TECHIN

LAMBDA IŠRAIŠKOS

Jaroslav Grablevski

Funkcinis interfeisas

- Funkcinis interfeisas - tai interfeisas kuris turi tik vieną abstraktų metodą (tačiau gali turėti daugiau nei vieną numatytąjį (angl. default method), bei statinį metodą).

- Funkcinius interfeisus patartina pažymėti *@FunctionalInterface* anotacija. Ši anotacija suteikia papildomo aiškumo - detalizuoja interfeiso prasmę (funkcinis), bei leidžia kompiliatoriui užtikrinti, kad jis toks ir liks.

Ar tai funkcinis interfeisas?

```java
// OK (vienas abstraktus metodas)
@FunctionalInterface
public interface Functional {
    void execute();
}

// Kompiliavimo klaida (ne funkcinis interfeisas)
@FunctionalInterface
public interface Functional {
    void execute();
    void executeSomethingElse();
}
```

Ar tai funkcinis interfeisas?

Is This A Functional Interface?

```
@FunctionalInterface
public interface Runnable {
  public abstract void run();
}
```

Yes. There is only one abstract method

Ar tai funkcinis interfeisas?

```
@FunctionalInterface
public interface Predicate<T> {
  default Predicate<T> and(Predicate<? super T> p) {…};
  default Predicate<T> negate() {…};
  default Predicate<T> or(Predicate<? super T> p) {…};
  static <T> Predicate<T> isEqual(Object target) {…};
  boolean test(T t);
}
```

Yes. There is still only one abstract method

Ar tai funkcinis interfeisas?

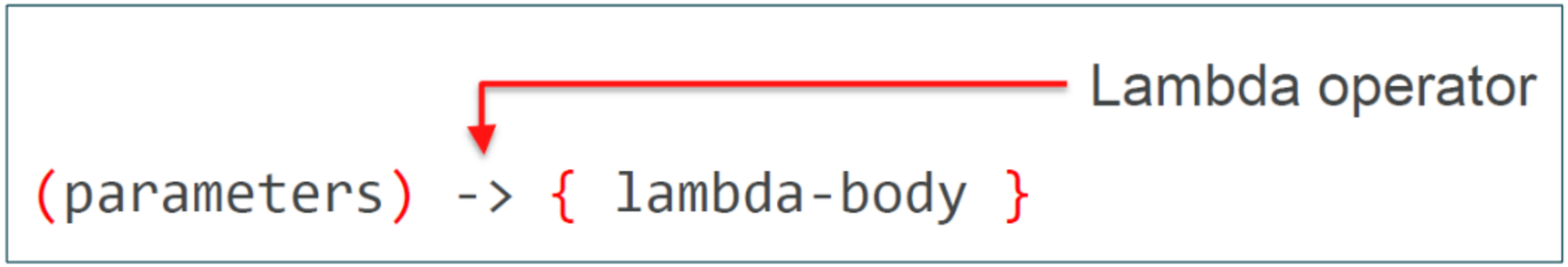
```
@FunctionalInterface
public interface Comparator {
  // Static and default methods elided
  int compare(T o1, T o2);
  boolean equals(Object obj);
}
```

Therefore only one abstract method

The `equals(Object)` method is implicit from the `Object` class

Lambda išraiškos

- Lambda išraiškos - tai anoniminės funkcijos, kurios realizuoja vieną funkcinį interfeisą - java kompiliatorius pats sugeneruoja funkcinio interfeiso realizaciją pagal pateiktą lambda išraišką.
- Tas pats rezultatas pasiekiamas naudojant anonimines klases, tačiau lambda išraiškos yra glaustesnės.
- Lambda išraiškos parametrai turi atitikti parametrus, kurie nurodyti funkcinio interfeiso abstrakčiame metode.



Lambda išraiškos (funkcinis interfeisas)

```
@FunctionalInterface
interface Func<T, R> {

    R apply(T val);

    static void doNothingStatic() {

    }

    default void doNothingByDefault() {

    }
}
```

Lambda išraiškos (funkcinio interfeiso implementacija)

```
//anonymous class
Func<Long, Long> square = new Func<Long, Long>() {
    @Override
    public Long apply(Long val) {
        return val * val;
    }
};

//lambda expression
Func<Long, Long> square2 = val -> val * val;

long val = square.apply(10L); // the result is 100L
long val2 = square2.apply(10L); // the result is 100L
```

Lambda išraiškos

```
// Java 7 (anoniminė klasė)
Arrays.sort(testStrings, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return (s1.length() - s2.length());
    }
});

// Java 8 (lambda)
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

Lambda išraiškos

```
@FunctionalInterface
interface Expression<T> {
    T evaluate();
}

@FunctionalInterface
interface ResultConsumer<T> {
    void consume(T t);
}

public static void main(String[] args) {
    ResultConsumer<Integer> consumer = (result) -> System.out.println(result);
    Expression<Integer> producer = () -> 2 + 2;

    consumer.consume(producer.evaluate());
}
```

Lambda išraiškos. Sintaksė

```
//be parametrų
() -> System.out.println("Zero parameter lambda");

//vienas parametras
param -> System.out.println("One parameter: " + param);

//du parametrai
(p1, p2) -> p1 + p2;

//kai reikia nurodyti tipus
(Integer p1, Integer p2) -> p1 + p2

// multiple statments
(param) -> {
    System.out.println("param: " + param);
    return param * 2;
}
```

Lambda išraiškos. Metodo parametras.

```
public static void acceptInterface(Function<Integer, Integer> f) {
    System.out.println(f.apply(10));
}
```

```
// it returns the next value
Function<Integer, Integer> f = (x) -> x + 1;

acceptInterface(f); // it prints 11

// or even without a reference
acceptInterface(x -> x + 1); // the result is the same: 11
```

Lambda išraiškos. Closure.

- In the body of a lambda expression, it's possible to capture values from a context where the lambda is defined. This technique is called **closure**.
- It's possible only if a context variable has a keyword final or it's effectively final, i.e. variable can't be changed. Otherwise, an error happens.

```
final String hello = "Hello, ";
Function<String, String> helloFunction = (name) -> hello + name;

System.out.println(helloFunction.apply("John"));
System.out.println(helloFunction.apply("Anastasia"));
```

Metodų nuorodos (Method references)

Tipas	Pavyzdys
Nuoroda į statinį metodą	ContainingClass::staticMethodName
Nuoroda į konkretaus objekto metodą	containingObject::instanceMethodName
Nuoroda i konkretaus tipo objekto metodą	ContainingType::methodName
Nuoroda į konstruktorių	ClassName::new

Metodų nuorodos

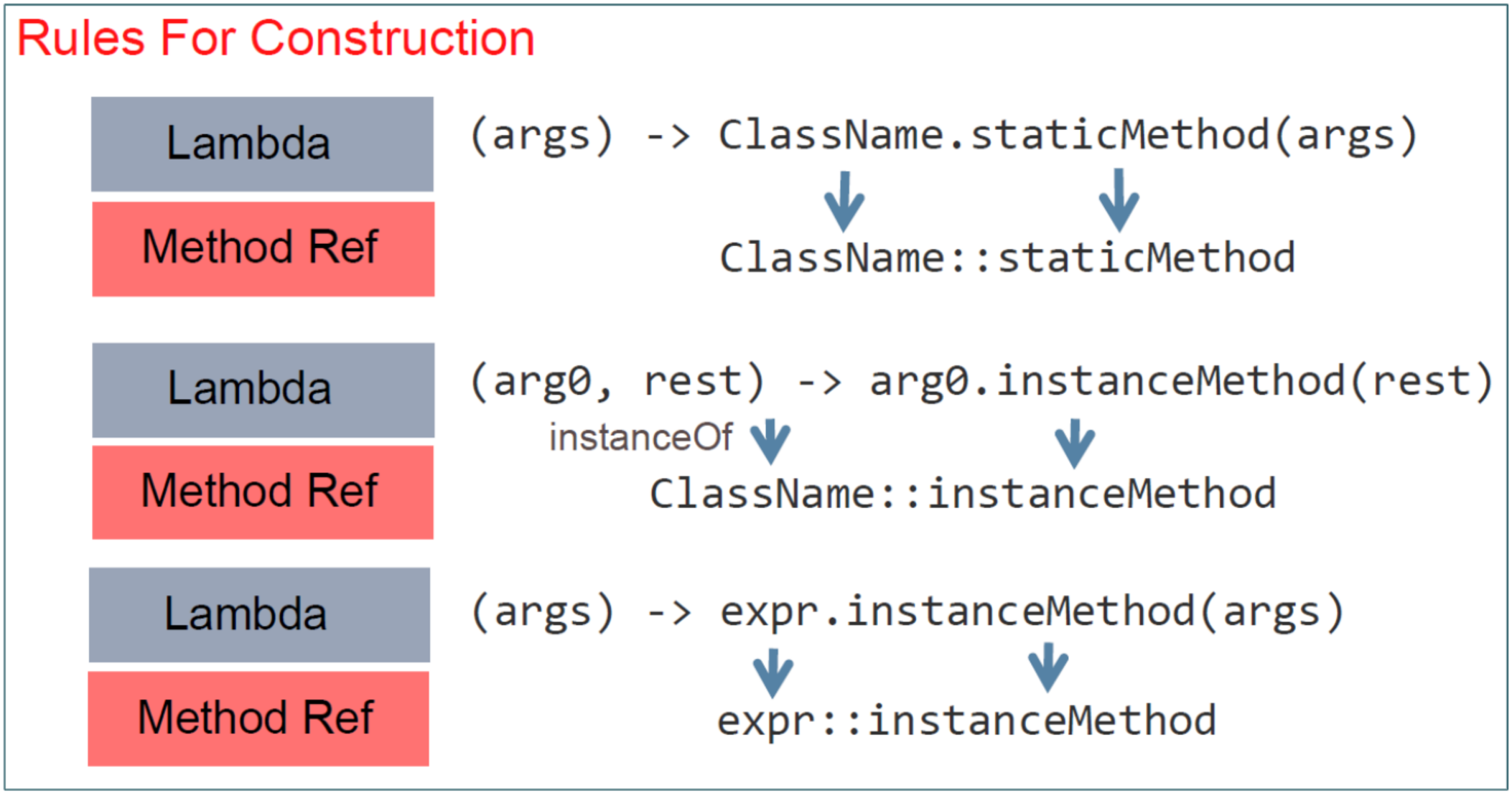
```
// Naudojame metodų nuorodas

    List<Integer> numbers = Arrays.asList(9, 5, 7, 1, 3, 8);
    numbers.sort(Integer::compare);
    numbers.forEach(System.out::println);


// Naudojame lambda išraiškas

    List<Integer> numbers2 = Arrays.asList(9, 5, 7, 1, 3, 8);
    numbers2.sort((x, y) -> Integer.compare(x, y));
    numbers2.forEach((number) -> System.out.println(number));
```

Metodų nuorodos



Metodų nuorodos

Examples

Lambda

```
(String s) -> Integer.parseInt(s);
```

Method Ref

```
Integer::parseInt
```

Lambda

```
(String s, int i) -> s.substring(i)
```

Method Ref

```
String::substring
```

Lambda

```
Axis a -> getLength(a)
```

Method Ref

```
this::getLength
```

Nuoroda į konstruktorių

```
Factory<List<String>> f = () -> new ArrayList<String>();
```

```
Factory<List<String>> f = ArrayList<String>::new;
```

Method reference

```
//Reference to a static method
Function<Double, Double> sqrt = Math::sqrt;
Function<Double, Double> sqrt2 = x -> Math.sqrt(x);

sqrt.apply(100.0d); // the result is 10.0d

//Reference to an instance method of an existing object
Scanner scanner = new Scanner(System.in); // IO scanner

Supplier<String> lineReader = scanner::nextLine; //method reference
Supplier<String> lineReader2 = () -> scanner.nextLine();

String firstLine = lineReader.get();
String secondLine = lineReader.get();
```

Method reference

```
//Reference to an instance method of an object of a particular type
Function<Long, Double> converter = Long::doubleValue;
Function<Long, Double> converter2 = val -> val.doubleValue();

converter.apply(100L); // the result is 100.0d
converter.apply(200L); // the result is 200.0d

//Reference to a constructor
Supplier<String> generator = String::new;
Supplier<String> generator2 = () -> new String();
```

Pagrindiniai funkciniai interfeisai

- **java.util.function** package
- Well defined set of general purpose functional interfaces
  - All have only one abstract method
  - Lambda expressions can be used wherever these types are referenced
  - Used extensively in the Java class libraries
  - Especially with the Stream API

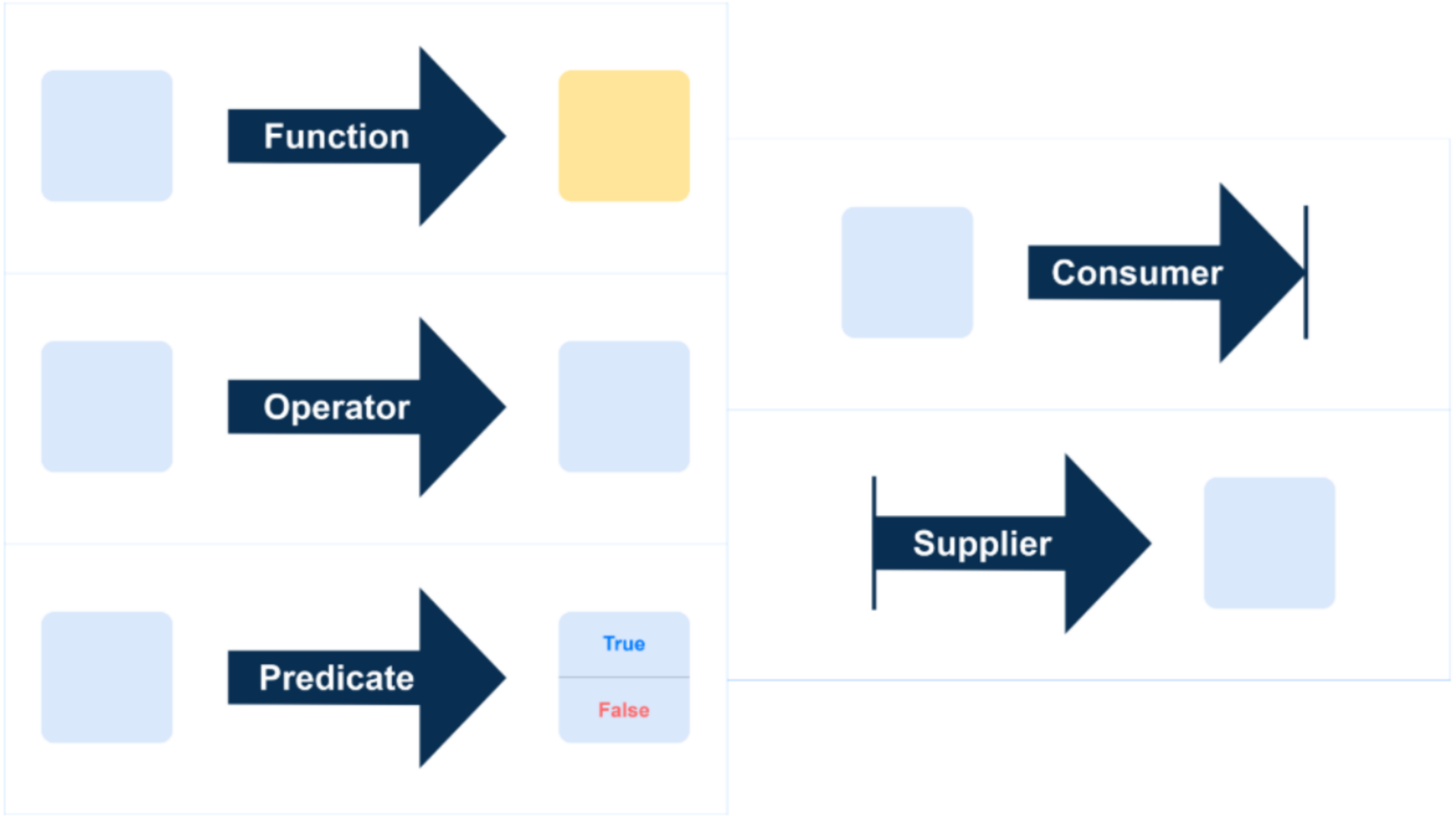
Pagrindiniai funkciniai interfeisai

- **functions** that accept arguments and produce results;
- **operators** that produce results of the same type as their arguments (a special case of function);
- **predicates** that accept arguments and return boolean values (boolean-valued function);
- **suppliers** that accept nothing and return values;
- **consumers** that accept arguments and return no result.

Pagrindiniai funkciniai interfeisai

Funkcinis interfeisas	Operacija	Aprašymas
Function<T, R>	R apply(T t)	Pritaiko nurodytą funkciją argumentui t, kurio tipas T ir grąžina rezultatą, kurio tipas R.
Predicate<T>	boolean test(T t)	Predikatas, dažniausiai naudojamas filtravimo operacijose. Paprastai, jei operacija grąžina rezultatą true, tai reikšmė tenkina filtravimo sąlygą.
Consumer<T>	void accept(T t)	Tai operacija, kuri priima argumentą ir negrąžina jokio rezultato. Paprastai generuojamas koks nors šalutinis efektas (rašoma į failą, spausdinama į konsolę ir pan.).
Supplier<T>	T get()	Operacija, kuri nepriklausomai nuo parametrų grąžina parametrizuojamo tipo T rezultatą.

Pagrindiniai funkciniai interfeisai



Pagrindiniai funkciniai interfeisai

Funkcinis interfeisas	Operacija	Aprašymas
Function<T, R>	R apply(T t)	Pritaiko nurodytą funkciją argumentui t, kurio tipas T ir grąžina rezultatą, kurio tipas R.
Predicate<T>	boolean test(T t)	Predikatas, dažniausiai naudojamas filtravimo operacijose. Paprastai, jei operacija grąžina rezultatą true, tai reikšmė tenkina filtravimo sąlygą.
Consumer<T>	void accept(T t)	Tai operacija, kuri priima argumentą ir negrąžina jokio rezultato. Paprastai generuojamas koks nors šalutinis efektas (rašoma į failą, spausdinama į konsolę ir pan.).
Supplier<T>	T get()	Operacija, kuri nepriklausomai nuo parametrų grąžina parametrizuojamo tipo T rezultatą.

Pagrindiniai funkciniai interfeisai

Interfeisas	Lambda išraiškos pavyzdys
Function<T,R>	Student s -> s.getName()
UnaryOperator<T>	String s -> s.toLowerCase()
BiFunction<T,U,R>	(String name, Student s) -> new Teacher(name, s)
Supplier<T>	() -> createLogMessage()
Consumer<T>	String s -> System.out.println(s)
BiConsumer<T,U>	(k, v) -> System.out.println("key:" + k + ", value:" + v)
Predicate<T>	Student s -> s.graduationYear() == 2011
BiPredicate<T,U>	(path, attr) -> String.valueOf(path).endsWith(".js") && attr.size() > 1024

Functions

```
// String to Integer function
Function<String, Integer> converter = Integer::parseInt;
converter.apply("1000"); // the result is 1000 (Integer)

// String to int function
ToIntFunction<String> anotherConverter = Integer::parseInt;
anotherConverter.applyAsInt("2000"); // the result is 2000 (int)

// (Integer, Integer) to Integer function
BiFunction<Integer, Integer, Integer> sumFunction = (a, b) -> a + b;
sumFunction.apply(2, 3); // it returns 5 (Integer)
```

Operators

```
// Long to Long multiplier
UnaryOperator<Long> longMultiplier = val -> 100_000 * val;
longMultiplier.apply(2L); // the result is 200_000L (Long)

// int to int operator
IntUnaryOperator intMultiplier = val -> 100 * val;
intMultiplier.applyAsInt(10); // the result is 1000 (int)

// (String, String) to String operator
BinaryOperator<String> appender = (str1, str2) -> str1 + str2;
appender.apply("str1", "str2"); // the result is "str1str2"
```

Predicates

```
// Character to boolean predicate
Predicate<Character> isDigit = Character::isDigit;
isDigit.test('h'); // the result is false (boolean)

// int to boolean predicate
IntPredicate isEven = val -> val % 2 == 0;
isEven.test(10); // the result is true (boolean)
```

Suppliers

```
Supplier<String> stringSupplier = () -> "Hello";
stringSupplier.get(); // the result is "Hello" (String)

BooleanSupplier booleanSupplier = () -> true;
booleanSupplier.getAsBoolean(); // the result is true (boolean)

IntSupplier intSupplier = () -> 33;
intSupplier.getAsInt(); // the result is 33 (int)
```

Consumers

```
// it prints a given string
Consumer<String> printer = System.out::println;
printer.accept("!!!"); // It prints "!!!"
```

JDK8 metodai kurie naudoja lambdas

Metodas	Lambda išraiškos pavyzdys
Iterable.forEach(Consumer c)	myList.forEach(s -> System.out.println(s));
Collection.removeIf(Predicate p)	myList.removeIf(s -> s.length() == 0);
List.replaceAll(UnaryOperator o)	myList.replaceAll(String::toUpperCase);
List.sort(Comparator c)	myList.sort((x, y) -> x.length() – y.length());
logger. finest(Supplier<String> msgSupplier)	logger.finest(() -> createComplexMessage());

JDK8 metodai kurie naudoja lambdas (Comparator)

```
messages.sort((m1, m2) -> m1.getCreated().compareTo(m2.getCreated()));

messages.sort(Comparator.comparing(Message::getCreated).reversed());

messages.sort(Comparator.comparing(Message::getLikes)
                        .reversed()
                        .thenComparing(Message::getFrom));
```