Wyrażenia lambda

W języku C/C++ chcąc dostosować działanie funkcji, algorytmów lub bibliotek najczęściej dostarczane są wskaźniki do funkcji.

Przykład – qsort

```
#include <stdio.h>
#include <stdlib.h>

int values[] = { 88, 56, 100, 2, 25 };

int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

int main () {
   int n;
   qsort(values, 5, sizeof(int), cmpfunc);
   for( n = 0 ; n < 5; n++ ) {
      printf("%d ", values[n]);
   }
   return 0;
}
```

Inną metodą parametryzacji (tylko C++) może

- być dostarczenie klasy ze statycznymi metodami jako argumentu parametryzacji szablonu
- użycie obiektów funkcyjnych

```
#include <list>
#include <algorithm>

class Dump
{
public:
     void operator()(int k){cout<<k<<" ";}
};

void main()
{
     list<int> li;
     for(int i=0;i<10;i++)
     {li.push_back(i);li.push_front(i);}
     for_each(li.begin(),li.end(),Dump());
     cout<<endl;
}
```

W języku Java podobną rolę pełnią obiekty klas realizujących pewien założony interfejs. Jeżeli ma być zachowane podobieństwo do wskaźników do funkcji i obiektów funkcyjnych – ten interfejs powinien mieć dokładnie jedną funkcję i wtedy nazywany jest interfejsem funkcyjnym (ang. *functional interface*).

W odróżnieniu od C++ funkcja interfejsu nie musi się jakoś specjalnie nazywać (w C++ oczekuje się, że funkcją jest `operator()`).

Przykład – algorytm całkowania metodą trapezów

```
public interface DoubleFunction {
    double evaluate(double x);
}
```

```
public class Integrator {
    double compute(double min,double max, DoubleFunction f){
        double delta=(max-min)/1000;
        double sum=0;
        double f1=f.evaluate(min);
        for(double x = min+delta;x<=max+1e-5;x+=delta){
            double f2=f.evaluate(x);
            sum+=(f1+f2)/2*delta;
            f1=f2;
        }
        return sum;
    }
}
```

Algorytm **metody trapezów** polega na podziale pola pod krzywą na pionowe słupki o niewielkiej szerokości $delta$ i zsumowaniu ich pól. Słupki te są obróconymi trapezami o podstawach $f(x)$ i $f(x + delta)$ i wysokości $delta$, stąd nazwa metody.

Obliczamy całkę funkcji kwadratowej

Przykład 1 – statyczna klasa definiująca funkcję

```
static class Foo implements DoubleFunction{
    @Override
    public double evaluate(double x) {
        return x * x - 2 * x +1;
    }
}
static void test1(){
double r = new Integrator().compute(0,10,new Foo());
System.out.println(r);
}
```

Wynik: 243.3334999999916

Sprawdzamy przeprowadzając symboliczne całkowanie

http://live.sympy.org/

```
>>> integrate(x**2 -2* x + 1, x)
```

$$\frac{x^3}{3} - x^2 + x$$

```
>>> integrate(x**2 -2* x + 1, (x,0,10))
...
```

$$\frac{730}{3}$$

Wynik wyznaczony symbolicznie: 730/3 = 243.333333

Przykład 2 – lokalna anonimowa klasa

```
static void test2(){
    DoubleFunction f = new DoubleFunction() {
        @Override
        public double evaluate(double x) {
            return x * exp(-3*x);
        }
    };
    double r = new Integrator().compute(0,10,f);
    System.out.println(r);
}
```

Wynik: 0.11110277815244185

```
>>> integrate(x*exp(-3*x))
```

$$\frac{1}{9}(-3x-1)\,e^{-3x}$$

```
>>> integrate(x*exp(-3*x),(x,0,10))
```

$$-\frac{31}{9e^{30}}+\frac{1}{9}$$

Wynik wyznaczony symbolicznie: 0.1111111111111

Przykład 3 – lokalna klasa anonimowa zdefiniowana bezpośrednio przy wywołaniu funkcji

```
static void test3(){
    double r = new Integrator().compute(0,10,
    new DoubleFunction() {
        @Override
        public double evaluate(double x) {
            return x*sin(x/5);
        }
    });
    System.out.println(r);
}
```

Wynik: 43.53977813969754

```
>>> integrate(x*sin(x/5),x)
```

$$-5x\cos\left(\frac{x}{5}\right)+25\sin\left(\frac{x}{5}\right)$$

```
>>> integrate(x*sin(x/5),(x,0,10))
```

$$-50\cos(2)+25\sin(2)$$

Wynik wyznaczony symbolicznie: 43.539777498

Przykład 4 – Do funkcji przekazywane jest wyrażenie lambda

```
static void test4(){
    double r
    = new Integrator().compute(0,10, (x) -> exp(-3*x) );
    System.out.println(r);
}
```

Wynik: 0.3333583329583101

```
>>> integrate(exp(-3*x))
```

$$-\frac{1}{3}e^{-3x}$$

```
>>> integrate(exp(-3*x),(x,0,10))
```

$$-\frac{1}{3e^{30}}+\frac{1}{3}$$

Wynik: 0.33333333333333

Wyrażenia lambda pozwalają na przekazywanie jako argumentów metod kodu funkcji, podobnie jak w przypadku tradycyjnych wskaźników do funkcji lub obiektów funkcyjnych C++.

Dają podobny efekt, jak jawne przekazywanie obiektów klas anonimowych realizujących pewien interfejs (z jednym wyjątkiem, interfejs zaimplementowany w klasach anonimowych może obejmować kilka metod).

Wyrażenie lambda w wyniku kompilacji przekształcane jest w obiekt anonimowej klasy implementującej interfejs funkcjonalny (czyli zawierający dokładnie jedną metodę).

Z kontekstu ustala się, jaki interfejs ma być implementowany – w tym jaka jest sygnatura metody, zwracana wartość i generowane wyjątki.

Składnia wyrażeń lambda

`parameter-list -> body`

`parameter-list` – lista formalnych parametrów oddzielonych przecinkami ujętych w nawiasy

`body` – pojedyncze wyrażenie lub instrukcja blokowa.

Lista parametrów może zawierać typy danych, ale często są one pomijane, ponieważ można ustalić je z kontekstu.

Przykład

```
interface InsertIntoTable {
    void assign(double[]x, int y,double z);
}
```

```
InsertIntoTable assign1
= (double[]a, int b,double c) -> {a[0]=b;a[1]=c;};

InsertIntoTable assign2
= (a,b,c) -> {a[0]=b;a[1]=c;};
```

Jeżeli **lista ma jeden parametr**, nawiasy można pominąć

```
DoubleConsumer dc = p->System.out.println(p);
```

`Java.util.function.DoubleConsumer` to biblioteczny interfejs definiujący metodę:
`void accept(double value).`

Jeżeli **ciało lambda jest pojedynczym wyrażeniem**, wtedy typ wyrażenia musi być zgodny z typem funkcji zdefiniowanej w interfejsie.

```
interface IntBinaryFunction{
    int process (int a, int b);
}

void call(IntBinaryFunction ibf, int x, int y){
    System.out.println(ibf.process(x,y));
}
```

```
IntBinaryFunction ibf = (a,b)->(a-b)*(a+b);

call((a,b)->(a*a +b*b),5,3);
```

Jeżeli **ciało wyrażenia lambda jest instrukcją blokową**, a określona w interfejsie funkcja jest typu różnego od `void`, wewnątrz instrukcji blokowej musi być zwracana wartość.

```
IntBinaryFunction ibf = (a,b)->{
    int prod=1;
    for(int i=0;i<b;i++)prod*=a;
    return prod;
};
```

Zastosowanie wyrażeń lambda w funkcjach kontenerów

W interfejsie kontenerów przewidziano możliwość dostarczenia własnej funkcji użytkownika (obiektu realizującego interfejs funkcjonalny) jako parametru kilku wybranych metod:

sort

`void sort(Comparator<? super E> c)` – sortuje zawartość stosując do porównania elementów obiekt klasy implementującej interfejs `Comparator`

```
public interface Comparator<T> {
    int compare(T var1, T var2);
...
}
```

forEach

`void forEach(Consumer<? super T> action)` – wykonuje dla każdego elementu kontenera akcję interfejsu `Consumer`

```
public interface Consumer<T> {
    void accept(T var1);
...
}
```

replaceAll

`void replaceAll(UnaryOperator<E> operator)` – zastępuje każdy element wartością zwróconą przez interfejs `UnaryOperator` (który dziedziczy po `Function`)

```
public interface Function<T, R> {
    R apply(T var1);
...
}
```

removeIf

`boolean removeIf(Predicate<? super E> filter)` – usuwa elementy, dla których predykat (czyli funkcja zwracająca `boolean`) jest prawdziwy.

```
public interface Predicate<T> {
    boolean test(T var1);
...
}
```

Przykład

```
static void convertStrings(){
  String txt = "Jeżeli na ciało nie działa " +
  "żadna siła lub siły działające równoważą sie to ciało "+
  "to pozostaje w spoczynku lub porusza się ruchem " +
  "jednostajnym prostoliniowym";
//    List<String> list = Arrays.asList(txt.split("\\s+"));
  List<String> list = new
    ArrayList(Arrays.asList(txt.split("\\s+")));

  list.removeIf(p->p.startsWith("s"));
  list.replaceAll(p->p.toUpperCase());
  list.sort((s1,s2)->s1.compareTo(s2));
  list.forEach(p-> System.out.print(p+" "));
}
```

Wynik: `CIAŁO CIAŁO DZIAŁA DZIAŁAJĄCWE JEDNOSTAJNYM JEŻELI LUB LUB NA NIE PORUSZA POZOSTAJE PROSTOLINIOWYM RUCHEM RÓWNOWAŻĄ TO TO W ŻADNA`

Odwołania do metod

Aby jeszcze bardziej skrócić wyrażenia lambda wprowadzono nowy element składni – odwołania do metod (ang. *method reference*).

Jeżeli ciało wyrażenia lambda obejmuje wyłącznie wywołanie metody bez podania jej parametrów, wówczas całe wyrażenie może zostać zastąpione przez:

- `reference::methodName` dla metod obiektu wskazywanego `reference`
- `Class::methodName` dla metod klasy (statycznych)
- `Class::methodName` dla metody obiektu dostarczonego jako argument wywołania metody interfejsu

Podczas kompilacji zostanie utworzony obiekt anonimowej klasy realizującej założony interfejs. W ciele metody znajdzie się wywołanie metody obiektu lub metody statycznej.

Przykład

```
static class BeReferenced {
    static String doubler(String p) {
        return p+p;
    }
    String firstChar(String p){
        return p.substring(0,1);
    }
}
```

Wywołanie

```
list.replaceAll(BeReferenced::doubler);
```

zastąpi każdy element podwojonym tekstem. Utworzona zostanie anonimowa klasa z odwołaniem do statycznej metody.

Z kolei

```
list.replaceAll(new BeReferenced()::firstChar);
```

przekaże do anonimowego obiektu referencję `ref` zwróconą przez **`new`** `BeReferenced()`, a następnie w funkcji `accept(String t)` klasy implementującej interfejs `Consumer` wywoła `ref.firstChar(t)`.

Odwołania do metod klas Java API

Jest to najczęstszy przypadek zastosowania odwołań do metod.

Dla wcześniejszego przykładu:

```
  list.removeIf(p->p.startsWith("s"));
  list.replaceAll(p->p.toUpperCase());
  list.sort((s1,s2)->s1.compareTo(s2));
  list.forEach(p-> System.out.print(p+" "));
```

sekwencja przetwarzania listy może zostać przepisana jako:

```
①list.removeIf(p->p.startsWith("s"));
②list.replaceAll(String::toUpperCase);
③list.sort(String::compareTo);
//    list.forEach(p-> System.out.print(p+" "));
④list.forEach(System.out::println);
```

① Nie da się zastąpić referencją metody, ponieważ jest potrzebny parametr – tu `"s"`,

② Zastąpiono referencją metody `String::toUpperCase`. Na liście przechowywane są obiekty typu `String`. Stąd wywołanie jest równoważne:

```
list.replaceAll(new UnaryOperator<String>() {
    @Override
    public String apply(String s) {
        return s.toUpperCase();
    }
});
```

Kompilator generuje "opakowanie" wywołania metody klasy `String`.

③ Zastąpiono referencją metody `String::compareTo`. Wywołanie jest równoważne:

```
list.sort(new Comparator<String>() {
      @Override
      public int compare(String s, String t1) {
          return s.compareTo(t1);
      }
  }
);
```

④ Co prawda, `println` to nie to samo, co `print(p+" ")`, ale efekt jest w miarę podobny.

Analogicznie można przekazywać odwołania do metod obiektów innych typów przechowywanych w kontenerach.

Przykład

```
static void sortDoubles(){
    List<Double> list=
      Arrays.asList(1.5, 2.1, 3.4, -5.2, 16.1, 48.);
//      list.sort((x,y)->x.compareTo(y));
      list.sort(Double::compareTo);
//      list.forEach(x -> System.out.println(x));
      list.forEach(System. out::println);
}
```

Kontener zawierający obiekty klasy użytkownika

Załóżmy, że kontener przechowuje obiekty klasy `Person`.

```
static class Person {
    String name;
    double height;
    Person(String name,double height){
        this.name = name;
        this.height = height;
    }
}
```

Zamierzamy posortować listę osób, według wzrostu. Interfejs `Comparator` wymaga implementacji funkcji zwracającej wartość typu `int`. Nie możemy więc po prostu odjąć wartości pól `height`.

Zamiast tego należy wywołać funkcję statyczną `Double.compare()`.

```
static void sortPersons(){
    List<Person> list = new ArrayList<>();
    list.add(new Person("Jan",178));
    list.add(new Person("Maria",169));
    list.add(new Person("Stanisław",187));
    list.add(new Person("Anna",164));
    list.sort((p,p1)->Double.compare(p.height,p1.height));
    list.forEach(
      p-> System.out.println(p.name+" "+p.height));
}
```

Wynik

```
Anna 164.0
Maria 169.0
Jan 178.0
Stanisław 187.0
```

Docelowy typ klasy dla wyrażenia lambda

Określenie docelowej klasy dla wyrażenia lambda jest przeprowadzane na podstawie kontekstu wywołania. W szczególności, dla każdego wyrażenia, nawet identycznego, będą generowane kolejne anonimowe klasy (mogące także różnić się interfejsami).

Przykład

```
interface PersonTester{
    boolean check(Person x);
}

static void targetTyping(){
①  Predicate<Person> pred1
    = x->x.name.endsWith("a") && x.height < 170;
②  Predicate<Person> pred2
    = x->x.name.endsWith("a") && x.height < 170;
③  PersonTester pred3
    = x->x.name.endsWith("a") && x.height < 170;
  Person p = new Person("Zofia",168);
  System.out.println(pred1.getClass()+ ":"+pred1.test(p));
  System.out.println(pred2.getClass()+ ":"+pred2.test(p));
  System.out.println(pred3.getClass()+ ":"+pred3.check(p));
}
```

Wynik:

```
class wyklady.lambada.Persons$$Lambda$3/1989780873:true
class wyklady.lambada.Persons$$Lambda$4/1480010240:true
class wyklady.lambada.Persons$$Lambda$5/1828972342:true
```

① ② ③ Za każdym razem tworzona jest nowa anonimowa klasa, mimo że wyrażenie lambda jest identyczne

① ② Mimo, że docelowy interfejs jest ten sam tworzone są kolejne klasy wewnętrzne

Widzialność zmiennych otaczającej instancji

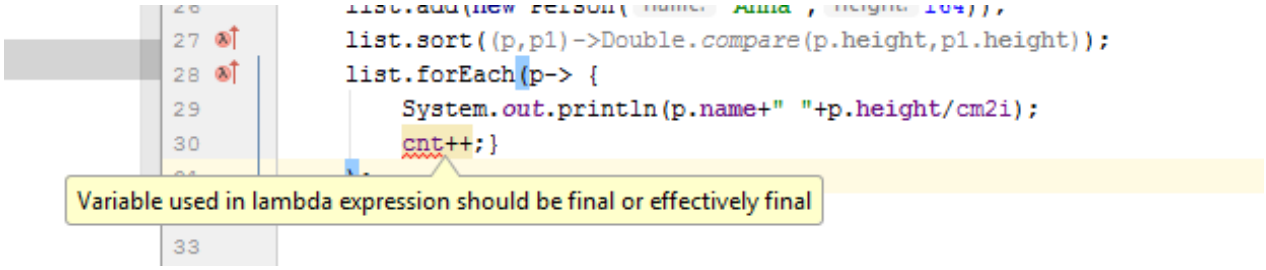
Anonimowe klasy powstałe w wyniku kompilacji wyrażeń lambda są klasami wewnętrznymi. Pojawia się więc pytanie – czy mają one dostęp do lokalnych zmiennych zdefiniowanych w funkcji oraz atrybutów otaczającej instancji?

Lokalne zmienne metody

```
void listPersons(){
    double cm2i = 2.51;
    int cnt=0;
    List<Person> list = new ArrayList<>();
    list.add(new Person("Jan",178));
    list.add(new Person("Stanisław",187));
    list.add(new Person("Maria",169));
    list.add(new Person("Anna",164));
    list.sort((p,p1)->Double.compare(p.height,p1.height));
    list.forEach(p-> {
①       System.out.println(p.name+" "+p.height/cm2i);
②       cnt++;}
    );
}
```

① w wyrażeniu lambda można odwołać się do zmiennej zdefiniowanej w funkcji, w której kontekście wyrażenie jest zdefiniowane

② nie jest możliwa jednak modyfikacja wartości zmiennej (traktowana jest jak zmienna finalna)



Atrybuty klasy zewnętrznej

W wyniku kompilacji wyrażeń lambda zdefiniowanych w niestatycznym kontekście tworzone są obiekty klas wewnętrznych. Mają one pełne prawa dostępu do zmiennych i metod otaczającej instancji.

```
public class Persons {
. . .
① int cnt=0;
  void listPersons(){
    double cm2i = 2.51;
    List<Person> list = new ArrayList<>();
    list.add(new Person("Jan",178));
    list.add(new Person("Stanisław",187));
    list.add(new Person("Maria",169));
    list.add(new Person("Anna",164));
    list.sort((p,p1)->Double.compare(p.height,p1.height));
    list.forEach(p-> {
      System.out.println(p.name+" "+p.height/cm2i);
②    cnt++;}
    );
  }

  public static void main(String[] args) {
    Persons p = new Persons();
    p.listPersons();
③   System.out.println("cnt="+p.cnt);
  }
}
```

① - `cnt` to atrybut klasy zewnętrznej (otaczającej instancji)

② - atrybut `cnt` może zostać zmodyfikowany w wyrażeniu lambda

③ - zgodnie z oczekiwaniami, `cnt` ma wartość 4

```
Anna 65.33864541832669
Maria 67.33067729083666
Jan 70.91633466135458
Stanisław 74.50199203187252
cnt=4
```

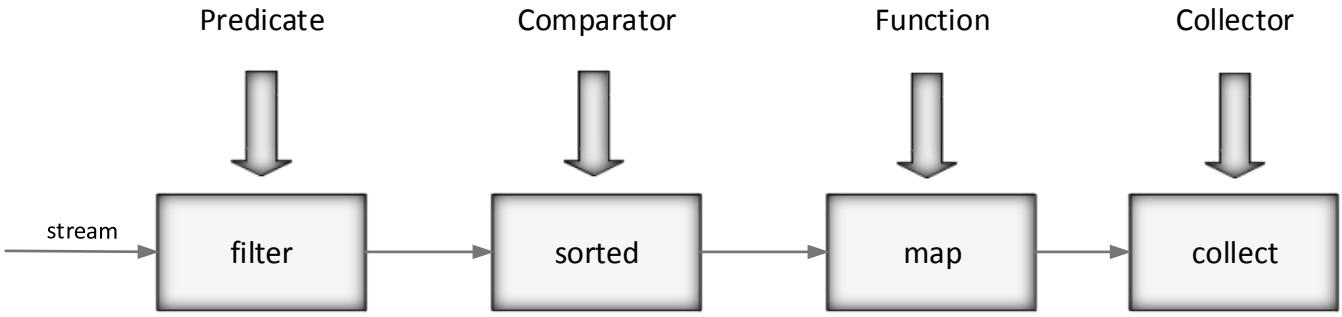
Strumienie

W Java 1.8 wprowadzono *strumienie* - nowy mechanizm mający na celu usprawnienie przetwarzania danych. W odróżnieniu od kolekcji, których dane cały czas są przechowywane w pamięci, strumienie otrzymują dane z zewnętrznego źródła (najczęściej jednak jakiegoś kontenera) i wykonują na nich operacje. Podczas przetwarzania w pamięci może być umieszczona jedynie część danych. Strumienie mogą być także przetwarzane równolegle.

Na strumieniu można wykonywać typowe operacje:

- filtrowania danych -- `filter()`
- sortowania – `sorted()`
- zamiany jednych danych w drugie – `map()`
- agregacji – `reduce()`, `collect()`, `max()`, itd.

Służą do tego zaimplementowane gotowe funkcje, które zazwyczaj zwracają referencję do strumienia, stąd można je łączyć w łańcuchy przetwarzania (potoki).



Specyfikując potok definiuje się kolejne operacje przetwarzania i podaje ich parametry w postaci klas realizujących przewidziany interfejs. Stąd typowymi argumentami operacji są wyrażenia lambda.

Ostatnia operacja potoku (terminalna) służy zwykle do agregacji danych lub zbierania ich i umieszczania w kontenerze założonego typu.

Przykład

Załóżmy, że przetwarzamy listę osób:

```
List<Person> list = new ArrayList<>();
list.add(new Person("Jan", 178));
list.add(new Person("Stanisław", 187));
list.add(new Person("Maria", 169));
list.add(new Person("Anna", 164));
list.add(new Person("Zofia", 167));
list.add(new Person("Jakub", 184));
```

Poniższy kod wybiera z kolekcji kobiety , sortuje je według wzrostu, następnie gromadzi wartości wzrostu na liście i zwraca listę.

```
List<Double> hlist =
①   list.stream()
②   .filter(p->p.name.endsWith("a"))
③   .sorted((p1,p2)->Double.compare(p1.height,p2.height))
④   .map(p->p.height)
⑤   .collect(Collectors.toList());

hlist.forEach(System.out::println);
```

① - zwraca strumień

② - filtruje elementy spełniające warunek przekazany jako predykat (wybiera kobiety – imiona kończą się na `"a"`)

③ - sortuje według wzrostu

④ - odwzorowuje obiekty klasy `Person` w `double`

⑤ - zbiera dane do listy za pomocą obiektu klasy `Collector`. Ostatecznie efektem przetwarzania jest zwracana lista.

Aby osiągnąć analogiczny efekt (i kolejność przetwarzania) w Java < 1.8 należałoby zaimplementować kod podobny do poniższego:

```
List<Person> tmp = new ArrayList<>();
for(Person p:list){
    if(p.name.endsWith("a"))tmp.add(p);
}
tmp.sort(new Comparator<Person>() {
    @Override
    public int compare(Person person, Person t1) {
        return Double.compare(person.height,t1.height);
    }
});
List<Double> hlist = new ArrayList<>();
for(Person p:tmp){
    hlist.add(p.height);
}
```

Różnice pomiędzy strumieniami a tradycyjną organizacją przetwarzania

- Podczas korzystania ze strumieni nie jest konieczna iteracja po elementach.
- Strumienie opisują sposób przetwarzania danych w sposób deklaratywny – tzn. deklarują typ, parametry i kolejność operacji.
- Dane ze źródła są kolejno dostarczane do potoku, jeżeli nie jest to konieczne – nie są akumulują się na danym etapie przetwarzania (ang. *lazy evaluation*).
- Warunkiem uruchomienia przetwarzania jest pojawienie się w potoku etapu terminalnego (kolektora)

Przykład (bez sortowania)

```
List<Double> hlist = list.stream()
        .filter(p->{
            System.out.println("filter@"+p);
            return p.name.endsWith("a");
        })
        .map(p->{
            System.out.println("map@"+p);
            return p.height;
        })
        .collect(Collectors.toList());
```

Wynik:

```
filter@Jan:178.0
filter@Stanisław:187.0
filter@Maria:169.0
map@Maria:169.0
filter@Anna:164.0
map@Anna:164.0
filter@Zofia:167.0
map@Zofia:167.0
filter@Jakub:184.0
```

Jak widać, kolejne elementy przechodzą zdefiniowaną sekwencję operacji i nie są po drodze gromadzone. W przypadku mężczyzn ścieżka jest krótsza – obiekty są odfiltrowywane.

Przykład – szukamy, jaki jest maksymalny wzrost wśród mężczyzn

```
list.stream()
        .filter(p->!p.name.endsWith("a"))
        .map(p->p.height)
        .max(Double::compare).
        ifPresent(System.out::println);
```

Funkcja `max()` zwraca `OptionalDouble` – wartość, która może być obecna lub nie (będzie nieobecna jeśli strumień, który dotarł do `max()` był pusty).

Przykład – zbieranie wyników do mapy, przetwarzanie strumienia mapy.

```
Map<Integer,Double> f =
    Arrays.asList(2,5,3,8,-12,45,-8,23,45,67,11,80)
      .stream()
      .distinct()
      .collect(Collectors.toMap(x -> x,x->Math.exp(-3*x)));

f.entrySet().
  stream().
  forEach(e-> System.out.printf(Locale.US, "f(%d)=%g\n",
                                e.getKey(),e.getValue()));
```

Przetwarzanie równoległe

Użycie równoległych strumieni pozwala na podział operacji pomiędzy wątki procesora. Potencjalnie, dzięki temu możliwe jest przyspieszenie przetwarzania, szczególnie dla dużej liczby danych, małego obciążenia, dużej liczby rdzeni procesora. Wydajność zależy też od charakteru wykonywanych operacji i tego, jak zostały zaimplementowane.

Obliczmy średnią 100 000 000 wygenerowanych losowo liczb

Kod sekwencyjny

```
static final int SIZE = 100000000;

public static void seqentialMean(){
    double [] array = new double[SIZE];
    for(int i=0;i<SIZE;i++){
        array[i]= Math.random()*SIZE/(i+1);
    }
    double t1 = System.nanoTime()/1e6;
    OptionalDouble mean = Arrays.stream(array).average();
    double t2 = System.nanoTime()/1e6;
    System.out.printf(Locale.US,
    "SEQ t2-t1=%f mean=%f\n",t2-t1,mean.getAsDouble());
}
```

Kod wykonywany równolegle

```
public static void parallelMean(){
    double [] array = new double[SIZE];
    for(int i=0;i<SIZE;i++){
        array[i]= Math.random()*SIZE/(i+1);
    }
    double t1 = System.nanoTime()/1e6;
    OptionalDouble mean = Arrays.stream(array)
      .parallel()
      .average();
    double t2 = System.nanoTime()/1e6;
    System.out.printf(Locale.US,
    "PAR t2-t1=%f mean=%f\n",t2-t1,mean.getAsDouble());
}
```

Dla porównania – generacja liczb losowych wewnątrz strumienia

```
static int i=0;
public static void parallelGenerateMean(){
    double t1 = System.nanoTime()/1e6;
    OptionalDouble mean = DoubleStream
      .generate(()->Math.random()*SIZE/++i)
      .limit(SIZE).parallel().average();
    double t2 = System.nanoTime()/1e6;
    System.out.printf(Locale.US,"PAR GEN t2-t1=%f
mean=%f\n",t2-t1,mean.getAsDouble());
}
```

W tym przypadku wyrażenie lambda wywołane w statycznym kontekście modyfikuje statyczną zmienną `i`.

Przykładowe wyniki dla 100 000 000

```
SEQ t2-t1=568.239378 mean=10.061352
PAR t2-t1=108.765394 mean=9.145111
PAR GEN t2-t1=35205.529144 mean=9.996682
```

Przykładowe wyniki dla 1 000 000

```
SEQ t2-t1=105.443914 mean=7.218113
PAR t2-t1=25.109826 mean=7.243354
PAR GEN t2-t1=331.822340 mean=7.622229
```

Przykładowe wyniki dla 100 000

```
SEQ t2-t1=88.077055 mean=6.043196
PAR t2-t1=7.983214 mean=6.378580
PAR GEN t2-t1=29.833604 mean=6.013130
```

Przykładowe wyniki dla 1 000

```
SEQ t2-t1=84.267616 mean=3.196165
PAR t2-t1=5.625056 mean=3.868378
PAR GEN t2-t1=3.474491 mean=4.479806
```

W tym konkretnym przykładzie przetwarzanie równoległe pozwala na 4-15 krotne przyspieszenie obliczeń (dość stary procesor i7, 8 rdzeni).

Dla dużej liczby danych ich generacja w ramach przetwarzania zdefiniowanego przez potok jest wolna (ale nie zużywa pamięci).

Podsumowanie

- Wyrażenia lambda są kompilowane do postaci obiektu anonimowej klasy realizującej założony interfejs funkcjonalny.
- Pozwalają na znaczne skrócenie kodu w stosunku do jawnie zdefiniowanych klas anonimowych
- Typ docelowy wyrażeń lambda ustalany jest na podstawie kontekstu wywołania (wartość po lewej stronie, formalny parametr funkcji)
- Strumienie pozwalają na deklaratywną specyfikację etapów przetwarzania
- Strumienie są tak zaprojektowane, aby używać w nich wyrażeń lambda. Parametrami funkcji przetwarzania strumieni są interfejsy, do których są kompilowane wyrażenia lambda.
- Równoległe przetwarzanie wewnątrz strumieni może znacznie skrócić czas przetwarzania.