JSR-335

Lambda expressions

Intro

Problem

Сложность решения некоторых простых задач в сравнении с функциональными языками программирования.

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

Solution

```
button.addActionListener(event -> System.out.println("Hello Wor
```

JSR-335: Lambda expressions

Lambda expressions

Functional Interface

```java
interface Operationable {
    int calculate(int x, int y);
}
```

Anonymous Classes

```java
public class LambdaApp {
    public static void main(String[] args) {
        Operationable op = new Operationable() {
            public int calculate(int x, int y) {
                return x + y;
            }
        };
        int z = op.calculate(20, 10);
        System.out.println(z);
    }
}
```

Example

```java
public class LambdaApp {
    public static void main(String[] args) {
        Operationable operation;
        operation = (x, y) -> x + y;

        int result = operation.calculate(10, 20);
        System.out.println(result);
    }
}
```

Steps for using Lambda Expression

```
// Defining reference to functional interface:
Operationable operation;

// Creating Lambda Expression
operation = (x, y) -> x + y;

// Using Lambda Expression
int result = operation.calculate(10, 20);
```

Example

```java
Operationable operation1 = (int x, int y)-> x + y;
Operationable operation2 = (int x, int y)-> x - y;
Operationable operation3 = (int x, int y)-> x * y;

System.out.println(operation1.calculate(20, 10));
System.out.println(operation2.calculate(20, 10));
System.out.println(operation3.calculate(20, 10));
```

Syntax Lambda Expressions

```java
(int a, int b) -> {
    return a + b;
}

(a, b) -> {
    return a + b;
}

() -> System.out.println("Hello World");

(String s) -> {
    System.out.println(s);
}

(s) -> System.out.println(s);

() -> 42;

() -> {
    return 3.1415;
```

Terminal Lambda Expressions

```java
interface Printable {
    void print(String s);
}

public class LambdaApp {
    public static void main(String[] args) {
        Printable printer = s -> System.out.println(s);
        printer.print("Hello Java!");
    }
}
```

Lambda Expressions and global variables

```java
public class LambdaApp {
    static int x = 10;
    static int y = 20;

    public static void main(String[] args) {
        Operation op = () -> {
            x = 30;
            return x + y;
        };
        System.out.println(op.calculate());
        System.out.println(x);
    }
}

interface Operation {
    int calculate();
}
```

Lambda Expressions and local variables

```java
public static void main(String[] args) {
    int n = 70;
    int m = 30;
    Operation op = () -> {
        // n = 100;
        return m + n;
    };
    // n = 100;
    System.out.println(op.calculate());
}
```

Generic Functional Interface

```java
public class LambdaApp {
    public static void main(String[] args) {
        Operationable<Integer> operation1 = (x, y) -> x + y;
        Operationable<String> operation2 = (x, y) -> x + y;

        System.out.println(operation1.calculate(20, 10));
        System.out.println(operation2.calculate("20", "10"));
    }
}

interface Operationable<T> {
    T calculate(T x, T y);
}
```

Lambda as parameters and results of methods

Lambda as parameters method

```java
public class LambdaApp {
    public static void main(String[] args) {
        Expression func = (n) -> n % 2 == 0;
        int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        System.out.println(sum(nums, func));
    }

    private static int sum(int[] numbers, Expression func) {
        int result = 0;
        for (int i : numbers) {
            if (func.isEqual(i))
                result += i;
        }
        return result;
    }
}

interface Expression {
    boolean isEqual(int n);
```

Method links as method parameters

```java
interface Expression {
    boolean isEqual(int n);
}

class ExpressionHelper {
    static boolean isEven(int n) {
        return n % 2 == 0;
    }

    static boolean isPositive(int n) {
        return n > 0;
    }
}

public class LambdaApp {
    public static void main(String[] args) {
        int[] nums = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
        System.out.println(sum(nums, ExpressionHelper::isEven
```

Links to constructors

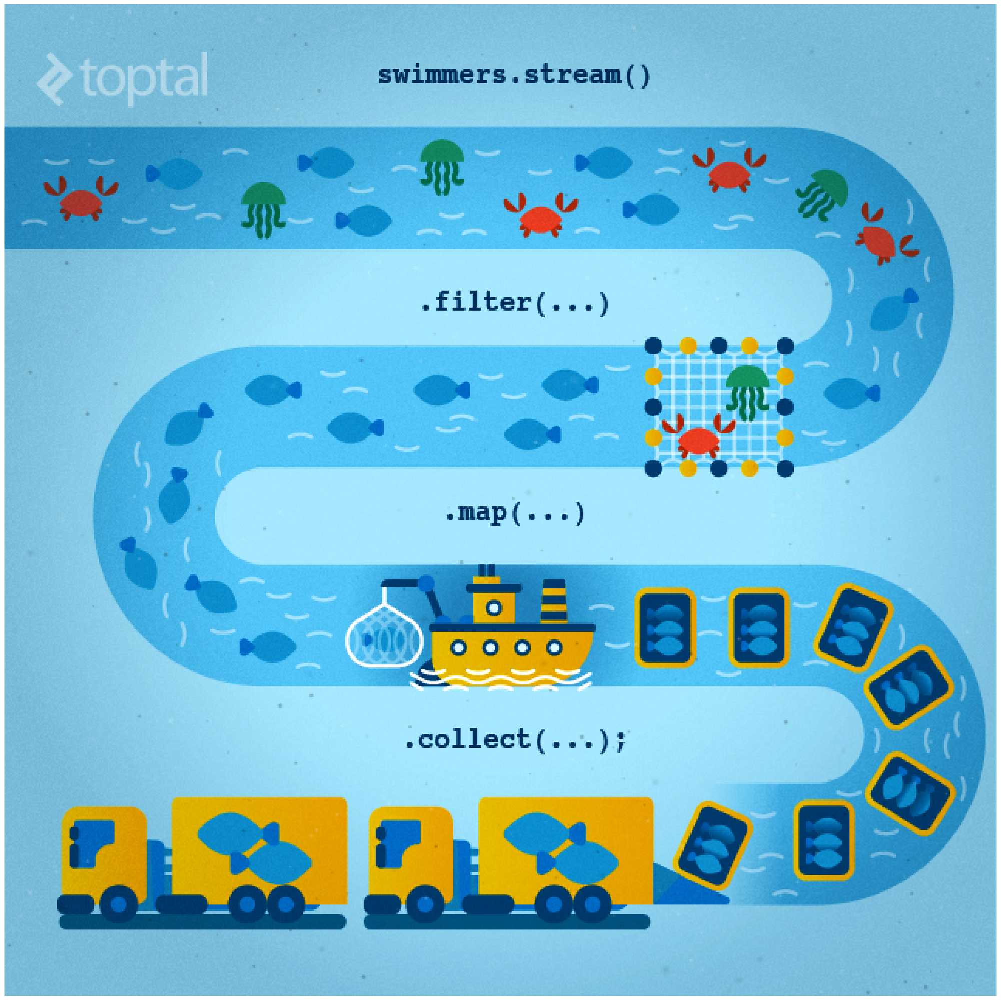
```java
public class LambdaApp {
    public static void main(String[] args) {
        UserBuilder userBuilder = User::new;
        User user = userBuilder.create("Tom");
        System.out.println(user.getName());
    }
}

interface UserBuilder {
    User create(String name);
}

class User {
    private String name;

    String getName() {
        return name;
    }

    User(String n) {
```

Lambda as a result of methods

```java
interface Operation {
    int execute(int x, int y);
}

public class LambdaApp {
    public static void main(String[] args) {
        Operation func = action(1);
        int a = func.execute(6, 5);
        System.out.println(a); // 11

        int b = action(2).execute(8, 2);
        System.out.println(b); // 6
    }

    private static Operation action(int number) {
        switch (number) {
            case 1:
                return (x, y) -> x + y;
            case 2:
                return (x, y) -> x - y;
```

Stream

Stream



BaseStream

Interface `BaseStream`

- `interface BaseStream<T , S extends BaseStream<T , S>>`

`BaseStream` Methods

- `close(): void`
- `isParallel(): boolean`
- `iterator(): Iterator<T>`
- `spliterator(): Spliterator<T>`
- `parallel(): S`
- `sequential(): S`
- `unordered(): S`

`BaseStream` Inheritors

- `Stream<T>`
- `IntStream`
- `DoubleStream`
- `LongStream`

`Stream<T>`

Intermediate methods

- `map(Function<? super T, ? extends R> mapper): Stream<R>`
- `filter(Predicate<? super T> predicate): Stream<T>`
- `sorted(): Stream<T>`
- `concat(Stream<? extends T> a, Stream<? extends T> b): Stream<T>`
- `distinct(): Stream<T>`

Intermediate methods

- `skip(long n): Stream<T>`
- `sorted(Comparator<? super T> comparator): Stream<T>`
- `takeWhile(Predicate<? super T> predicate): Stream<T>`
- `dropWhile(Predicate<? super T> predicate): Stream<T>`
- `limit(long maxSize): Stream<T>`

Terminal methods

- `forEach(Consumer<? super T> action): void`
- `allMatch(Predicate<? super T> predicate): boolean`
- `anyMatch(Predicate<? super T> predicate): boolean`
- `count(): long`
- `noneMatch(Predicate<? super T> predicate): boolean`
- `toArray(): Object[]`

Terminal methods

- `collect(Collector<? super T, A, R> collector): <R, A> R`
- `flatMap(Function<? super T, ? extends Stream<? extends R>> mapper): <R> Stream<R>`
- `findFirst(): Optional<T>`
- `findAny(): Optional<T>`
- `max(Comparator<? super T> comparator): Optional<T>`
- `min(Comparator<? super T> comparator): Optional<T>`

Creating `Stream`

Creating `Stream`

- `default Stream<E> stream`
- `default Stream<E> parallelStream`
- `Arrays.stream(T[] array)`
- `Stream.of(T..values)`

Optional

Methods

- `Optional<T>.empty(): <T>`
- `filter(Predicate<? super T> predicate): Optional<T>`
- `flatMap(Function<? super T,Optional<U>> mapper): Optional<U>`
- `get(): T`
- `ifPresent(Consumer<? super T> consumer): void`
- `isPresent(): boolean`

Methods

- `map(Function<? super T,? extends U> mapper): Optional<U>`
- `Optional<T>.of(T value): <T>`
- `Optional<T>.ofNullable(T value): <T>`
- `orElse(T other): T`
- `orElseGet(Supplier<? extends T> other): T`
- `<X extends Throwable> orElseThrow(Supplier<? extends X> exceptionSupplier): T`

Method `collect()`

`Collectors` Methods

- `toList(): List<T>`
- `toSet(): Set<T>`
- `toMap(): Map<K, U>`
- `toCollection(): Collection<T>`
- `groupingBy(Function<? super T, ? extends K>): Collector<T, ?, Map<K, List<T>>>`
- `partitioningBy(Predicate<? super T>): Collector<T, ?, Map<Boolean, List<T>>>`

Collectors Methods

- `counting()`
- `summing()`
- `maxBy(Comparator<? super T>)`
- `minBy(Comparator<? super T>)`
- `summarizing()`
- `mapping()`

Parallel Stream

Methods

- `parallel()`
- `sequential()`
- `forEachOrdered()`
- `unordered()`