInterface is:

@FunctionalInterface

**public** **interface** MyPersonalFunctionalInterface<T> {

**public** **void** say(T arg);

}

## Method references

<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>

There are four kinds of method references:

|  |  |
| --- | --- |
| **Kind** | **Example** |
| Reference to a static method | ContainingClass::staticMethodName |
| Reference to an instance method of a particular object | containingObject::instanceMethodName |
| Reference to an instance method of an arbitrary object of a particular type | ContainingType::methodName |
| Reference to a constructor | ClassName::new |

### Reference to a Static Method

Here you can reference a static method from your class like this: YourClass::theStaticMethod. In our case it is: Car::compareByMileage. But this makes sense in a context because you are not providing the method arguments: Arrays.sort(carsArray, Car::compareByMileage); The method reference Car::compareByMileage is equivalent to the lambda expression:(a,b) -> Car.compareByMileage(a,b)

### Reference to an Instance Method of a Particular Object

Here is an example of accessing a non-static method on an instance. Again the context is important. You can do this only where the method is expecting a functional interface.

ComparisonProvider comparisonProvider = **new** ComparisonProvider();

Arrays.sort(carsArray, comparisonProvider::compareByBrand);

Here, comparisonProvider.compareByBrand would be:

**public** **class** ComparisonProvider {

**public** **int** compareByBrand(Car a, Car b) {

**return** a.getBrand().compareTo(b.getBrand());

}

}

In this case, comparisonProvider::compareByBrand invokes the compareByName method and JRE infers the method arguments (Car, Car).

Another interesting example here. Instead of this:

File[] hiddenFiles = **new** File(".").listFiles(**new** FileFilter() {

**public** **boolean** accept(File file) {

**return** file.isHidden();

}

});

Do this:

File[] hiddenFiles = **new** File(".").listFiles(File::isHidden);

### Reference to an Instance Method of an Arbitrary Object of a Particular Type

Here we show an example of an instance method of an arbitrary type:

String[] array = { "Zoi", "Wow", "Aoc", "Dao", "Fao", "Aao" };

Arrays.*sort*(array, String::compareToIgnoreCase);

### Reference to a Constructor

You can refer to constructors just like you would refer to a static method:

Set<Car> carsSet = transferElements(cars, HashSet::new);

The Java compiler infers that you want to create a HashSet collection that contains elements of type Person. Alternatively, you can specify this as follows:

Set<Car> carsSetNew = *transferElements*(cars, HashSet<Car>::**new**);

This is equivalent to:

Set<Car> carsSetLambda = *transferElements*(cars, ()->{**return** **new** HashSet<>();});

Where transferElements is:

**public** **static** <T, SOURCE **extends** Collection<T>, DEST **extends** Collection<T>>

DEST transferElements(SOURCE sourceCollection, Supplier<DEST> collectionFactory) {

DEST result = collectionFactory.get();

**for** (T t : sourceCollection) {

result.add(t);

}

**return** result;

}

The functional interface Supplier contains one method get that takes no arguments and returns an object.

## Default methods and statics in interfaces

<https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html>

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.

You specify that a method definition in an interface is a default method with the default keyword at the beginning of the method signature.

Here is an example of an interface called MonitorControls:

**public** **interface** MonitorControls {

**void** showMenu();

**void** adjustBrightness(Integer newValue);

**void** adjustContrast(Integer newValue);

**boolean** getStatus();

**void** setStatus(**boolean** on);

**default** **void** power() {

**if** (getStatus()) {

System.***out***.println("Turning off");

setStatus(**false**);

} **else** {

System.***out***.println("Turning on");

setStatus(**true**);

}

}

In this case, the method power is a default method. It provides the implementation. One interesting thing is that you can call other methods from default methods and they will get correctly resolved during runtime. Here is the class that implements this interface:

**public** **class** DesktopMonitorControls **implements** MonitorControls, Connectors {

**private** **boolean** on;

@Override

**public** **void** showMenu() {

System.***out***.println("Showing Menu");

}

@Override

**public** **void** adjustBrightness(Integer newValue) {

System.***out***.println("Adjusting Brightness");

}

@Override

**public** **void** adjustContrast(Integer newValue) {

System.***out***.println("Adjusting Contrast");

}

@Override

**public** **boolean** getStatus() {

**return** **this**.on;

}

@Override

**public** **void** setStatus(**boolean** on) {

**this**.on = on;

}

}

To be noted here that there is no implementation of the power method provided. If we use this class as it is:

MonitorControls controls = **new** DesktopMonitorControls();

controls.power();

controls.adjustBrightness(10);

controls.power();

we get the result:

Turning on

Adjusting Brightness

Turning off

As by magic, the power() method becomes available and more interestingly, it can correctly resolve the current status of the monitor by calling the getStatus() implementation.

What if you provide an implementation of power()?

@Override

**public** **void** power() {

System.***out***.println("NEW POWER");

}

In this case you get as expected:

NEW POWER

Adjusting Brightness

NEW POWER

Another interesting question: what happens if 2 interfaces have the same default method? Say you have this interface:

**public** **interface** Connectors {

**void** hdmi();

**void** rgb();

**default** **void** power(){

System.***out***.println("Power cable is present");

}

}

And your class implements this interface as well:

**public** **class** DesktopMonitorControls **implements** MonitorControls, Connectors {

In this case you’ll get the compilation error:

Duplicate default methods named power with the parameters () and () are inherited from the types Connectors and MonitorControls

So the compiler doesn’t know which default version to invoke during runtime, the one from Connectors or from MonitorControls. An IDE would propose to implement the method either from MonitorControls or from Connectors and it will generate either:

@Override

**public** **void** power() {

// **TODO** Auto-generated method stub

MonitorControls.**super**.power();

}

Or

@Override

**public** **void** power() {

// **TODO** Auto-generated method stub

Connectors.**super**.power();

}

Interestingly enough, it uses the keyword **super** which usually refers to classes. Care should be taken when using these default methods as they might get easily misused.

In addition to default methods, you can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) 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.

You can add this into your Connectors interface:

**static** **void** mystat(){

System.***out***.println("Nothing Special");

}

And then use it:

Connectors.*mystat*();

## Repeating Annotations

## Type Annotations

## Improved type inference

## Method parameter reflection

# java.lang and java.util Packages

## Standard Encoding and Decoding Base64

<https://en.wikipedia.org/wiki/Base64>

A quote from Thomas Hobbes' *Leviathan* (be aware of spaces between lines):

Man is distinguished, not only by his reason, but by this singular passion from

other animals, which is a lust of the mind, that by a perseverance of delight

in the continued and indefatigable generation of knowledge, exceeds the short

vehemence of any carnal pleasure.

is represented as a byte sequence of 8-bit-padded [ASCII](https://en.wikipedia.org/wiki/ASCII) characters encoded in [MIME](https://en.wikipedia.org/wiki/MIME)'s Base64 scheme as follows:

TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBieSB0aGlz

IHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaCBpcyBhIGx1c3Qgb2Yg

dGhlIG1pbmQsIHRoYXQgYnkgYSBwZXJzZXZlcmFuY2Ugb2YgZGVsaWdodCBpbiB0aGUgY29udGlu

dWVkIGFuZCBpbmRlZmF0aWdhYmxlIGdlbmVyYXRpb24gb2Yga25vd2xlZGdlLCBleGNlZWRzIHRo

ZSBzaG9ydCB2ZWhlbWVuY2Ugb2YgYW55IGNhcm5hbCBwbGVhc3VyZS4=

In the above quote, the encoded value of *Man* is *TWFu*. Encoded in ASCII, the characters *M*, *a*, and *n* are stored as the bytes 77, 97, and 110, which are the 8-bit binary values 01001101, 01100001, and 01101110. These three values are joined together into a 24-bit string, producing010011010110000101101110. Groups of 6 bits (6 bits have a maximum of 26 = 64 different binary values) are [converted into individual numbers](https://en.wikipedia.org/wiki/Binary_number#Counting_in_binary) from left to right (in this case, there are four numbers in a 24-bit string), which are then converted into their corresponding Base64 character values.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Text content** | **M** | | | | | | | | **a** | | | | | | | | **n** | | | | | | | |
| **ASCII** | 77 (0x4d) | | | | | | | | 97 (0x61) | | | | | | | | 110 (0x6e) | | | | | | | |
| **Bit pattern** | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| **Index** | 19 | | | | | | 22 | | | | | | 5 | | | | | | 46 | | | | | |
| **Base64-encoded** | **T** | | | | | | **W** | | | | | | **F** | | | | | | **u** | | | | | |

As this example illustrates, Base64 encoding converts three [octets](https://en.wikipedia.org/wiki/Octet_(computing)) into four encoded characters.

The Base64 index table:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | **Char** |  | **Value** | **Char** |  | **Value** | **Char** |  | **Value** | **Char** |
| 0 | A | 16 | Q | 32 | g | 48 | w |
| 1 | B | 17 | R | 33 | h | 49 | x |
| 2 | C | 18 | S | 34 | i | 50 | y |
| 3 | D | 19 | T | 35 | j | 51 | z |
| 4 | E | 20 | U | 36 | k | 52 | 0 |
| 5 | F | 21 | V | 37 | l | 53 | 1 |
| 6 | G | 22 | W | 38 | m | 54 | 2 |
| 7 | H | 23 | X | 39 | n | 55 | 3 |
| 8 | I | 24 | Y | 40 | o | 56 | 4 |
| 9 | J | 25 | Z | 41 | p | 57 | 5 |
| 10 | K | 26 | a | 42 | q | 58 | 6 |
| 11 | L | 27 | b | 43 | r | 59 | 7 |
| 12 | M | 28 | c | 44 | s | 60 | 8 |
| 13 | N | 29 | d | 45 | t | 61 | 9 |
| 14 | O | 30 | e | 46 | u | 62 | + |
| 15 | P | 31 | f | 47 | v | 63 | / |

When the number of bytes to encode is not divisible by three (that is, if there are only one or two bytes of input for the last 24-bit block), then the following action is performed:

Add extra bytes with value zero so there are three bytes, and perform the conversion to base64. If there was only one significant input byte, only the first two base64 digits are picked (12 bits), and if there were two significant input bytes, the first three base64 digits are picked (18 bits). '=' characters might be added to make the last block contain four base64 characters.

As a result, when the last group contains one octet, the four [least significant bits](https://en.wikipedia.org/wiki/Least_significant_bit) of the final 6-bit block are set to zero; and when the last group contains two octets, the two least significant bits of the final 6-bit block are set to zero.

Here is an example of how to use Encoder and Decoder in Java 8:

**package** com.vvirlan;

**import** java.net.URL;

**import** java.**util**.Base64;

**class** Base64Sample {

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

**try** {

// Encode URL

URL originalUrl = **new** URL("http://example.com");

Base64.Encoder urlEncoder = Base64.*getUrlEncoder*();

System.***out***.println("URL: " + originalUrl);

**byte**[] encodedUrl = urlEncoder.encode(originalUrl.toString().getBytes("UTF8"));

System.***out***.println("Base64 Encoded URL : " + **new** String(encodedUrl, "UTF-8"));

// Encode text

String originalText = "thisIsAnExampleString";

Base64.Encoder textEncoder = Base64.*getEncoder*();

**byte**[] encodedText = textEncoder.encode(originalText.getBytes("UTF-8"));

System.***out***.println("Base64 Encoded String : " + **new** String(encodedText, "UTF-8"));

// Decode URL and text

Base64.Decoder urlDecoder = Base64.*getUrlDecoder*();

Base64.Decoder textDecoder = Base64.*getDecoder*();

**byte**[] urlDecoded = urlDecoder.decode(encodedUrl);

**byte**[] textDecoded = textDecoder.decode(encodedText);

System.***out***.println("Base64 Decoded URL : " + **new** String(urlDecoded, "UTF-8"));

System.***out***.println("Base64 Decoded String : " + **new** String(textDecoded, "UTF-8"));

}

**catch** (Exception e) {

System.***out***.println("Invalid URL Exception");

}

}

}

Here are the APIs for **Base64**:

public class **Base64** extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

This class consists exclusively of static methods for obtaining encoders and decoders for the Base64 encoding scheme. The implementation of this class supports the following types of Base64 as specified in [RFC 4648](http://www.ietf.org/rfc/rfc4648.txt) and [RFC 2045](http://www.ietf.org/rfc/rfc2045.txt).

**Basic**

Uses "The Base64 Alphabet" as specified in Table 1 of RFC 4648 and RFC 2045 for encoding and decoding operation. The encoder does not add any line feed (line separator) character. The decoder rejects data that contains characters outside the base64 alphabet.

**URL and Filename safe**

Uses the "URL and Filename safe Base64 Alphabet" as specified in Table 2 of RFC 4648 for encoding and decoding. The encoder does not add any line feed (line separator) character. The decoder rejects data that contains characters outside the base64 alphabet.

**MIME**

Uses the "The Base64 Alphabet" as specified in Table 1 of RFC 2045 for encoding and decoding operation. The encoded output must be represented in lines of no more than 76 characters each and uses a carriage return '\r' followed immediately by a linefeed '\n' as the line separator. No line separator is added to the end of the encoded output. All line separators or other characters not found in the base64 alphabet table are ignored in decoding operation.

Unless otherwise noted, passing a null argument to a method of this class will cause a [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) to be thrown.

|  |  |
| --- | --- |
| static [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) | [**getDecoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getDecoder--)()  Returns a [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) that decodes using the [**Basic**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#basic) type base64 encoding scheme. |
| static [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**getEncoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getEncoder--)()  Returns a [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) that encodes using the [**Basic**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#basic) type base64 encoding scheme. |
| static [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) | [**getMimeDecoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getMimeDecoder--)()  Returns a [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) that decodes using the [**MIME**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#mime) type base64 decoding scheme. |
| static [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**getMimeEncoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getMimeEncoder--)()  Returns a [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) that encodes using the [**MIME**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#mime) type base64 encoding scheme. |
| static [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**getMimeEncoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getMimeEncoder-int-byte:A-)(int lineLength, byte[] lineSeparator)  Returns a [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) that encodes using the [**MIME**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#mime) type base64 encoding scheme with specified line length and line separators. |
| static [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) | [**getUrlDecoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getUrlDecoder--)()  Returns a [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) that decodes using the [**URL and Filename safe**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#url) type base64 encoding scheme. |
| static [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**getUrlEncoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getUrlEncoder--)()  Returns a [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) that encodes using the [**URL and Filename safe**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#url) type base64 encoding scheme. |

public static class **Base64.Decoder** extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. The Base64 padding character '=' is accepted and interpreted as the end of the encoded byte data, but is not required. So if the final unit of the encoded byte data only has two or three Base64 characters (without the corresponding padding character(s) padded), they are decoded as if followed by padding character(s). If there is a padding character present in the final unit, the correct number of padding character(s) must be present, otherwise IllegalArgumentException (IOException when reading from a Base64 stream) is thrown during decoding.

Instances of [Base64.Decoder](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) class are safe for use by multiple concurrent threads.

Unless otherwise noted, passing a null argument to a method of this class will cause a [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) to be thrown.

|  |  |
| --- | --- |
| byte[] | [**decode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#decode-byte:A-)(byte[] src)  Decodes all bytes from the input byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme, writing the results into a newly-allocated output byte array. |
| int | [**decode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#decode-byte:A-byte:A-)(byte[] src, byte[] dst)  Decodes all bytes from the input byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme, writing the results into the given output byte array, starting at offset 0. |
| [**ByteBuffer**](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) | [**decode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#decode-java.nio.ByteBuffer-)([**ByteBuffer**](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) buffer)  Decodes all bytes from the input byte buffer using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme, writing the results into a newly-allocated ByteBuffer. |
| byte[] | [**decode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#decode-java.lang.String-)([**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) src)  Decodes a Base64 encoded String into a newly-allocated byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme. |
| [**InputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html) | [**wrap**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#wrap-java.io.InputStream-)([**InputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html) is)  Returns an input stream for decoding [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoded byte stream. |

public static class **Base64.Encoder** extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. Instances of [Base64.Encoder](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) class are safe for use by multiple concurrent threads. Unless otherwise noted, passing a null argument to a method of this class will cause a [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) to be thrown.

|  |  |
| --- | --- |
| byte[] | [**encode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#encode-byte:A-)(byte[] src)  Encodes all bytes from the specified byte array into a newly-allocated byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme. |
| int | [**encode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#encode-byte:A-byte:A-)(byte[] src, byte[] dst)  Encodes all bytes from the specified byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme, writing the resulting bytes to the given output byte array, starting at offset 0. |
| [**ByteBuffer**](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) | [**encode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#encode-java.nio.ByteBuffer-)([**ByteBuffer**](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) buffer)  Encodes all remaining bytes from the specified byte buffer into a newly-allocated ByteBuffer using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html)encoding scheme. |
| [**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) | [**encodeToString**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#encodeToString-byte:A-)(byte[] src)  Encodes the specified byte array into a String using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme. |
| [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**withoutPadding**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#withoutPadding--)()  Returns an encoder instance that encodes equivalently to this one, but without adding any padding character at the end of the encoded byte data. |
| [**OutputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html) | [**wrap**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#wrap-java.io.OutputStream-)([**OutputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html) os)  Wraps an output stream for encoding byte data using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme. |

## Unsigned Arithmetic Support

# Stream API

## Aggregate Operations

<https://docs.oracle.com/javase/tutorial/collections/streams/index.html>

**Differences Between Aggregate Operations and Iterators**

Aggregate operations, like forEach, appear to be like iterators. However, they have several fundamental differences:

* **They use internal iteration**: Aggregate operations do not contain a method like next to instruct them to process the next element of the collection. With internal delegation, your application determines what collection it iterates, but the JDK determines how to iterate the collection. With external iteration, your application determines both what collection it iterates and how it iterates it. However, external iteration can only iterate over the elements of a collection sequentially. Internal iteration does not have this limitation. It can more easily take advantage of parallel computing, which involves dividing a problem into sub-problems, solving those problems simultaneously, and then combining the results of the solutions to the sub-problems. See the section [Parallelism](https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html) for more information.
* **They process elements from a stream**: Aggregate operations process elements from a stream, not directly from a collection. Consequently, they are also called stream operations.
* **They support behavior as parameters**: You can specify [lambda expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) as parameters for most aggregate operations. This enables you to customize the behavior of a particular aggregate operation.

## Reduction

<https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html>

Map<Person.Sex, List<Person>> byGender =

persons

.stream()

.collect(Collectors.*groupingBy*(Person::getSex));

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

## Parallelism

<https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html>

An example of parallel stream processing:

Date start = **new** Date();

**double** average = persons

.parallelStream()

.filter(p->p.getSex().equals(Person.Sex.***MALE***))

.mapToInt(Person::getAge)

.average()

.getAsDouble();

Date end = **new** Date();

**long** diff = end.getTime() - start.getTime();

System.***out***.println("Average age of Males: "+average+" Time: "+diff+" ms.");

# Date Time API

# Optional

# Concurrency

# Tools

## jdeps

# Nashorn, JavaScript Engine