**Встроенные функциональные интерфейсы в Java 8**

В JDK 8 вместе с самой функциональностью лямбда-выражений также было добавлено некоторое количество встроенных функциональных интерфейсов, которые мы можем использовать в различных ситуациях и в различные API в рамках JDK 8. В частности, ряд далее рассматриваемых интерфейсов широко применяется в Stream API - новом прикладном интерфейсе для работы с данными. Рассмотрим основные из этих интерфейсов:

* **Predicate<T> -** проверяет соблюдение некоторого условия
* **Consumer<T> -** выполняет некоторое действие над объектом, при этом ничего не возвращая
* **Function<T,R> -** представляет функцию перехода от объекта типа A к объекту типа B
* **Supplier<T> -** не принимает никаких аргументов, но должен возвращать что-то делающий объект
* **UnaryOperator<T> -** принимает в качестве параметра один объект и выполняет над ним какую/ие-то операцию/и
* **BinaryOperator<T> -** принимает в качестве параметра два объекта и выполняет над ними бинарную операцию (+,-,\*)

**Predicate<T>**

Функциональный интерфейс Predicate<T> проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T:

public interface Predicate<T> {

    boolean test(T t);

}

Например:

**import java.util.function.Predicate;**

**public class LambdaApp {**

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

**Predicate<Integer> isPositive = x -> x > 0;**

**System.out.println(isPositive.test(5)); // true**

**System.out.println(isPositive.test(-7)); // false**

**}**

**}**

**BinaryOperator<T>**

BinaryOperator<T> принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа T:

public interface BinaryOperator<T> {

    T apply(T t1, T t2);

}

Например:

**import java.util.function.BinaryOperator;**

**public class LambdaApp {**

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

**BinaryOperator<Integer> multiply = (x, y) -> x\*y;**

**System.out.println(multiply.apply(3, 5)); // 15**

**System.out.println(multiply.apply(10, -2)); // -20**

    }

}

**UnaryOperator<T>**

UnaryOperator<T> принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операций в виде объекта типа T:

public interface UnaryOperator<T> {

    T apply(T t);

}

Например:

**import java.util.function.UnaryOperator;**

**public class LambdaApp {**

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

**UnaryOperator<Integer> square = x -> x\*x;**

**System.out.println(square.apply(5)); // 25**

    }

}

**Function<T,R>**

Функциональный интерфейс Function<T,R> представляет функцию перехода от объекта типа T к объекту типа R:

public interface Function<T, R> {

    R apply(T t);

}

Например:

**import java.util.function.Function;**

**public class LambdaApp {**

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

**Function<Integer, String> convert = x-> String.valueOf(x) + " долларов";**

**System.out.println(convert.apply(5)); // 5 долларов**

**}**

**}**

**Consumer<T>**

Consumer<T> выполняет некоторое действие над объектом типа T, при этом ничего не возвращая:

public interface Consumer<T> {

    void accept(T t);

}

Например:

**import java.util.function.Consumer;**

**public class LambdaApp {**

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

**Consumer<Integer> printer = x-> System.out.printf("%d долларов \n", x);**

**printer.accept(600); // 600 долларов**

**}**

**}**

**Supplier<T>**

Supplier<T> не принимает никаких аргументов, но должен возвращать объект типа T:

public interface Supplier<T> {

    T get();

}

Например:

**import java.util.Scanner;**

**import java.util.function.Supplier;**

**public class LambdaApp {**

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

**Supplier<User> userFactory = ()->{**

**Scanner in = new Scanner(System.in);**

**System.out.println("Введите имя: ");**

**String name = in.nextLine();**

**return new User(name);**

**};**

**User user1 = userFactory.get();**

**User user2 = userFactory.get();**

**System.out.println("Имя user1: " + user1.getName());**

**System.out.println("Имя user2: " + user2.getName());**

**}**

**}**

**class User{**

**private String name;**

**String getName(){**

**return name;**

**}**

**User(String n){**

**this.name=n;**

**}**

**}**

Консольный вывод:

Введите имя:

Том

Введите имя:

Сэм

Имя user1: Том

Имя user2: Сэм