



Современные технологии программирования

λ-выражения в Java 8

Функция как параметр

Во многих языках функцию можно передавать в качестве параметра

- Динамическое определение типа:
 - JavaScript, Lisp, Sceme, ...
- Строгая типизация:
 - Ruby, Scala, ...
- Функциональный подход позволяет писать более краткий и результативный код

Javascript:

```
var testStrings =  
    ["one", "two", "three", "four"];  
testStrings.sort(function(s1, s2) {  
    return(s1.length - s2.length);});
```

Основное преимущество: лаконичный и выразительный код

Java 7

```
button.addActionListener(  
    new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            doSomethingWith(e);  
        }  
    });
```

Java 8

```
button.addActionListener(e -> doSomethingWith(e));
```

Дополнительное преимущество: новый способ мышления

- Функциональный подход: многие классы задач решаются проще, код становится легким для чтения, что упрощает его дальнейшее сопровождение.
- Поддержка потоков: потоки являются обертками источников данных (массивы, коллекции, ...), которые используют лямбда-выражения.

ОСНОВНЫЕ МОМЕНТЫ

Вы пишете код, который похож на функцию

```
Arrays.sort(testStrings,  
            (s1, s2) -> s1.length() - s2.length());  
taskList.execute(() -> downloadSomeFile());  
someButton.addActionListener(  
    event -> handleButtonClick());  
double d = MathUtils.integrate(  
    x -> x*x, 0, 100, 1000);
```

И получаете экземпляр класса, который реализует интерфейс, который ожидается в данном случае.

Интерфейс содержит **ТОЛЬКО ОДИН** абстрактный метод

Такой интерфейс называется **функциональным** или **SAM**-интерфейсом (Single Abstract Method). Он является типом лямбда-выражения.

λ-выражение



- ~~• Анонимная функция~~
- ~~• Выражение описывающее анонимную функцию~~
- Выражение описывающее анонимную функцию, результатом исполнения которого является некоторый объект, реализующий требуемый функциональный интерфейс

Где используют λ -выражения

В переменной или параметре, где ожидается интерфейс с одним методом

```
public interface Blah {  
    String foo(String s);}
```

В коде, который использует интерфейс

```
public void someMethod(Blah b) {  
    ...  
    b.foo(...)  
    ...  
}
```

В коде, который вызывает интерфейс, можно использовать λ -выражение

```
String result = someMethod(s -> s.toUpperCase() + "!");
```

λ-выражение как аргумент метода

```
Arrays.sort(testStrings,  
            (s1, s2) -> s1.length() - s2.length());  
taskList.execute(() -> downloadSomeFile());  
someButton.addActionListener(  
    event -> handleButtonClick());  
double d = MathUtils.integrate(  
    x -> x*x, 0, 100, 1000);
```


λ-выражение как переменная

```
AutoCloseable c = () ->  
    cleanupForTryWithResources ();
```

```
Thread.UncaughtExceptionHandler handler =  
    (thread, exception) ->  
        doSomethingAboutException ();
```

```
Formattable f =  
    (formatter, flags, width, precision) ->  
        makeFormattedString ();
```

```
ContentHandlerFactory fact = mimeType ->  
    createContentHandlerForMimeType ();
```

Итоги: упрощение синтаксиса

Замена кода

```
new SomeInterface() {  
    @Override  
    public SomeType someMethod (аргументы)  
    {  
        тело  
    }  
}
```

на код

(аргументы) -> { тело }

Пример

Было

```
Arrays.sort(testStrings,  
            new Comparator<String>() {  
                public int compare(String s1, String s2) {  
                    return(s1.length() - s2.length());  
                }  
            });
```

Стало

```
Arrays.sort(testStrings,  
            (String s1, String s2) ->  
            { return(s1.length() - s2.length()); }  
            );
```

Сортировка строк по длине

Было

```
String[] testStrings =  
    {"one", "two", "three", "four"};  
...  
Arrays.sort(testStrings, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return(s1.length() - s2.length());  
    }  
});
```

Стало

```
Arrays.sort(testStrings,  
    (String s1, String s2) -> {  
        return(s1.length() - s2.length());  
    }  
);
```

Выведение типов

- В списке аргументов можно пренебречь указанием типов

- Общий вид λ -выражения

`(тип1 var1, тип2 var2 ...)` \rightarrow { тело метода }

- λ -выражение с выводением типов

`(var1, var2 ...)` \rightarrow { тело метода }

Сортировка строк по длине

Было

```
String[] testStrings =  
    {"one", "two", "three", "four"};  
...  
Arrays.sort(testStrings, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return(s1.length() - s2.length());  
    }  
});
```

Стало

```
Arrays.sort(testStrings,  
    (s1, s2) -> {  
        return(s1.length() - s2.length());  
    }  
);
```

Возвращаемое значение

- В теле метода используйте выражение, а не блок.
- Значение выражения будет возвращено.
- Если тип возвращаемого значения `void`, то метод ничего не вернет.

Было

```
(var1, var2 ...) -> { return выражение }
```

Стало

```
(var1, var2 ...) -> выражение
```

Сортировка строк по длине

Было

```
String[] testStrings =  
    {"one", "two", "three", "four"};  
...  
Arrays.sort(testStrings, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return(s1.length() - s2.length());  
    }  
});
```

Стало

```
Arrays.sort(testStrings, (s1, s2) ->  
    s1.length() - s2.length());
```


Скобки

Если метод зависит от одного аргумента, скобки можно опустить.

В таком случае тип аргумента не указывается.

Было

```
(varName) -> someResult()
```

Стало

```
varName -> someResult()
```

НОВЫЙ СИНТАКСИС

Было

```
button1.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent event) {  
        setBackground(Color.BLUE);  
    }  
});
```

Стало

```
button1.addActionListener(event ->  
    setBackground(Color.BLUE));
```

Использование значений

- Лямбда-выражения могут ссылаться на переменные, которые не объявлены как `final` (но значение таким переменным можно присвоить только один раз)
- Такие переменные называются эффективно финальными (их можно корректно объявить как `final`)
- Также можно ссылаться на изменяемые переменные экземпляра: `"this"` в лямбда-выражении ссылается на главный класс (не вложенный, который создается для лямбда-выражения)

Явное объявление

```
final String s = "...";  
doSomething(someArg -> use(s));
```

Эффективно финальная переменная

```
String s = "...";  
doSomething(someArg -> use(s));
```

Аннотация @Override

Какой смысл использовать аннотацию @Override?

```
public class MyServlet extends HttpServlet {  
    @Override  
    public void doGet(...) ... { ... }  
}
```

Корректный код будет работать и без @Override, но @Override

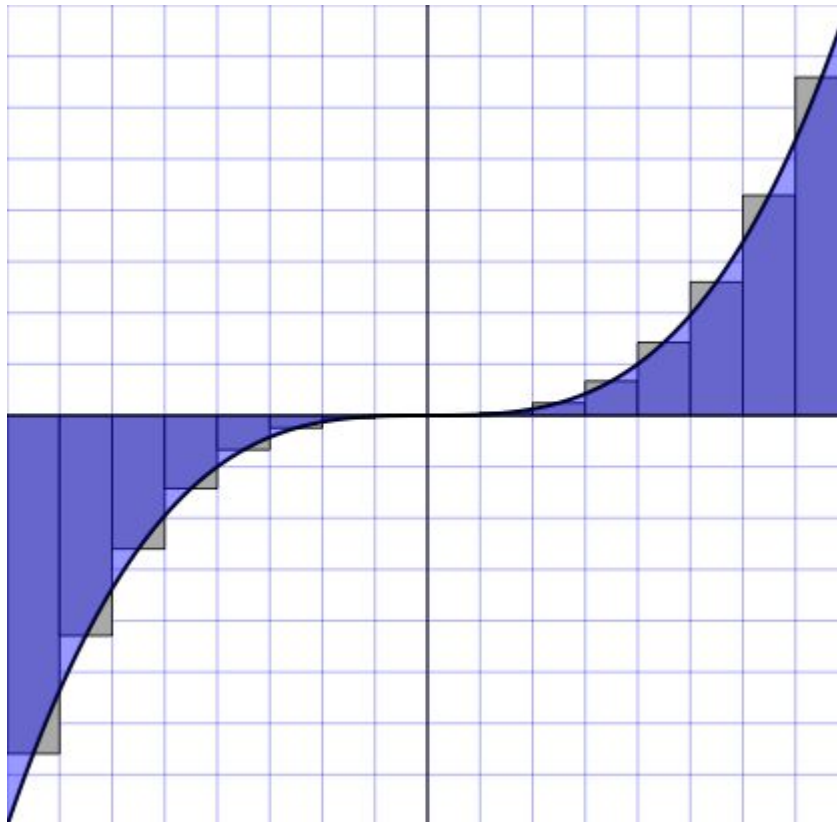
- Отслеживает ошибки во время компиляции
- Описывает суть метода
 - Сообщает остальным разработчикам, что данный метод из супер-класса, и в HttpServlet API описана его реализация

Аннотация `@FunctionalInterface`

- Отслеживает ошибки во время компиляции
 - Если разработчик добавит второй абстрактный метод в интерфейс, интерфейс не будет скомпилирован.
- Описывает суть интерфейса
 - Сообщает остальным разработчикам, что данный интерфейс будет использоваться с лямбда-выражениями
- Аннотация не обязательна

Пример. Численное интегрирование

- Обычное численное интегрирование методом средних прямоугольников



Пример. Численное интегрирование

- Использовать лямбда-выражения для интегрируемой функции.
 - Определить функциональный интерфейс с методом `double eval(double x)` для описания интегрируемой функции.
- Для проверки интерфейса во время компиляции и для объявления, что интерфейс функциональный и его можно использовать в лямбда-выражениях, используем аннотацию **@FunctionalInterface**

Интерфейс @Integrable

```
@FunctionalInterface  
public interface Integrable {  
    double eval(double x);  
}
```


Метод численного интегрирования

```
public static double integrate(Integrable function,  
                               double x1, double x2,  
                               int numSlices){  
  
    if (numSlices < 1) {  
        numSlices = 1;  
    }  
    double delta = (x2 - x1)/numSlices;  
    double start = x1 + delta/2;  
    double sum = 0;  
    for(int i=0; i<numSlices; i++) {  
        sum += delta * function.eval(start + delta * i);  
    }  
    return sum;  
}
```

Метод для тестирования

```
public static void integrationTest(Integrable function,
                                   double x1, double x2) {
    for(int i=1; i<7; i++) {
        int numSlices = (int)Math.pow(10, i);
        double result =
            integrate(function, x1, x2, numSlices);
        System.out.printf(
            "Для разбиения =%,10d результат = %, .8f%n",
            numSlices, result);
    }
}
```

Тестирование

```
integrationTest(x -> x*x, 10, 100);  
integrationTest(x -> Math.pow(x,3), 50, 500);  
integrationTest(x -> Math.sin(x), 0, Math.PI);  
integrationTest(x -> Math.exp(x), 2, 20);
```

Ссылка на методы

- Можно использовать ссылку `ИмяКласса :: имяСтатическогоМетода` или `имяПеременной :: методЭкземпляраКласса` в лямбда-выражениях
- Например, `Math :: cos` или `myVar :: myMethod`
- Это еще один способ задания функции, которая уже описана, в данном случае не нужно писать лямбда-выражение, вместо этого используйте ссылку на этот метод
- Функция должны соответствовать сигнатуре метода функционального интерфейса
- Тип определяется только из контекста

Пример. Численное интегрирование

Было

```
integrationTest(x -> Math.sin(x), 0, Math.PI);  
integrationTest(x -> Math.exp(x), 2, 20);
```

Стало

```
integrationTest(Math::sin(x), 0, Math.PI);  
integrationTest(Math::exp(x), 2, 20);
```

Пакет `java.util.function`

- Такие интерфейсы как `Integrable` очень широко используются.
- Поэтому в Java 8 нужны интерфейсы с более общим названием, который можно применять в подобных случаях.
- Пакет `java.util.function` определяет множество простых функциональных (SAM) интерфейсов.
- Они называются согласно аргументам и возвращаемым значениям.

Пакет `java.util.function`

- Например, можно заменить интерфейс `Integrable` на встроенный функциональный интерфейс `DoubleUnaryOperator`.
- Для того, чтобы узнать имя метода, нужно посмотреть API.
- Не смотря на то, что лямбда-выражения не ссылаются на имя метода, код, в котором используются лямбда-выражения должен ссылаться на соответствующие методы интерфейса.

Типизированные и обобщенные интерфейсы

- **Тип задан**

- Примеры (существует множество других интерфейсов)

- `IntPredicate (int in, boolean out)`
- `LongUnaryOperator (long in, long out)`
- `DoubleBinaryOperator(double in1, double in2, double out)`

- Пример

- `DoubleBinaryOperator f =
 (d1, d2) -> Math.cos (d1 + d2);`

- **Обобщенные**

- Также существуют обобщенные интерфейсы (`Function<T, R>`, `Predicate<T>`) с более широкой степенью применения

Пример. Численное интегрирование

Можно заменить в предыдущем примере

```
public static double integrate(Integrable function, ...)
{
... function.eval(...); ...
}
```

на следующий код

```
public static double integrate(DoubleUnaryOperator
function, ...) {
... function.applyAsDouble(...); ...
}
```

Пример. Численное интегрирование

- После этого можно удалить интерфейс **Integable**, т.к. **DoubleUnaryOperator** – функциональный (SAM) интерфейс, который содержит метод с точно такой же сигнатурой как у метода интерфейса **Integrable**.

Интерфейсы

`java.util.function`

- `java.util.function` содержит много интерфейсов различного назначения
- Например, простые интерфейсы: `IntPredicate`, `LongUnaryOperator`, `DoubleBinaryOperator`
- А также обобщенные
- `Predicate<T>` — аргумент `T`, возвращает `boolean`
- `Function<T, R>` — аргумент `T`, возвращает `R`
- `Consumer<T>` — аргумент `T`, ничего не возвращает (`void`)
- `Supplier<T>` — нет аргументов, возвращает `T`
- `BinaryOperator<T>` — аргументы `T` и `T`, возвращает `T`

Общий случай

- Если вы собираетесь создать функциональный интерфейс для лямбда-выражения, посмотрите документацию `java.util.function` и убедитесь, что можете использовать один из функциональных интерфейсов:
 - `DoubleUnaryOperator`, `IntUnaryOperator`, `LongUnaryOperator`
 - Аргумент `double/int/long`, возвращает такой же тип
 - `DoubleBinaryOperator`, `IntBinaryOperator`, `LongBinaryOperator`

Общий случай

- `DoublePredicate`, `IntPredicate`, `LongPredicate`
 - Аргумент `double/int/long`, возвращает `boolean`
- `DoubleConsumer`, `IntConsumer`, `LongConsumer`
 - Аргумент `double/int/long`, не возвращает значение (`void`)
- Обобщенные интерфейсы: `Function`, `Predicate`, `Consumer` и др.

Интерфейс Predicate

- `boolean test(T t)`
 - Позволяет задать «функцию» для проверки условия•
- Преимущество
 - Позволяет искать по коллекции элементы, которые соответствуют условию, написать гораздо более краткий код, чем без лямбда-выражений
- Пример синтаксиса

```
Predicate<Employee> matcher =  
    e -> e.getSalary() > 50000;  
if (matcher.test(someEmployee)) {  
    doSomethingWith(someEmployee);  
}
```

Пример. Без Predicate

Поиск сотрудника по имени

```
public static Employee findEmployeeByFirstName
    (List<Employee> employees,
     String firstName) {
    for(Employee e: employees) {
        if(e.getFirstName().equals(firstName)) {
            return(e);
        }
    }
    return(null);
}
```

Пример. Без Predicate

Поиск сотрудника по зарплате

```
public static Employee findEmployeeBySalary
    (List<Employee> employees,
     double salaryCutoff) {
    for(Employee e: employees) {
        e.getSalary() >= salaryCutoff) {
            return (e);
        }
    }
    return (null);
}
```


Рефакторинг 1

Поиск первого сотрудника, удовлетворяющего условию

```
public static Employee firstMatchingEmployee
    (List<Employee> candidates,
     Predicate<Employee> matchFunction) {
    for (Employee possibleMatch: candidates) {
        if (matchFunction.test(possibleMatch)) {
            return (possibleMatch) ;
        }
    }
    return (null) ;
}
```

Рефакторинг 1. Преимущества

- Теперь можно передать различные функции для поиска по разным критериям. Код более краткий и понятный.

```
firstMatchingEmployee (employees ,  
    e -> e.getSalary() > 500000) ;  
firstMatchingEmployee (employees ,  
    e -> e.getLastName() .equals ("...") ) ;  
firstMatchingEmployee (employees ,  
    e -> e.getId() < 10) ;
```

- Но код по-прежнему «привязан» к классу **Employee**

Рефакторинг 2

Поиск первого сотрудника, удовлетворяющего условию

```
public static <T> T firstMatch
    (List<T> candidates,
     Predicate<T> matchFunction) {
    for(T possibleMatch: candidates) {
        if(matchFunction.test(possibleMatch)) {
            return(possibleMatch);
        }
    }
    return(null);
}
```

Метод `firstMatch`

Предыдущий пример по-прежнему работает:

```
firstMatch(employees,  
           e -> e.getSalary() > 500000);  
firstMatch(employees,  
           e -> e.getLastName().equals("..."));  
firstMatch(employees,  
           e -> e.getId() < 10);
```

Метод `firstMatch`

Но также работают и более общие примеры кода

```
Country firstBigCountry = firstMatch(countries,  
    c -> c.getPopulation() > 1000000);  
  
Car firstCheapCar = firstMatch(cars,  
    c -> c.getPrice() < 15000);  
  
Company firstSmallCompany = firstMatch(companies,  
    c -> c.numEmployees() <= 50);  
  
String firstShortString = firstMatch(strings,  
    s -> s.length() < 4);
```

Интерфейс `Function`

- `R apply(T t)`
 - Позволяет задать «функцию», которая принимает аргумент `T` и возвращает `R`.
 - Интерфейс `BiFunction` работает аналогичным образом, но метод `apply` принимает два аргумента типа `T`.
- Преимущество
 - Позволяет преобразовать значение или коллекцию значений, написать гораздо более краткий код, чем без лямбда-выражений

Интерфейс Function

- Пример синтаксиса

```
Function <Employee, Double> raise =  
    e -> e.getSalary() + 1000;  
for (Employee employee: employees) {  
    employee.setSalary(raise.apply(employee));  
}
```

Пример. Без Function

Вычисление суммы зарплат сотрудников

```
public static int salarySum(List<Employee> employees)
{
    int sum = 0;
    for(Employee employee: employees) {
        sum += employee.getSalary();
    }
    return sum;
}
```


Пример. С Function

Вычисление суммы произвольных объектов

```
public static <T> int mapSum(List<T> entries,  
                             Function<T, Integer> mapper) {  
    int sum = 0;  
    for(T entry: entries) {  
        sum += mapper.apply(entry);  
    }  
    return sum;  
}
```

Интерфейс BinaryOperator

- `T apply(T t1, T t2)`
 - Позволяет задать «функцию», которая принимает два аргумента `T` и возвращает `T`
- Синтаксис

```
BinaryOperator <Integer> adder = (n1, n2) -> n1 + n2;  
// в правой части также можно записать  
//Integer::sum  
int sum = adder.apply(num1, num2);
```

Применение BinaryOperator

- Делает `mapSum` более гибкой
- Вместо

```
mapSum(List<T> entries, Function<T, Integer> mapper)
```

- Можно обобщить ее, передав оператор (который был жестко задан в методе `mapSum`):

```
mapCombined(List<T> entries,  
             Function<T, R> mapper,  
             BinaryOperator<R> combiner)
```

Интерфейс Consumer

- `void accept(T t)`
 - Позволяет задать «функцию», которая принимает аргумент `T` и выполняет некоторый побочный эффект
- Синтаксис

```
Consumer <Employee> raise =  
    e -> e.setSalary(e.getSalary() * 1.1);  
for(Employee employee: employees) {  
    raise.accept(employee);  
}
```

Применение Consumer

- Во встроенном метода `forEach` класса `Stream` используется интерфейс `Consumer`

```
employees.forEach (
```

```
e -> e.setSalary(e.getSalary() * 11/10) )
```

```
values.forEach (System.out::println)
```

```
textFields.forEach (field -> field.setText("")) )
```

Интерфейс `Supplier`

- `T get()`
 - Позволяет задать «функцию» без аргументов и возвращает `T`. и выполняет некоторый побочный эффект
- Синтаксис

```
Supplier <Employee> maker1 = Employee::new;  
Supplier <Employee> maker2 =  
    () -> randomEmployee();  
Employee e1 = maker1.get();  
Employee e2 = maker2.get();
```

Область видимости переменных

- Лямбда-выражения используют статические области действия переменных
- Выводы:
 - Ключевое слово **this** ссылается на внешний класс, а не на анонимный (тот, в который преобразуется лямбда-выражение)
 - Нет переменной **OuterClass.this**
 - До тех пор, пока лямбда внутри вложенного класса
 - Лямбда не может создавать новые переменные с такими же именами как у метода, вызвавшего лямбда
 - Лямбда может ссылаться (но не изменять) локальные переменные из окружающего кода
 - Лямбда может обращаться (и изменять) переменные экземпляра окружающего класса

Примеры

- Ошибка: повторное использование имени переменной

```
double x = 1.2;
```

```
someMethod(x -> doSomethingWith(x));
```

- Ошибка: повторное использование имени переменной

```
double x = 1.2;
```

```
someMethod(y -> { double x = 3.4; ... });
```

- Ошибка: лямбда изменяет локальную переменную

```
double x = 1.2;
```

```
someMethod(y -> x = 3.4);
```


Примеры

- Изменение переменной экземпляра

```
private double x = 1.2;  
public void foo() { someMethod(y -> x = 3.4); }
```

- Имя переменной в лямбда совпадает с именем переменной экземпляра

```
private double x = 1.2;  
public void bar() {  
    someMethod(x -> x + this.x);  
}
```

Ссылка на методы

- Ссылки на методы можно использовать в лямбда-выражениях.
- Если есть метод, сигнатура которого совпадает с сигнатурой абстрактного метода функционального интерфейса, можно использовать ссылку `ИмяКласса :: имяМетода`.

Ссылка на методы

| Вызов метода | Пример | Эквивалентное лямбда |
|--|--|---|
| <code>Класс :: статМетод</code> | <code>Math :: cos</code> | <code>x -> Math.cos(x)</code> |
| <code>переменная :: методЭкземпляраКласса</code> | <code>someString :: toUpperCase</code> | <code>() -> someString.toUpperCase ()</code> |
| <code>Класс :: методЭкземпляраКласса</code> | <code>String :: toUpperCase</code> | <code>s -> s.toUpperCase()</code> |
| <code>Класс :: new</code> | <code>Employee :: new</code> | <code>() -> new Employee()</code> |

Вызов метода экземпляра класса

- **переменная :: методЭкземпляраКласса**

- Создает лямбда-выражение, которое принимает то количество аргументов, которое указано в методе.

```
String test = "PREFIX:";
```

```
List<String> result1 = transform(strings, test::concat);
```

- **Класс :: методЭкземпляраКласса**

- Создает лямбда-выражение, которое принимает на один аргумент больше, чем соответствующий метод. Первый аргумент – объект, от которого вызывается метод, остальные аргументы – параметры метода.

```
List<String> result2=
```

```
    transform(strings, String::toUpperCase);
```

Методы по умолчанию

- В функциональных интерфейсах должен быть только один абстрактный метод. Этот метод и определяет лямбда-выражение. Но в интерфейсы Java 8 можно включать методы с реализацией, а также статические методы.
- Т.о. интерфейсы в Java 8 становятся похожими на абстрактные классы.

ИСХОДНЫЙ КОД `Function`

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);

    default <V> Function<V, R> compose
        (Function<? super V, ? extends T> before) {
        ...
    }

    default <V> Function<T, V> andThen(...) { ... }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

Методы, возвращающие лямбда

- Метод может возвращать лямбда-выражение (в действительности, объект, который реализует функциональный интерфейс).
 - В интерфейсах Predicate, Function, Consumer есть встроенные методы, возвращающие лямбда-выражение.

```
Predicate<Employee> isRich =  
    e -> e.getSalary() > 200000;  
Predicate<Employee> isEarly =  
    e -> e.getEmployeeId() <= 10;  
allMatches (employees, isRich.and(isEarly))
```

Методы интерфейса `Predicate`

- **`and`**

- В качестве аргумента принимает `Predicate`, возвращает `Predicate`, в котором метод `test` возвращает `true`, если оба исходных объекта `Predicate` возвращают `true` для заданных аргументов. Метод по умолчанию.

- **`or`**

- В качестве аргумента принимает `Predicate`, возвращает `Predicate`, в котором метод `test` возвращает `true`, если хотя бы один из исходных объектов `Predicate` возвращает `true` для заданных аргументов. Метод по умолчанию.

Методы интерфейса `Predicate`

- **`negate`**
 - Метод без аргументов. Возвращает `Predicate`, в котором метод `test` возвращает отрицание возвращаемого значения исходного объекта `Predicate`. Метод по умолчанию.
- **`isEqual`**
 - Принимает в качестве аргумента `Object`, возвращает `Predicate`, в котором метод `test` возвращает `true`, если объект `Predicate` эквивалентен аргументу `Object`. Статический метод.

Пример

```
Predicate<Employee> isRich =  
    e -> e.getSalary() > 200000;  
Predicate<Employee> isEarly =  
    e -> e.getEmployeeId() <= 10;  
System.out.printf("Состоятельные: %s.%n",  
    allMatches(employees, isRich));  
  
System.out.printf("Нанятые раньше: %s.%n",  
    allMatches(employees, isEarly));  
  
System.out.printf("Состоятельные и раньше нанятые: %s.%n",  
    allMatches(employees, isRich.and(isEarly)));  
  
System.out.printf("Состоятельные или раньше нанятые:  
    %s.%n",    allMatches(employees, isRich.or(isEarly)));  
  
System.out.printf("Не состоятельные: %s.%n",  
    allMatches(employees, isRich.negate()));
```

Методы интерфейса `Function`

- **`compose`**
 - `f1.compose(f2)` означает сначала выполнить `f2` и затем передать результат `f1`. Метод по умолчанию.
- **`andThen`**
 - `f1.andThen(f2)` означает сначала выполнить `f1` и затем передать результат `f2`. Т.е. `f2.andThen(f1)` это то же самое, что `f1.compose(f2)`. Метод по умолчанию.
- **`identity`**
 - Создает функцию, у которой метод `apply` возвращает аргумент без изменений. Статический метод.

Цепочки функций

- Можно заменить метод transform так, чтобы он принимал произвольное количество Function (вместо одного).

```
public static <T> List<T> transform2(List<T>
    origValues, Function<T,T> ... transformers) {
    Function<T,T> composedFunction
        = composeAll(transformers);
    return transform(origValues, composedFunction);
}
```

```
public static <T> Function<T,T> composeAll
    (Function<T,T> ... functions) {
    Function<T,T> result =
        Function.identity();
    for (Function<T,T> f: functions) {
        result = result.compose(f);
    }
    return result;
}
```

Пример

```
List<Double> nums = Arrays.asList(2.0, 4.0, 6.0, 8.0);
System.out.printf("Числа: %s.%n", nums);
Function<Double,Double> square = x -> x * x;
Function<Double,Double> half = x -> x / 2;
System.out.printf("square.compose(half): %s.%n",
    transform(nums, square.compose(half)));
System.out.printf("square.andThen(half): %s.%n",
    transform(nums, square.andThen(half)));
System.out.printf("half.andThen(square): %s.%n",
    transform(nums, half.andThen(square)));

// transform2
System.out.printf("square.compose(half): %s.%n",
    transform2(nums, square, half));
System.out.printf("identity: %s.%n",
    transform(nums, Function.identity()));
Function<Double,Double> round = Math::rint;
System.out.printf("округленный корень: %s.%n",
    transform(nums, round.compose(Math::sqrt)));
```

Пример

```
Числа: [2.0, 4.0, 6.0, 8.0].  
square.compose(half): [1.0, 4.0, 9.0, 16.0].  
square.andThen(half): [2.0, 8.0, 18.0, 32.0].  
half.andThen(square): [1.0, 4.0, 9.0, 16.0].  
square.compose(half): [1.0, 4.0, 9.0, 16.0].  
identity: [2.0, 4.0, 6.0, 8.0].  
округленный корень: [1.0, 2.0, 2.0, 3.0].
```

Методы интерфейса `Consumer`

- **`andThen`**

- `f1.andThen(f2)` возвращает `Consumer`, который передает аргумент в `f1` (т.е. его методу `accept`), после этого передает аргумент `f2`. Метод по умолчанию.

- **Различие методов в `Consumer` и `Function`**

- В метода `andThen` `Consumer` аргумент передается методу `accept` из `f1`, а затем **тот же аргумент** передается в `f2`.

- В методе `andThen` из `Function` аргумент передается методу `apply` из `f1`, а затем полученный результат передается в метод `apply` из `f2`.

Пример

```
List<Employee> myEmployees = Arrays.asList(  
    new Employee("Bill", "Gates", 1, 200000),  
    new Employee("Larry", "Ellison", 2, 100000));  
System.out.println("Сотрудники:");  
processEntries(myEmployees, System.out::println);  
Consumer<Employee> giveRaise =  
    e -> e.setSalary(e.getSalary() * 11 / 10);  
System.out.println("«Сотрудники после повышения:");  
processEntries(myEmployees,  
    giveRaise.andThen(System.out::println));
```

```
Сотрудники:  
Bill Gates [Employee#1 $200,000]  
Larry Ellison [Employee#2 $100,000]  
Сотрудники после повышения:  
Bill Gates [Employee#1 $220,000]  
Larry Ellison [Employee#2 $110,000]
```