Оглавление

[Функциональный интерфейс](#_Toc46093412)

[Runnable](#_Toc46093413)

[Function<T, R>: R apply(T t)](#_Toc46093414)

[Function<T, R>: R apply(T t) (IntFunction<R>, LongFunction<R>, DoubleFunction<R>)](#_Toc46093415)

[UnaryOperator<T> (унарный оператор): T apply(T t) (UnaryOperator<T>, IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator)](#_Toc46093416)

[\_To\_Function (IntToLongFunction, IntToDoubleFunction, LongToIntFunction, LongToDoubleFunction, DoubleToIntFunction, DoubleToLongFunction)](#_Toc46093417)

[To\_Function<T> (ToIntFunction<T>, ToLongFunction<T>, ToDoubleFunction<T>)](#_Toc46093418)

[To\_BiFunction<T, U> (ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>)](#_Toc46093419)

[BiFunction<T, U, R>: R apply(T t, U u)](#_Toc46093420)

[BinaryOperator<T> (бинарный оператор): T apply(T t, T u) (IntBinaryOperator, LongBinaryOperator)](#_Toc46093421)

[Predicate<T>: boolean test(T t)](#_Toc46093422)

[Predicate<T>: boolean test(T t) (IntPredicate, LongPredicate, DoublePredicate)](#_Toc46093423)

[BiPredicate<T, U>: boolean test(T t, U u)](#_Toc46093424)

[Consumer<T> (потребитель): void accept(T t)](#_Toc46093425)

[Consumer<T>: void accept(T t) (IntConsumer, LongConsumer, DoubleConsumer)](#_Toc46093426)

[BiConsumer<T, U>: void accept(T t, U u)](#_Toc46093427)

[Obj\_Consumer<T> (ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>)](#_Toc46093428)

[Supplier<T> (поставщик): T get()](#_Toc46093429)

[Supplier<T>: T get() (IntSupplier, LongSupplier, DoubleSupplier)](#_Toc46093430)

[BooleanSupplier: boolean getAsBoolean()](#_Toc46093431)

# Функциональный интерфейс

Интерфейс, который определяет только один абстрактный метод.

Чтобы точно определить интерфейс как функциональный, добавляется аннотация @FunctionalInterface, работающая по принципу @Override. Она обозначит замысел и не даст определить второй абстрактный метод в интерфейсе.

Интерфейс может включать сколько угодно default методов и при этом оставаться функциональным, потому что default методы — не абстрактные.

# Runnable

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.lang;  @FunctionalInterface  public interface Runnable {  public abstract void run();  } |

# Function<T, R>: R apply(T t)

## Function<T, R>: R apply(T t) (IntFunction<R>, LongFunction<R>, DoubleFunction<R>)

Методы по умолчанию могут использоваться для построения цепочек вызовов (compose, andThen).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Function<String, Integer> toInteger =  Integer::valueOf;  Function<String, String> backToString =  toInteger.andThen(String::valueOf);  backToString.apply("123"); // "123" |

Function<T, R>: R apply(T t)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | @FunctionalInterface  public interface Function<T, R> {  R apply(T t);  default <V> Function<V, R> compose(  Function<? super V, ? extends T>  before) {  Objects.requireNonNull(before);  return (V v) -> apply(before.apply(v));  }  default <V> Function<T, V> andThen(  Function<? super R, ? extends V>  after) {  Objects.requireNonNull(after);  return (T t) -> after.apply(apply(t));  }  static <T> Function<T, T> identity() {  return t -> t;  }  } |

IntFunction<R>: R apply(int value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface IntFunction<R> {  R apply(int value);  } |

LongFunction<R>: R apply(long value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface LongFunction<R> {  R apply(long value);  } |

DoubleFunction<R>: R apply(double value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface DoubleFunction<R> {  R apply(double value);  } |

## UnaryOperator<T> (унарный оператор): T apply(T t) (UnaryOperator<T>, IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator)

|  |  |
| --- | --- |
| 1.  2. | UnaryOperator<Integer> operator = x -> x \* x;  System.out.println(operator.apply(5)); // 25 |

UnaryOperator<T>: T apply(T t)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | @FunctionalInterface  public interface UnaryOperator<T> extends Function<T, T> {  static <T> UnaryOperator<T> identity() {  return t -> t;  }  } |

IntUnaryOperator: int applyAsInt(int operand)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | @FunctionalInterface  public interface IntUnaryOperator {  int applyAsInt(int operand);  default IntUnaryOperator compose(  IntUnaryOperator before) {  Objects.requireNonNull(before);  return (int v) -> applyAsInt(  before.applyAsInt(v));  }  default IntUnaryOperator andThen(  IntUnaryOperator after) {  Objects.requireNonNull(after);  return (int t) -> after.applyAsInt(  applyAsInt(t));  }  static IntUnaryOperator identity() {  return t -> t;  }  } |

LongUnaryOperator: long applyAsLong(long operand)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | @FunctionalInterface  public interface LongUnaryOperator {  long applyAsLong(long operand);  default LongUnaryOperator compose(LongUnaryOperator before) {  Objects.requireNonNull(before);  return (long v) -> applyAsLong(before.applyAsLong(v));  }  default LongUnaryOperator andThen(LongUnaryOperator after) {  Objects.requireNonNull(after);  return (long t) -> after.applyAsLong(applyAsLong(t));  }  static LongUnaryOperator identity() {  return t -> t;  }  } |

DoubleUnaryOperator: double applyAsDouble(double operand)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | @FunctionalInterface  public interface DoubleUnaryOperator {  double applyAsDouble(double operand);  default DoubleUnaryOperator compose(  DoubleUnaryOperator before) {  Objects.requireNonNull(before);  return (double v) -> applyAsDouble(  before.applyAsDouble(v));  }  default DoubleUnaryOperator andThen(  DoubleUnaryOperator after) {  Objects.requireNonNull(after);  return (double t) -> after.applyAsDouble(  applyAsDouble(t));  }  static DoubleUnaryOperator identity() {  return t -> t;  }  } |

## \_To\_Function (IntToLongFunction, IntToDoubleFunction, LongToIntFunction, LongToDoubleFunction, DoubleToIntFunction, DoubleToLongFunction)

IntToLongFunction: long applyAsLong(int value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface IntToLongFunction {  long applyAsLong(int value);  } |

IntToDoubleFunction: double applyAsDouble(int value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface IntToDoubleFunction {  double applyAsDouble(int value);  } |

LongToIntFunction: int applyAsInt(long value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface LongToIntFunction {  int applyAsInt(long value);  } |

LongToDoubleFunction: double applyAsDouble(long value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface LongToDoubleFunction {  double applyAsDouble(long value);  } |

DoubleToIntFunction: int applyAsInt(double value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface DoubleToIntFunction {  int applyAsInt(double value);  } |

DoubleToLongFunction: long applyAsLong(double value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface DoubleToIntFunction {  int applyAsInt(double value);  } |

## To\_Function<T> (ToIntFunction<T>, ToLongFunction<T>, ToDoubleFunction<T>)

ToIntFunction<T>: int applyAsInt(T value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ToIntFunction<T> {  int applyAsInt(T value);  } |

ToLongFunction<T>: long applyAsLong(T value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ToLongFunction<T> {  long applyAsLong(T value);  } |

ToDoubleFunction<T>: double applyAsDouble(T value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ToDoubleFunction<T> {  double applyAsDouble(T value);  } |

## To\_BiFunction<T, U> (ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>)

ToIntBiFunction<T, U>: int applyAsInt(T t, U u)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ToIntBiFunction<T, U> {  int applyAsInt(T t, U u);  } |

ToLongBiFunction<T, U>: long applyAsLong(T t, U u)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ToLongBiFunction<T, U> {  long applyAsLong(T t, U u);  } |

ToDoubleBiFunction<T, U>: double applyAsDouble(T t, U u)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ToDoubleBiFunction<T, U> {  double applyAsDouble(T t, U u);  } |

## BiFunction<T, U, R>: R apply(T t, U u)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | @FunctionalInterface  public interface BiFunction<T, U, R> {  R apply(T t, U u);  default <V> BiFunction<T, U, V> andThen(  Function<? super R, ? extends V>  after) {  Objects.requireNonNull(after);  return (T t, U u) -> after.apply(  apply(t, u));  }  } |

## BinaryOperator<T> (бинарный оператор): T apply(T t, T u) (IntBinaryOperator, LongBinaryOperator)

|  |  |
| --- | --- |
| 1.  2.  3. | BinaryOperator<Integer> operator =  (a, b) -> a + b;  System.out.println(operator.apply(1, 2)); // 3 |

BinaryOperator<T>: T apply(T t, T u)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | @FunctionalInterface  public interface BinaryOperator<T>  extends BiFunction<T,T,T> {  public static <T> BinaryOperator<T> minBy(  Comparator<? super T> comparator) {  Objects.requireNonNull(comparator);  return (a, b) -> comparator.compare(a, b)  <= 0 ? a : b;  }  public static <T> BinaryOperator<T> maxBy(  Comparator<? super T> comparator) {  Objects.requireNonNull(comparator);  return (a, b) -> comparator.compare(a, b)  >= 0 ? a : b;  }  } |

IntBinaryOperator: int applyAsInt(int left, int right)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface IntBinaryOperator {  int applyAsInt(int left, int right);  } |

LongBinaryOperator: long applyAsLong(long left, long right)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface LongBinaryOperator {  long applyAsLong(long left, long right);  } |

DoubleBinaryOperator: double applyAsDouble(double left, double right)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface DoubleBinaryOperator {  double applyAsDouble(double left, double right); |

# Predicate<T>: boolean test(T t)

## Predicate<T>: boolean test(T t) (IntPredicate, LongPredicate, DoublePredicate)

Интерфейс содержит различные методы по умолчанию, позволяющие строить сложные условия (and, or, negate).

|  |  |
| --- | --- |
| 1.  2.  3. | Predicate predicate = (s) -> s.length() > 0;  predicate.test("foo"); // true  predicate.negate().test("foo"); // false |

Predicate<T>: boolean test(T t)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | @FunctionalInterface  public interface Predicate<T> {  boolean test(T t);  default Predicate<T> and(  Predicate<? super T> other) {  Objects.requireNonNull(other);  return (t) -> test(t) && other.test(t);  }  default Predicate<T> negate() {  return (t) -> !test(t);  }  default Predicate<T> or(  Predicate<? super T> other) {  Objects.requireNonNull(other);  return (t) -> test(t) || other.test(t);  }  static <T> Predicate<T> isEqual(  Object targetRef) {  return (null == targetRef)  ? Objects::isNull  : object -> targetRef.equals(object);  }  } |

IntPredicate: boolean test(int value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21. | @FunctionalInterface  public interface IntPredicate {  boolean test(int value);  default IntPredicate and(IntPredicate other) {  Objects.requireNonNull(other);  return (value) -> test(value)  && other.test(value);  }  default IntPredicate negate() {  return (value) -> !test(value);  }  default IntPredicate or(IntPredicate other) {  Objects.requireNonNull(other);  return (value) -> test(value)  || other.test(value);  }  } |

LongPredicate: boolean test(long value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | @FunctionalInterface  public interface LongPredicate {  boolean test(long value);  default LongPredicate and(  LongPredicate other) {  Objects.requireNonNull(other);  return (value) -> test(value)  && other.test(value);  }  default LongPredicate negate() {  return (value) -> !test(value);  }  default LongPredicate or(  LongPredicate other) {  Objects.requireNonNull(other);  return (value) -> test(value)  || other.test(value);  }  } |

DoublePredicate: boolean test(double value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | @FunctionalInterface  public interface DoublePredicate {  boolean test(double value);  default DoublePredicate and(  DoublePredicate other) {  Objects.requireNonNull(other);  return (value) -> test(value)  && other.test(value);  }  default DoublePredicate negate() {  return (value) -> !test(value);  }  default DoublePredicate or(  DoublePredicate other) {  Objects.requireNonNull(other);  return (value) -> test(value)  || other.test(value);  }  } |

## BiPredicate<T, U>: boolean test(T t, U u)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | @FunctionalInterface  public interface BiPredicate<T, U> {  boolean test(T t, U u);  default BiPredicate<T, U> and(  BiPredicate<? super T, ? super U>  other) {  Objects.requireNonNull(other);  return (T t, U u) -> test(t, u)  && other.test(t, u);  }  default BiPredicate<T, U> negate() {  return (T t, U u) -> !test(t, u);  }  default BiPredicate<T, U> or(  BiPredicate<? super T, ? super U>  other) {  Objects.requireNonNull(other);  return (T t, U u) -> test(t, u)  || other.test(t, u);  }  } |

# Consumer<T> (потребитель): void accept(T t)

## Consumer<T>: void accept(T t) (IntConsumer, LongConsumer, DoubleConsumer)

|  |  |
| --- | --- |
| 1.  2.  3. | Consumer<String> hello = (name) ->  System.out.println("Hello, " + name);  hello.accept("world"); |

Consumer<T>: void accept(T t)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | @FunctionalInterface  public interface Consumer<T> {  void accept(T t);  default Consumer<T> andThen(  Consumer<? super T> after) {  Objects.requireNonNull(after);  return (T t) -> {  accept(t);  after.accept(t);};  }  } |

IntConsumer: void accept(int value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | @FunctionalInterface  public interface IntConsumer {  void accept(int value);  default IntConsumer andThen(  IntConsumer after) {  Objects.requireNonNull(after);  return (int t) -> {  accept(t);  after.accept(t); };  }  } |

LongConsumer: void accept(long value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | @FunctionalInterface  public interface LongConsumer {  void accept(long value);  default LongConsumer andThen(  LongConsumer after) {  Objects.requireNonNull(after);  return (long t) -> {  accept(t);  after.accept(t); };  }  } |

DoubleConsumer: void accept(double value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | @FunctionalInterface  public interface DoubleConsumer {  void accept(double value);  default DoubleConsumer andThen(  DoubleConsumer after) {  Objects.requireNonNull(after);  return (double t) -> {  accept(t);  after.accept(t); };  }  } |

## BiConsumer<T, U>: void accept(T t, U u)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | @FunctionalInterface  public interface BiConsumer<T, U> {  void accept(T t, U u);  default BiConsumer<T, U> andThen(  BiConsumer<? super T, ? super U>  after) {  Objects.requireNonNull(after);  return (l, r) -> {  accept(l, r);  after.accept(l, r);  };  }  } |

## Obj\_Consumer<T> (ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>)

ObjIntConsumer<T>: void accept(T t, int value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ObjIntConsumer<T> {  void accept(T t, int value);  } |

ObjLongConsumer<T>: void accept(T t, long value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ObjLongConsumer<T> {  void accept(T t, long value);  } |

ObjDoubleConsumer<T>: void accept(T t, double value)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface ObjDoubleConsumer<T> {  void accept(T t, double value);  } |

# Supplier<T> (поставщик): T get()

## Supplier<T>: T get() (IntSupplier, LongSupplier, DoubleSupplier)

|  |  |
| --- | --- |
| 1.  2. | Supplier now = LocalDateTime::now;  now.get(); |

Supplier<T>: T get()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface Supplier<T> {  T get();  } |

IntSupplier: int getAsInt()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface IntSupplier {  int getAsInt();  } |

LongSupplier: long getAsLong()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface LongSupplier {  long getAsLong();  } |

DoubleSupplier: double getAsDouble()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface DoubleSupplier {  double getAsDouble();  } |

## BooleanSupplier: boolean getAsBoolean()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @FunctionalInterface  public interface BooleanSupplier {  boolean getAsBoolean();  } |