Методы интерфейса публичные и абстрактные (явно не пишут). Поля — публичные, статические и финальные (константы). Могут быть публичные статический методы и default‑методы (с реализацией по умолчанию).

Интерфейс называется функциональным, если в нем один абстрактный метод (дефолтные и статические методы не в счет). Используется, чтобы сделать код более удобным и менее громоздким.

Помечается аннотацией @FunctionalInterface. При наличии — код не скомпилируется, если больше (меньше) 1-го абстрактного метода.

Объявление

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.io;  @FunctionalInterface  public interface FileFilter {  Boolean accept(File pathname);  } |

Семейства

|  |  |
| --- | --- |
| 1. | Consumer<T> (потребители)  void accept(T t);  int, long, double, Bi |
| 2. | Supplier<T> (поставщики)  T get()  boolean, int, long, double. |
| 3. | Predicate<T>  boolean test(T t)  int, long, double, Bi |
| 4. | Function<T, R>  R apply(T t)  комбинации double, long, int; Bi |
| 5. | UnaryOperator<T> (операторы)  T apply(T t) (???)  int, long, double, Binary |

Создание

1) Завести именованный анонимный класс, реализующий интерфейс.

IN — имя интерфейса; mIN — имя метода интерфейса IN; vN — имя переменной; cN — имя класса.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | psvm() {  IN vN = new IN();  vN.mIN (new cN());  }  private static class cN  implements anyInterface {  @Override  // переопределение метода  } |

2) Завести анонимный класс, реализующий интерфейс.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | IN vN = new IN();  vN.mIN(new anyInterface() {  @Override  // переопределение метода  }); |

3) Использовать лямбда‑выражения.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | IN vN = new IN();  vN.mIN(() -> {  // тело лямбды выражения  }); |

4) Ссылка на метод.

|  |  |
| --- | --- |
| 1.  2. | IN vN = new IN();  vN.mIN(anyClass::anyMethod); |

Анонимный класс — вызов класса (интерфейса) при помощи new с переопределенным методом (методами).

Если тело из единственного выражения return, то return не пишется.

Переменны лямбда‑выражения (обращаться)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | pubic class Demo {  private int counter;  public void foo() {  IntUnaryOperator square =  x -> **x \* x**;  IntSupplier sequence =  () -> **counter++**;  int **bonus** = 10;  IntUnaryOperator bonusAdder =  (x) -> x + **bonus**;  }  } |

Можно обращаться  
- к параметрам лямбды (7);  
- объявлять и использовать переменные внутри тела лямбда выражения (10);  
- обращаться к полям класса (как читать, так и писать);  
- обращаться к переменным внутри метода (12,14: bonus) (переменная финальная, то есть значения им должны быть присвоено ровно один раз до создания лямбды, после чего меняться оно уже не может. final можно не писать).

|  |  |
| --- | --- |
| 1.  2.  3. | int[] counter = new int[] {0};  IntSupplier sequence = () -> counter[0]++; |

(Отсюда, в частности, следует, что из лямбда нельзя присваивать новое значение переменным содержащего ее метода. Для обхода этого ограничения иногда используют трюк с массивом единичной длины (1: new int[] {0}). Ссылка на массив (1: counter) является эффективно финальной, однако на содержимое массива это ограничение не распространяется (3: counter[0]++) и его можно обновлять.)

Самый простой, есть нюансы:  
- можем ссылаться на стат. метод (2) [имя\_класса::имя\_стат.метода];  
- можем ссылаться на нестат. метод (5), [объект::имя\_нестат.\_метода];  
- способ на нестат. метод (8) [имя\_класса::имя\_нестат.метода] (но тогда первый передаваемый в месте вызова функционального интерфейса параметр (7: Object) будет тем самым объектом, на котором данный нестатический метод (8: toString) будет вызван)  
- ссылаться на конструктор [имя\_класса::new] (11).

6.3.4 Дополнительные возможности

У стандартных функциональных интерфейсов есть еще интересные статические и дефолтные методы, позволяющие делать разные полезные вещи.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | IntPredicate isOdd = x -> x % 2 != 0;  IntPredicate isEven = isOdd.negate();  IntPredivate p1 = ..., p2 = ...;  IntPredicate p3 = p1.and(p2); |

- negate() получить обратный ему Predicate (1: isOdd);

- p1.and(p2)).построить конъюнкцию из 2-х Predicate (5: p1,p2);

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | Consumer<Object> printer =  System.out::println;  List<Object> objects = new ArrayList<>();  Consumer<Object> collection = objects::add;  Consumer<Object> combinedConsumer =  printer.andThen(collectior); |

2 Consumer (1: printer, 5: collection) можем скомбинировать в 3-ий Consumer (7: combinedConsumer), который вызовет сначала первое действие (8: printer), а затем второе (8: collectior).

(то есть в данном случае сначала напечатает объект в консоль (2), а потом добавит в список (5: objects::add))

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | DoubleUnaryOperator sqare = x -> x \* x;  DoubleUnaryOperator sin = Math::sin;  DoubleUnaryOperator complexFunction1 =  sin.andThen(square);  DoubleUnaryOperator complexFunction2 =  sin.compose(square); |

Похожим образом можно строить композиции функций при помощи методов andThen (5) и compose() (7). Разница между этими методами в порядке применения. Метод andThen() сделает нам функцию sin2(x), а метод compose() сделает нам функцию sin(x2).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | Comparator<Double> absoluteValueComparator =  (a, b) -> Double.compare(  Math.abs(a), Math.abs(b));  Comparator<Double> absoluteValueComparator2 =  Comparator.comparing(  Math::abs, Double::compare); |

**Компаратор.**

Вместо явного описания алгоритма 1-3 (сравнения 2-х double по их абсолютной величине) модно 5‑6 (вызвать метод comparing(); первый параметр - метод будет применяться к каждому значению перед тем как выполняется сравнение, и второй параметр — метод будет сравнивать результаты применения первого метода к исходным значениям, чтобы получить финальный результат)