

Wyrażenia lambda i interfejsy funkcyjne

Czego się dowiesz

- Czym są wyrażenia lambda,
- czym są interfejsy funkcyjne,
- przykłady wykorzystania interfejsów Function, Consumer, Predicate

Wstęp

Java od zawsze była prezentowana jako szandarowy przykład języka obiektowego, tzn. wszystko ma być obiektem i już (oprócz typów prostych). W Javie 8 zaczęto wprowadzać koncepcje programowania funkcyjnego, które w ostatnich latach jest coraz bardziej doceniane. Programowanie funkcyjne jest początkowo ciężkie do zrozumienia dla kogoś, kto rozpoczynał programowanie w języku obiektowym czy proceduralnym, nie jest ono do końca naturalne w rozumieniu tego, że program jest po prostu pewnym skończonym ciągiem instrukcji, które zmieniają stan obiektów i zmiennych w programie.

W programowaniu funkcyjnym skupiamy się na wyniku poprzez jego opis za pomocą matematycznej funkcji. Języki funkcyjne dostarczają nam narzędzi, które pozwalają na opis problemu i pozwalają nie martwić się jak właściwie coś jest wykonywane.

W Javie 8 wprowadzono elementy programowania funkcyjnego, które są przydatne przede wszystkim w przetwarzaniu kolekcji, czyli np. list. Pomagają także zastępować klasy anonimowe prostymi wyrażeniami, które są po prostu bardziej czytelne. Nie oznacza to oczywiście, że od teraz wszędzie gdzie tylko się da należy używać w Javie programowania funkcyjnego, jednak pokażę kilka zastosowań, gdzie staje się to po prostu przydatne i znacząco skraca zapis. Na początku zaczniemy jednak od podstawowych pojęć i mechanizmów.

Wyrażenia Lambda

Wyrażenie lambda, to taka konstrukcja w języku Java, która przyjmuje dowolne argumenty lub nie przyjmuje ich wcale i może zwracać na ich podstawie pewien wynik.

W rozumieniu matematycznym funkcję rozumiemy, np jako:

$$f(x) = x * x$$

czyli funkcję potęgującą dowolną liczbę. W Javie zapiszemy ją jako wyrażenie lambda takiej postaci:

(int x) -> x*x;

Widzimy tutaj trzy elementy:

- **(int x)** - argumenty funkcji podane w nawiasach okrągłych,
- **->** - strzałka, czyli operator wskazujący, że jest to wyrażenie lambda,
- **x*x** - wyrażenie, ciało funkcji.

Możliwy jest także zapis w nieco innej formie.

1. Wykorzystanie słowa kluczowego return

(int x) -> return x*x;

2. Pomińnięcie typów argumentów funkcji.

Język Java jest silnie typowany, czyli typ zmiennych musi być znany na etapie kompilacji. Przekazując argumenty funkcji kompilator będzie najczęściej w stanie wydedukować ich typ na podstawie wykonywanego wyrażenia, więc przy argumentach możemy pominąć podawanie typu i zapisać krótko:

(x) -> x*x;

3. Dodatkowe lub zbędne nawiasy.

Możemy wykonać bardziej złożone instrukcje w ciele funkcji, a także pominąć nawiasy okrągłe przy jej argumentach:

```
x -> { if(x>0) return x*x; else return 0; }
```

przy czym nie muszą one być zapisane w jednym wierszu.

Na tym etapie wyrażenia lambda wydają się bezużyteczne, ale wyobraź sobie teraz, że taką funkcję możemy przekazać np. do kolekcji i zostanie ona wykonana na każdym elemencie naszej listy. Będzie to znaczna oszczędność czasu, a także podniesie to czytelność kodu.

Interfejsy funkcyjne

Jeżeli chcesz w Javie przekazać wyrażenie lambda jako argument metody, czy konstruktora, albo chcesz je przypisać zmiennej, to musisz najpierw zdefiniować interfejs funkcyjny (eng. functional interface) lub skorzystać z jednego z dostępnych interfejsów funkcyjnych w bibliotece JDK.

Interfejsem funkcyjnym nazywamy interfejs, który posiada tylko jedną metodę abstrakcyjną.

W Javie 8 zdefiniowano interfejsy funkcyjne dla najpopularniejszych zastosowań, pełną listę możesz znaleźć **pod tym linkiem**, poniżej wymieniono te, które pokażemy na przykładach:

- **Consumer<T>** - posiada metodę **accept(T t)** - przyjmuje argument typu T, ma za zadanie wykonać pewną operację i nie zwraca wyniku,
- **Function<T, R>** - posiada metodę **apply(T t)** - reprezentuje funkcję przyjmującą argument typu T i zwracającą argument typu R,
- **Predicate<T>** - posiada metodę **test(T t)** - przyjmuje argument typu T i zwraca wartość typu boolean,
- **Supplier<T>** - posiada metodę **get()** - tworzy nowy obiekt typu T.

Interfejs Function

Najprostszym przykładem wykorzystania funkcji może być transformacja pewnego napisu na inny. Załóżmy, że chcemy stworzyć funkcję, która przyjmuje jako argument dowolny napis, a jej zadaniem jest zwrócenie tego samego napisu, ale zamienionego na małe litery i bez zbędnych białych znaków na początku i na końcu.

Moglibyśmy po prostu wywołać kilka metod jedna po drugiej osiągając zamierzony efekt:

```
String original = "    WIELKI NAPIS    ";  
original = original.toLowerCase().trim();
```

Powyższy kod można także opakować w metodę, która przyjmuje String i w wyniku także zwraca String. Zamiast wywoływać kilka metod, można wywołać tylko ją:

```
String getLowerCaseTrim(String original) {  
    return original.toLowerCase().trim();  
}
```

Powyższa metoda przyjmuje obiekt typu String i zwraca w wyniku również String. Odpowiada ona więc takiemu wyrażeniu lambda jak `String s -> String`. Wśród interfejsów funkcyjnych znajdziemy interfejs `Function`, którego metoda `apply()` pasuje do tej sytuacji. Ma ona następującą sygnaturę:

```
R apply(T t);
```

Jak widać, jest to interfejs parametryzowany typami generycznymi. Pod `T` oraz `R` możemy podstawić dowolne typy obiektowe. Jeżeli pod jeden i drugi parametr podstawimy typ `String`, to metoda przyjmuje obiekt `String` i zwraca `String` - dokładnie tak samo jak w naszym przykładzie. Interfejs ten możemy wykorzystać w naszym kodzie, a metodę `getLowerCaseTrim()` możemy zastąpić wyrażeniem lambda. Na początek przypiszmy wyrażenie lambda do zmiennej:

```
Function<String, String> func = (String s) -> s.toLowerCase().trim();
```

Wyrażenie lambda, zapisane po prawej stronie równania, przyjmuje parametr typu `String`, zamienia tekst na małe litery, usuwa białe znaki z końca i początku i zwraca tak zmodyfikowany napis w wyniku. Ze względu na to, że nasza zmienna ma określony typ generyczny, czyli zapisaliśmy `Function<String, String>`, to wirtualna maszyna Javy może wywnioskować typy i wyrażenie lambda można zapisać z pominięciem typu:

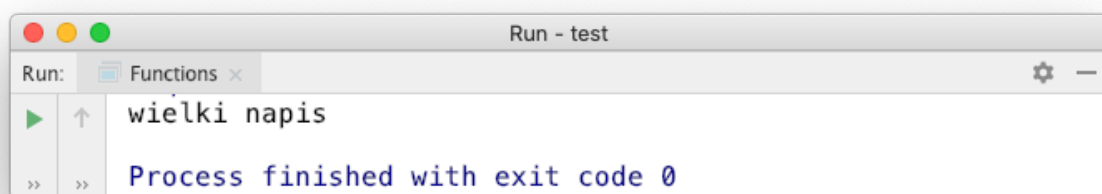
```
Function<String, String> func = text -> text.toLowerCase().trim();
```

W celu wywołania utworzonej funkcji z jakimś argumentem należy wywołać metodę, która zdefiniowana jest w naszym interfejsie funkcyjnym. U nas chodzi o metodę *apply()*.

```
import java.util.function.Function;

class Functions {
    public static void main(String[] args) {
        // funkcja przyjmuje String i zwraca String
        Function<String, String> func = text -> text.toLowerCase().trim();
        String original = "    WIELKI NAPIS    ";
        // wywołujemy funkcję przekazując jej original jako argument
        String lowerCaseTrim = func.apply(original);
        System.out.println(lowerCaseTrim);
    }
}
```

Zapis `func.apply(original)` oznacza "wywołaj wyrażenie lambda/funkcję przypisaną do zmiennej `func` z argumentem `original`".



Interfejs Consumer

Założmy, że teraz chcemy wyświetlić jakiś tekst trzy razy. Możemy dodać po prostu trzy instrukcje `System.out.println()` co wyglądałoby tak:

```
class Functions {  
    public static void main(String[] args) {  
        System.out.println("abc");  
        System.out.println("abc");  
        System.out.println("abc");  
    }  
}
```

Jeżeli w ramach tej samej metody będziemy chcieli wyświetlić także inny tekst trzy razy, to zaczyna nam się duplikować dużo kodu:

```
class Functions {  
    public static void main(String[] args) {  
        System.out.println("abc");  
        System.out.println("abc");  
        System.out.println("abc");  
        System.out.println("xxx");  
        System.out.println("xxx");  
        System.out.println("xxx");  
    }  
}
```

Pierwsze co robimy w takiej sytuacji, to wydzielamy metodę z powtarzalnym kodem. W naszym przypadku metoda ta będzie przyjmowała `String` i wyświetli go 3 razy.

```
class Functions {  
    public static void main(String[] args) {  
        print3Times("abc");  
        print3Times("xxx");  
    }  
  
    private static void print3Times(String abc) {  
        System.out.println(abc);  
        System.out.println(abc);  
        System.out.println(abc);  
    }  
}
```

Metoda *print3Times()* odpowiada takiemu wyrażeniu lambda jak `String s -> void`, czyli przyjmuje parametr typu `String` i nie zwraca nic w wyniku. Wśród interfejsów funkcyjnych znajduje się interfejs `Consumer` z metodą *accept()* o takiej sygnaturze:

```
void accept(T t);
```

Również jest to interfejs generyczny, więc jeśli pod T podstawimy *String*, to otrzymujemy taką samą sygnaturę jak nasza metoda *print3Times()*. Oznacza to, że zamiast metody, możemy zdefiniować wyrażenie lambda, które przypiszemy do zmiennej *Consumer*.

```
import java.util.function.Consumer;

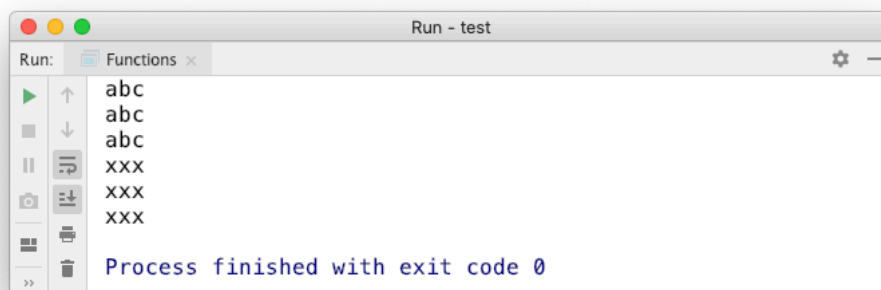
class Functions {
    public static void main(String[] args) {
        Consumer<String> print3Times = s -> {
            System.out.println(s);
            System.out.println(s);
            System.out.println(s);
        };

        print3Times.accept("abc");
        print3Times.accept("xxx");
    }
}
```

Ponieważ w ramach ciała wyrażenia lambda wykonujemy kilka operacji, konkretnie wywołujemy trzykrotnie metodę *System.out.println()*, to konieczne było dodanie dodatkowych nawiasów klamrowych. Zauważ, że w ramach nawiasów klamrowych nie ma żadnego returna, wynika to z sygnatury metody *accept()*, która jest typu *void*.

W celu wywołania wyrażenia lambda, na zmiennej *print3Times* wywołujemy metodę *accept()*, przekazując jej argument, który chcemy wyświetlić.

W tym przypadku nie zyskujemy wiele w porównaniu do zdefiniowania osobnej metody, czytelność pozostaje praktycznie na tym samym poziomie, ilość samego kodu również jest zbliżona, ale jak widzisz jest to ciekawa alternatywa.



Interfejs Predicate

Założmy, że mamy w programie zmienną z wiekiem pewnej osoby. Chcemy sprawdzić, czy osoba ta jest pełnoletnia. Jeśli tak, to wykonamy pewną operację. Standardowo zapiszemy to w taki sposób:

```
class Functions {  
    public static void main(String[] args) {  
        int personAge = 19;  
        if (personAge >= 18) {  
            //jakieś operacje  
        }  
    }  
}
```

Kod ten da się usprawnić. Nie do końca wiadomo, dlaczego porównujemy się akurat z wartością 18 - jest to magiczna liczba. Moglibyśmy wprowadzić jakąś stałą, ale jeszcze lepiej będzie wydzielić metodę, której nadamy nazwę, czyli znaczenie.

```
class Functions {  
    public static void main(String[] args) {  
        int personAge = 19;  
        if (checkIfAdult(personAge)) {  
            //jakieś operacje  
        }  
    }  
  
    static boolean checkIfAdult(int age) {  
        return age >= 18;  
    }  
}
```

Dzięki takiemu zapisowi raczej już jest jasne co oznacza liczba 18. Metoda *checkIfAdult()* przyjmuje parametr typu *int* i zwraca wartość typu *boolean*, odpowiada więc takiemu wyrażeniu lambda jak *int x -> boolean*. Odpowiada to interfejsowi *Predicate*, w którym zdefiniowana jest metoda *test()*:

```
boolean test(T t);
```

Jeżeli pod *T* podstawimy typ *Integer*, to otrzymujemy niemal to samo co w naszym przypadku. Dzięki temu, że mamy mechanizm autoboxingu i autounboxing, możemy założyć, że nie będzie tutaj żadnego problemu. Zamiast osobnej metody, możemy więc zdefiniować zmienną typu *Predicate* i przypisać do niej wyrażenie lambda, które będzie pasowało do sygnatury metody *test()*.


```
import java.util.function.Predicate;

class Functions {
    public static void main(String[] args) {
        int personAge = 19;
        Predicate<Integer> checkIfAdult = age -> age >= 18;
        if (checkIfAdult.test(personAge)) {
            //jakieś operacje
        }
    }
}
```

Rozwiązanie takie ma sens szczególnie wtedy, kiedy danego predykatu używamy tylko w jednej metodzie. Definiowanie osobnych metod może nam niepotrzebnie "zaśmiecać" klasę i musimy się wtedy zastanawiać, czy metoda taka była wykorzystywana tylko w tym jednym miejscu, czy jeszcze gdzieś. Definiując zmienną lokalną, tak jak powyżej, nie ma tego problemu.

Interfejs Supplier

W ostatnim przykładzie pokażę Ci jak możemy tworzyć obiekty z danymi wylosowanymi z kilku tablic. Na początku potrzebna będzie nam klasa *Person*, która reprezentuje osobę, a każda osoba opisana jest przez imię, nazwisko i wiek.

Person.java

```
class Person {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

```
public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

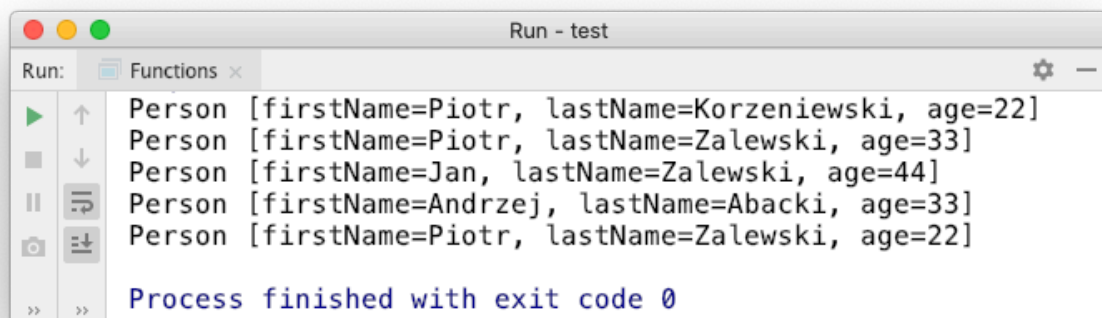
@Override
public String toString() {
    return "Person [firstName=" + firstName + ", lastName=" + lastName + ", age=" + age + "]";
}
}
```

Teraz w metodzie main zdefiniuję tablice z imionami, nazwiskami i liczbami reprezentującymi wiek osób. Na początku stworzymy listę 5 obiektów Person z losowym imieniem, nazwiskiem i wiekiem korzystając z klasycznej pętli.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

class Functions {
    public static void main(String[] args) {
        String[] firstNames = {"Jan", "Karol", "Piotr", "Andrzej"};
        String[] lastNames = {"Abacki", "Kowalski", "Zalewski", "Korzeniewski"};
        int[] ages = {22, 33, 44, 55};
        Random random = new Random();
        List<Person> people = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            String firstName = firstNames[random.nextInt(firstNames.length)];
            String lastName = lastNames[random.nextInt(lastNames.length)];
            int age = ages[random.nextInt(ages.length)];
            Person randomPerson = new Person(firstName, lastName, age);
            people.add(randomPerson);
        }
        for (Person person : people) {
            System.out.println(person);
        }
    }
}
```

W wyniku zobaczysz np.:



```
Run: Functions x
Person [firstName=Piotr, lastName=Korzeniewski, age=22]
Person [firstName=Piotr, lastName=Zalewski, age=33]
Person [firstName=Jan, lastName=Zalewski, age=44]
Person [firstName=Andrzej, lastName=Abacki, age=33]
Person [firstName=Piotr, lastName=Zalewski, age=22]

Process finished with exit code 0
```

Obiekty standardowo tworzymy wywołując konstruktor, zapisując:

```
Person person = new Person("Jan", "Kowalski", 42);
```

Można powiedzieć, że wywołanie konstruktora to trochę jakby powiedzieć, że tworzymy coś z niczego. Gdyby odzwierciedlić to przy pomocy wyrażenia lambda, to miałyby ono np. taką sygnaturę `() -> Person`.

Wśród interfejsów funkcyjnych istnieje coś takiego jak `Supplier`, gdzie zdefiniowana jest metoda `get()` o takiej sygnaturze:

```
T get();
```

Metoda `get()` nie przyjmuje żadnych parametrów i zwraca obiekt typu `T`. Jeżeli pod `T` podstawimy `Person`, to otrzymujemy to, czego szukamy. Zapiszmy fragment kodu i wyrażenie lambda, które pozwoli nam utworzyć obiekty `Person` z losowymi danymi.

```
class PersonOperators {  
    public static void main(String[] args) {  
        String[] firstNames = {"Jan", "Karol", "Piotr", "Andrzej"};  
        String[] lastNames = {"Abacki", "Kowalski", "Zalewski", "Korzeniewski"};  
        int[] ages = {22, 33, 44, 55};  
        Random random = new Random();  
        Supplier<Person> supplier = () -> {  
            String firstName = firstNames[random.nextInt(firstNames.length)];  
            String lastName = lastNames[random.nextInt(lastNames.length)];  
            int age = ages[random.nextInt(ages.length)];  
            return new Person(firstName, lastName, age);  
        };  
        System.out.println(supplier.get());  
        System.out.println(supplier.get());  
        System.out.println(supplier.get());  
    }  
}
```

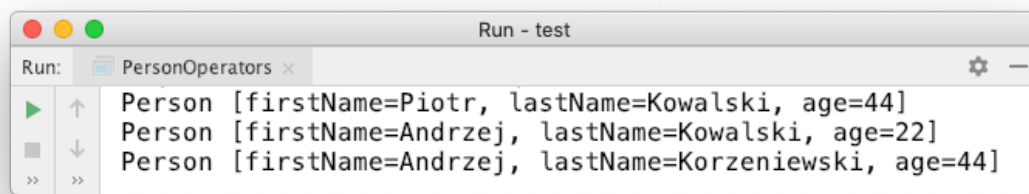
Na początku zdefiniowane są trzy tablice przechowujące odpowiednio imiona, nazwiska oraz wiek osób. Obiekt typu `Random` pozwoli nam wylosować liczbę, odpowiadającą indeksowi obiektu lub wartości, który będziemy pobierali z tablicy. Do zmiennej `supplier` przypisujemy wyrażenie lambda, które jest zgodne z sygnaturą abstrakcyjnej metody `get()` zdefiniowanej w tym interfejsie. Takie wyrażenie lambda nie przyjmuje żadnego argumentu, a zwraca obiekt, w naszym przypadku typu `Person`.

Zapis typu `firstNames[random.nextInt(firstNames.length)]` oznacza "pobierz z tablicy *firstNames* obiekt zapisany pod losowym indeksem z zakresu od 0 do długości tej tablicy. Analogicznie postępujemy z losowaniem nazwiska i wieku, a na końcu zwracamy gotowy obiekt *Person*.

Wywołując metodę `get()`, za każdym razem zostanie utworzony obiekt *Person* z losowym imieniem, nazwiskiem i wiekiem. Im więcej danych będzie w tablicach źródłowych, tym lepiej. Zamiast wywoływać metodę `get()` bezpośrednio w metodzie `println()`, moglibyśmy oczywiście też najpierw obiekty zapamiętać w zmiennych:

```
Person person1 = supplier.get();
System.out.println(person1);
```

Po uruchomieniu powyższego przykładu zobaczysz np. taki wydruk:



Teraz wystarczy dodać do tego wszystkiego pętlę i gotowe.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.function.Supplier;

class Functions {
    public static void main(String[] args) {
        String[] firstNames = {"Jan", "Karol", "Piotr", "Andrzej"};
        String[] lastNames = {"Abacki", "Kowalski", "Zalewski", "Korzeniewski"};
        int[] ages = {22, 33, 44, 55};
        Random random = new Random();
        Supplier<Person> supplier = () -> {
            String firstName = firstNames[random.nextInt(firstNames.length)];
            String lastName = lastNames[random.nextInt(lastNames.length)];
            int age = ages[random.nextInt(ages.length)];
            return new Person(firstName, lastName, age);
        };
        List<Person> people = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            people.add(supplier.get());
        }
        for (Person person : people) {
            System.out.println(person);
        }
    }
}
```

Podsumowanie

W tej lekcji pokazałem ci podstawową mechanikę korzystania z wyrażeń lambda. Zapamiętaj, że w Javie istnieje coś takiego jak interfejsy funkcyjne, czyli takie interfejsy, które mają tylko jedną metodę abstrakcyjną (mogą mieć także inne metody, np. domyślne, statyczne, albo prywatne). Do zmiennej tego typu można przypisać wyrażenie lambda, które będzie zgodne z metodą zdefiniowaną w takim interfejsie. Prawdziwe korzyści korzystania z wyrażeń lambda zobaczysz jednak, gdy połączymy je z operacjami na kolekcjach oraz strumieniach.

Wyrażenia lambda i typy generyczne

Czego się dowiesz

jak wykorzystać interfejsy Consumer, Predicate, Function i Supplier w połączeniu z typami generycznymi.

Interfejs Consumer

Interfejsy funkcyjne i wyrażenia lambda często będą wykorzystywane do tworzenia uogólnionych metod, które pozwolą nam wykonywać różnorakie operacje np. na kolekcjach. W większości przypadków będziemy korzystali z już istniejących metod, które wchodzi w skład np. interfejsu List, ale poniżej chcę Ci pokazać jak takie metody można zapisać samodzielnie.

Załóżmy, że masz długą listę osób reprezentowanych przez typ Person (imię, nazwisko, wiek):

Person.java

```
class Person {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [firstName=" + firstName + ", lastName=" + lastName + ", age=" + age + "]";
    }
}
```

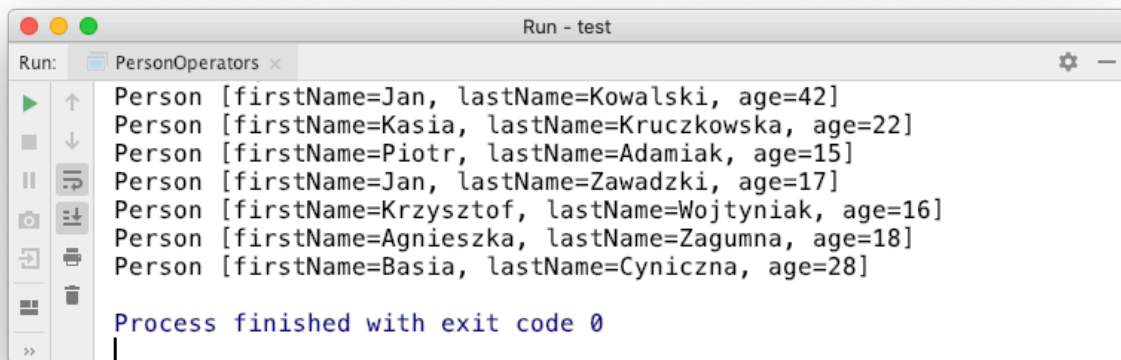
Powyższa klasa jest ci już doskonale znana. Utwórzmy teraz listę kilku osób i wyświetlmy informacje o tych obiektach w konsoli.

PersonOperators.java

```
import java.util.ArrayList;
import java.util.List;

class PersonOperators {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Jan", "Kowalski", 42));
        people.add(new Person("Kasia", "Kruczkowska", 22));
        people.add(new Person("Piotr", "Adamiak", 15));
        people.add(new Person("Jan", "Zawadzki", 17));
        people.add(new Person("Krzysztof", "Wojtyniak", 16));
        people.add(new Person("Agnieszka", "Zagumna", 18));
        people.add(new Person("Basia", "Cyniczna", 28));

        for (Person person : people) {
            System.out.println(person);
        }
    }
}
```



Jeżeli chcielibyśmy wykonywać operacje na liście i wyświetlać ją kilkakrotnie, to dobrze będzie wydzielić pętlę wyświetlającą dane do osobnej metody, np.

```
class PersonOperators {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        //uzupełnienie listy danymi

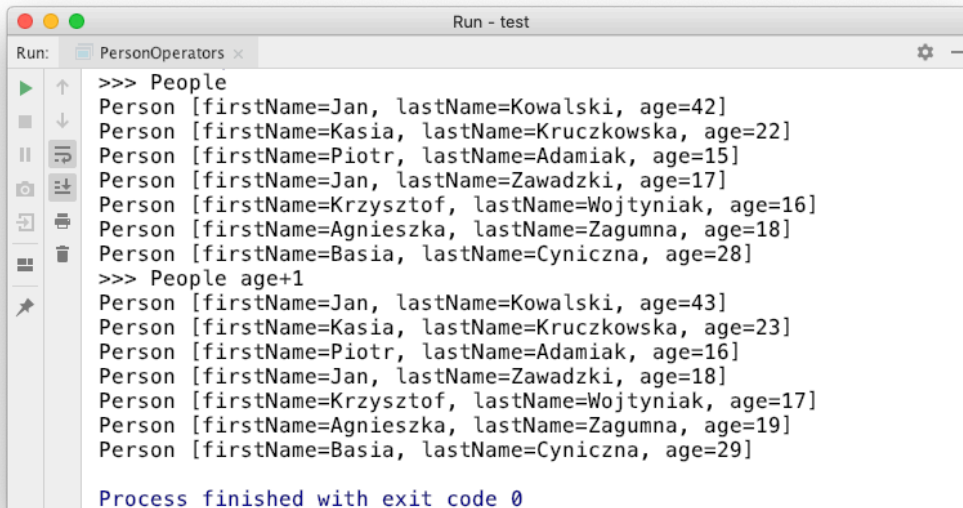
        printList(people);
    }

    private static void printList(List<Person> people) {
        for (Person person : people) {
            System.out.println(person);
        }
    }
}
```

Założmy, że teraz chcemy najpierw wyświetlić oryginalną listę, później zwiększyć wiek każdej osoby w niej zapisanej, a następnie dla potwierdzenia wyświetlić jej zawartość jeszcze raz. W metodzie main możemy dopisać dodatkową pętlę, w której wywołamy metodę `setAge()` na każdym z obiektów.

```
class PersonOperators {  
    public static void main(String[] args) {  
        List<Person> people = new ArrayList<>();  
        //uzupełnienie listy danymi  
  
        System.out.println(">>> People");  
        printList(people);  
        System.out.println(">>> People age+1");  
        for (Person person : people) {  
            int currentAge = person.getAge();  
            person.setAge(currentAge + 1);  
        }  
        printList(people);  
    }  
  
    private static void printList(List<Person> people) {  
        for (Person person : people) {  
            System.out.println(person);  
        }  
    }  
}
```

W wyniku zobaczysz wydruk potwierdzający, że wiek każdej osoby został zwiększony:



```
Run: PersonOperators x  
>>> People  
Person [firstName=Jan, lastName=Kowalski, age=42]  
Person [firstName=Kasia, lastName=Kruczkowska, age=22]  
Person [firstName=Piotr, lastName=Adamiak, age=15]  
Person [firstName=Jan, lastName=Zawadzki, age=17]  
Person [firstName=Krzysztof, lastName=Wojtyniak, age=16]  
Person [firstName=Agnieszka, lastName=Zagumna, age=18]  
Person [firstName=Basia, lastName=Cyniczna, age=28]  
>>> People age+1  
Person [firstName=Jan, lastName=Kowalski, age=43]  
Person [firstName=Kasia, lastName=Kruczkowska, age=23]  
Person [firstName=Piotr, lastName=Adamiak, age=16]  
Person [firstName=Jan, lastName=Zawadzki, age=18]  
Person [firstName=Krzysztof, lastName=Wojtyniak, age=17]  
Person [firstName=Agnieszka, lastName=Zagumna, age=19]  
Person [firstName=Basia, lastName=Cyniczna, age=29]  
  
Process finished with exit code 0
```

Jeżeli operację zwiększania wieku chcielibyśmy powtarzać, to podobnie jak z wyświetlaniem, warto byłoby wydzielić taką operację do osobnej metody. Zastanówmy się jednak przez chwilę co tak naprawdę oznacza wyświetlanie jakiegoś obiektu, albo zwiększanie jego wieku.

Wiersz:

```
System.out.println(person);
```

można tak naprawdę wyobrazić sobie jako takie wyrażenie lambda:

```
Person person -> void
```

Wyrażenie lambda przyjmuje argument typu *Person* i nic nie zwraca w wyniku (wyświetla obiekt). Jeżeli spojrzymy na zwiększanie wieku, czyli wiersz:

```
person.setAge(currentAge + 1);
```

albo jego uproszczona wersja:

```
person.setAge(person.getAge() + 1);
```

to również w tym przypadku będzie to taka sama sygnatura wyrażenia lambda:

```
Person person -> void
```

ponieważ możemy to przetłumaczyć jako "weź obiekt person i zwiększ w nim wiek o 1". Setter ze swojej definicji jest typu void, więc nic nie zwraca.

Skoro zarówno wyświetlanie, jak i zwiększanie wieku sprowadza się do takiego samego wyrażenia lambda, to możemy taki kod uogólnić. Wykorzystamy odpowiedni interfejs funkcyjny i typy generyczne do wydzielenia bardziej ogólnej metody.

Jeżeli przejrzymy listę dostępnych interfejsów funkcyjnych, znajdujących się w pakiecie `java.util.function`, to znajdziemy tam m.in. interfejs *Consumer*, który ma tylko jedną metodę *accept()*, która wygląda tak:

```
void accept(T t);
```

Jeżeli zapisalibyśmy tę metodę jako wyrażenie lambda, to sprowadza się ona do czegoś takiego jak `T t -> void`, jeśli pod typ generyczny *T* podstawimy *Person*, to otrzymujemy takie samo wyrażenie lambda, jak przy wyświetlaniu obiektu, albo zwiększaniu wieku z wykorzystaniem settera. Nazwa interfejsu, czyli *Consumer*, dużo mówi o jego funkcji. Zajmuje się on tym, że bierze jakiś obiekt, wykonuje na nim pewną

operację i nie zwraca żadnego wyniku, czyli krótko mówiąc: konsumuje go.

Zapiszmy ogólną metodę, która pozwoli nam skonsumować każdy obiekt w liście, czyli wykonać na nim pewną operację, która nie daje nic w wyniku.

```
private static <T> void consumeList(List<T> list, Consumer<T> consumer) {  
    for (T t : list) {  
        consumer.accept(t);  
    }  
}
```

Metoda jest sparametryzowana typem T. Jeżeli do metody prześlemy listę przechowującą obiekty typu *Person*, to pod T zostanie podstawione *Person*. Analogicznie dla listy typu *String*, pod T zostanie podstawiony *String*. Jako drugi argument metody można przekazać np. obiekt klasy anonimowej implementującej interfejs *Consumer*, ale dużo bardziej praktyczne będzie przekazać tam pewne wyrażenie lambda (jest to możliwe, bo *Consumer* jest interfejsem funkcyjnym). Zapis `consumer.accept(t)` oznacza więc tak naprawdę "wykonaj wyrażenie lambda przypisane do zmiennej consumer z argumentem t (czyli kolejnym obiektem z listy).

Dzięki takiemu zapisowi możemy w wygodny sposób wykonywać operacje "konsumowania" każdego obiektu z listy, dzięki czemu zarówno wyświetlanie, jak i zwiększanie wieku każdej osoby będzie sprowadzało się teraz do tylko jednej linijki kodu:

```
import java.util.ArrayList;  
import java.util.List;  
import java.util.function.Consumer;  
  
class PersonOperators {  
    public static void main(String[] args) {  
        List<Person> people = new ArrayList<>();  
        //uzupełnienie listy danymi  
  
        System.out.println(">>> People");  
        consumeList(people, p -> System.out.println(p));  
        System.out.println(">>> People age+1");  
        consumeList(people, p -> p.setAge(p.getAge() + 1));  
        consumeList(people, p -> System.out.println(p));  
    }  
  
    private static <T> void consumeList(List<T> list, Consumer<T> consumer) {  
        for (T t : list) {  
            consumer.accept(t);  
        }  
    }  
}
```

Powyższa metoda daje bardzo duże możliwości. Jeżeli chcielibyśmy teraz zwiększyć wiek każdej osoby o 5, to wystarczy zapisać:

```
consumeList(people, p -> p.setAge(p.getAge() + 5));
```

a w celu wyświetlenia samych nazwisk:

```
consumeList(people, p -> System.out.println(p.getLastName()));
```

Jak widzisz zawsze jest to zaledwie jedna linijka kodu, nie ma potrzeby definiowania do tego celu nowych pętli, czy metod.

Interfejs Predicate i filtrowanie

Kolejna czynność, którą chcę wykonać na liście, to pobranie z niej wyłącznie osób pełnoletnich, czyli takich, które mają co najmniej 18 lat i wyświetlenie tych osób w konsoli. W pierwszym kroku utworzę nową listę, w której będą tylko pełnoletnie osoby. Następnie do jej wyświetlenia wykorzystam przed chwilą zdefiniowaną metodę *consumeList()*. Filtrowanie osób i tworzenie nowej listy wydzielę dla czytelności do osobnej metody, którą nazwę *filterAdults()*.

```
private static List<Person> filterAdults(List<Person> people) {  
    List<Person> adults = new ArrayList<>();  
    for (Person person : people) {  
        if (person.getAge() >= 18)  
            adults.add(person);  
    }  
    return adults;  
}
```

Metoda przyjmuje listę obiektów typu *Person*. Stworzyłem w niej na początku pustą listę i w pętli dodaję do niej obiekty spełniające warunek `person.getAge() >= 18`.

Zanim przejdziemy do testowania metody i wyświetlania obiektów, zapiszmy jeszcze jedną metodę, która przefiltruje listę i zwróci listę wszystkich osób, które mają na imię Jan. Metoda taka będzie bardzo podobna, inny będzie przede wszystkim warunek w instrukcji `if`:

```
private static List<Person> filterJanPeople(List<Person> people) {  
    List<Person> janPeople = new ArrayList<>();  
    for (Person person : people) {  
        if ("Jan".equals(person.getFirstName()))  
            janPeople.add(person);  
    }  
    return janPeople;  
}
```

Takich metod filtrujących mógłbym wymyślać jeszcze wiele, ale zauważ, że w ogólności zmienia się w nich jedynie warunek, który jest zapisany w instrukcji if. Jeżeli spojrzymy na te warunki, czyli `person.getAge() >= 18` oraz `"Jan".equals(person.getFirstName())`, to oba sprowadzają się do takiego wyrażenia lambda jak `Person p -> boolean`, czyli "weź obiekt typu *Person*, wykonaj pewne działanie i zwróć w wyniku wartość logiczną". W pierwszym przypadku z obiektu typu *Person* wyciągamy wiek i porównujemy go z wartością 18, a w drugim z obiektu pobieramy nazwisko i porównujemy je z napisem Jan.

Jeśli ponownie sięgniemy do listy interfejsów funkcyjnych, to znajdziemy tam interfejs Predicate, który posiada abstrakcyjną metodę `test()` o takiej sygnaturze:

```
boolean test(T t);
```

co przekłada się na wyrażenie lambda `T t -> boolean`. Jeśli pod typ generyczny *T* podstawimy *Person*, to otrzymamy dokładnie takie wyrażenie lambda, jakie potrzebne jest w naszych ifach. Wykorzystując tę wiedzę, możemy podobnie jak w przypadku wyświetlania i zwiększania wieku, zdefiniować nową, uogólnioną metodę, która przefiltruje listę na podstawie dowolnego predykatu.

```
private static <T> List<T> filterByPredicate(List<T> list,  
Predicate<T> predicate) {  
    List<T> result = new ArrayList<>();  
    for (T t : list) {  
        if (predicate.test(t))  
            result.add(t);  
    }  
    return result;  
}
```

Fragment

```
if (predicate.test(t))
    result.add(t);
```

możemy rozumieć jako "dodaj obiekt *t* do listy *result*, jeżeli wyrażenie lambda przypisane do zmiennej *predicate*, wywołane z argumentem *t* zwróci w wyniku wartość *true*", albo mówiąc w języku matematyków "dodaj obiekt *t* do listy *result*, jeżeli predykat dla argumentu *t* jest spełniony".

Z tak przygotowaną metodą filtrującą, możemy wrócić do metody *main* i przetestować filtrowanie na podstawie różnych kryteriów.

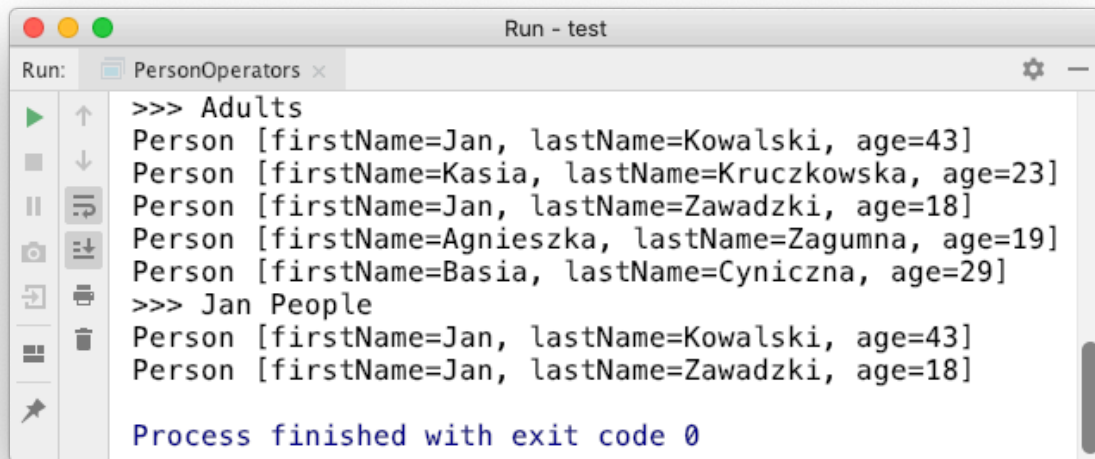
```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

class PersonOperators {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        //uzupełnienie listy danymi

        System.out.println(">>> Adults");
        List<Person> adults = filterByPredicate(people, person ->
person.getAge() >= 18);
        consumeList(adults, p -> System.out.println(p));
        System.out.println(">>> Jan People");
        List<Person> janPeople = filterByPredicate(people, person ->
"Jan".equals(person.getFirstName()));
        consumeList(janPeople, p -> System.out.println(p));
    }

    private static <T> List<T> filterByPredicate(List<T> list,
Predicate<T> predicate) {
        List<T> result = new ArrayList<>();
        for (T t : list) {
            if (predicate.test(t))
                result.add(t);
        }
        return result;
    }

    //pozostałe metody z tej lekcji
}
```



```
Run - test
Run: PersonOperators x
>>> Adults
Person [firstName=Jan, lastName=Kowalski, age=43]
Person [firstName=Kasia, lastName=Kruczkowska, age=23]
Person [firstName=Jan, lastName=Zawadzki, age=18]
Person [firstName=Agnieszka, lastName=Zagumna, age=19]
Person [firstName=Basia, lastName=Cyniczna, age=29]
>>> Jan People
Person [firstName=Jan, lastName=Kowalski, age=43]
Person [firstName=Jan, lastName=Zawadzki, age=18]

Process finished with exit code 0
```

Filtrowanie listy osób (i nie tylko osób, bo metoda jest generyczna, więc można nią filtrować dowolną listę), daje nam bardzo dużą elastyczność. W zależności od tego jakie wyrażenie lambda przekazemy w miejsce predykatu, to najczęściej wywołanie metody nie będzie zajmowało więcej niż 1 linijka kodu.

Interfejs Function i mapowanie

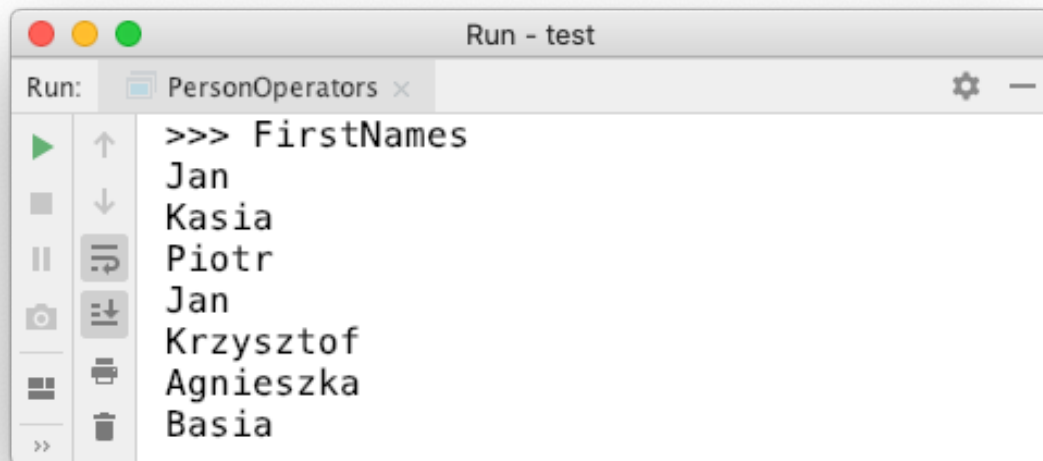
Kolejny przykład, który chcę Ci pokazać, będzie polegał na tym, że zamiast listy obiektów typu *Person*, chciałbym mieć listę z samymi imionami tych osób. Ponownie zaczniemy od tego co już znamy, czyli stworzymy nową listę sparametryzowaną typem *String*, zapisujemy pętlę, w której przechodzimy po każdym elemencie listy z obiektami *Person*, wyciągamy z nich imiona i dodajemy do wcześniej utworzonej listy *Stringów*.

```
class PersonOperators {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        //uzupełnienie listy danymi

        List<String> firstNames = new ArrayList<>();
        for (Person person : people) {
            firstNames.add(person.getFirstName());
        }
        consumeList(firstNames, s -> System.out.println(s));
    }
}

//pozostałe metody z tej lekcji
}
```

Zwróć uwagę, że tym razem przekazałem do metody *consumeList()* listę przechowującą obiekty *String*. Z tego powodu wyrażenie lambda też operuje na obiektach *String* i użyty zapis `s -> System.out.println(s)` to krótsza wersja `(String s) -> System.out.println(s)`. Po uruchomieniu programu zobaczysz listę imion:



Operację wyciągnięcia imion z obiektów *Person* możemy wydzielić do osobnej metody i nasz kod wyglądałby w taki sposób:

```
class PersonOperators {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        //uzupełnienie listy danymi

        System.out.println(">>> FirstNames");
        List<String> firstNames = getFirstNames(people);
        consumeList(firstNames, (String s) -> System.out.println(s));
    }

    private static List<String> getFirstNames(List<Person> people) {
        List<String> firstNames = new ArrayList<>();
        for (Person person : people) {
            firstNames.add(person.getFirstName());
        }
        return firstNames;
    }
}

//pozostałe metody z tej lekcji
}
```

To co dzieje się w tej metodzie, to tak naprawdę przekształcenie listy typu *List<Person>* na *List<String>*. Operację, która polega na wykonaniu pewnej czynności na każdym obiekcie kolekcji, a w wyniku zwraca kolekcję z z wynikiem tej czynności, nazywamy mapowaniem.

Jeżeli przyjrzymy się temu fragmentowi kodu:

```
for (Person person : people) {  
    firstNames.add(person.getFirstName());  
}
```

to rozpisując go z wykorzystaniem klasycznej pętli for i nieco rozciągając zapis, osiągniemy coś takiego:

```
for (int i = 0; i < people.size(); i++) {  
    Person person = people.get(i);  
    String firstName = person.getFirstName();  
    firstNames.add(firstName);  
}
```

Interesują nas tutaj przede wszystkim dwa wiersze:

```
Person person = people.get(i);  
String firstName = person.getFirstName();
```

które sprowadzają się do takiego wyrażenia lambda jak `Person p -> String`. Po przeszukaniu listy interfejsów funkcyjnych, znajdziemy coś takiego jak Function. Jest to interfejs funkcyjny, który ma zdefiniowaną jedną metodę abstrakcyjną *apply()* o takiej sygnaturze:

```
R apply(T t);
```

To co odróżnia interfejs *Function* od innych interfejsów funkcyjnych, które przedstawiłem ci do tej pory, to fakt, że wykorzystuje on dwa parametry generyczne. Dzieje się tak dlatego, że metoda *apply()* przekształca obiekt typu T w obiekt typu R, czyli takie wyrażenie lambda `T t -> R`. W naszym przypadku pod T podstawimy *Person*, a pod R typ *String*.

Zapiszmy uogólnioną metodę, która pozwoli nam wykonać operację przekształcenia listy jednego typu na inny.


```
class PersonOperators {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        //uzupełnienie listy danymi

        System.out.println(">>> FirstNames");
        List<String> firstNames = convertList(people, person ->
person.getFirstName());
        consumeList(firstNames, (String s) -> System.out.println(s));
    }

    private static <T, R> List<R> convertList(List<T> list, Function<T, R>
function) {
        List<R> resultList = new ArrayList<>();
        for (T t : list) {
            R result = function.apply(t);
            resultList.add(result);
        }
        return resultList;
    }
}

//pozostałe metody z tej lekcji
}
```

Wywołując metodę *convertList()* w metodzie *main()* w taki sposób *convertList(people, person -> person.getFirstName())*, linijkę kodu

```
R result = function.apply(t);
```

należy sobie tłumaczyć jako:

```
String result = person.getFirstName()
```

To co ważne, to dzięki tej samej metodzie możesz bez problemu wyciągnąć np. wiek wszystkich osób. Wystarczy, że jako argument jej wywołania, przekażesz inne wyrażenie lambda:

```
List<Integer> ages = convertList(people, person -> person.getAge());
```

W tym przypadku funkcja służy do mapowania obiektów z typu *Person*, na typ *Integer*.

Interfejs Supplier i generowanie obiektów

Ostatni przykład będzie dotyczył tworzenia obiektów. Na początku tej lekcji stworzyłem listę gotowych obiektów, ale teraz chciałbym to przerobić w taki sposób, żeby były one częściowo generowane losowo. Przygotuję sobie na początku kilka tablic, w których będą imiona, nazwiska oraz liczby (wiek osób) i na wybierając z nich losowe wartości będę tworzył obiekty *Person*.

Jako punkt wyjścia potraktuję tutaj kod, który stworzyłem w poprzedniej lekcji, wykorzystując interfejs *Supplier*.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.function.Consumer;
import java.util.function.Supplier;

class PersonOperators {
    public static void main(String[] args) {
        String[] firstNames = {"Jan", "Karol", "Piotr", "Andrzej"};
        String[] lastNames = {"Abacki", "Kowalski", "Zalewski", "Korzeniewski"};
        int[] ages = {22, 33, 44, 55};
        Random random = new Random();
        Supplier<Person> supplier = () -> {
            String firstName = firstNames[random.nextInt(firstNames.length)];
            String lastName = lastNames[random.nextInt(lastNames.length)];
            int age = ages[random.nextInt(ages.length)];
            return new Person(firstName, lastName, age);
        };
        List<Person> people = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            people.add(supplier.get());
        }
        consumeList(people, p -> System.out.println(p));
    }

    private static <T> void consumeList(List<T> list, Consumer<T> consumer) {
        for (T t : list) {
            consumer.accept(t);
        }
    }
}
```

Pójdźmy jednak o krok dalej i zdefiniujmy metodę generyczną, która będzie tworzyła dla nas listę obiektów, na podstawie przekazanego do niej wyrażenia lambda (Suppliera).

```
private static <T> List<T> generateRandomList(int elements, Supplier<T>
supplier) {
    List<T> result = new ArrayList<>();
    for (int i = 0; i < elements; i++) {
        result.add(supplier.get());
    }
    return result;
}
```

Metoda stworzy dla nas listę i wypełni ją obiektami dostarczonymi przez *supplier*. Parametr *elements* pozwala określić ile obiektów ma się znaleźć w liście. Metodę możemy teraz wywołać w metodzie *main()*, np. w taki sposób:

```
List<Person> people = generateRandomList(5, () -> {
    String firstName = firstNames[random.nextInt(firstNames.length)];
    String lastName = lastNames[random.nextInt(lastNames.length)];
    int age = ages[random.nextInt(ages.length)];
    return new Person(firstName, lastName, age);
});
```

ale jeśli kod *supplier* jest kilkunastokrotny, to lepszym pomysłem może być rozbięcie tego na dwa kroki:

```
Supplier<Person> supplier = () -> {
    String firstName = firstNames[random.nextInt(firstNames.length)];
    String lastName = lastNames[random.nextInt(lastNames.length)];
    int age = ages[random.nextInt(ages.length)];
    return new Person(firstName, lastName, age);
};
List<Person> people = generateRandomList(5, supplier);
```

Ostatecznie otrzymujemy taki kod:

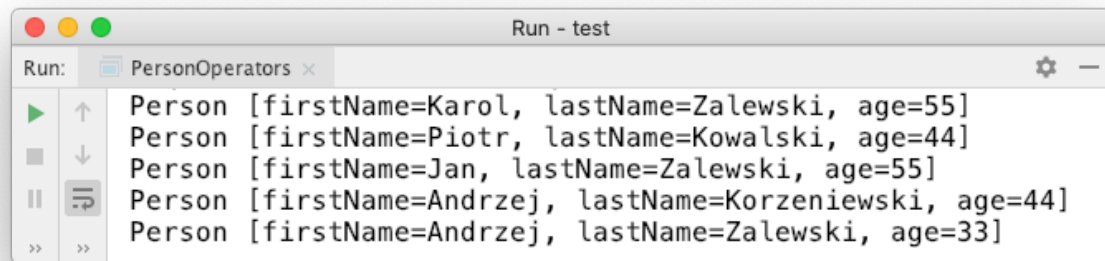
```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.function.Consumer;
import java.util.function.Supplier;

class PersonOperators {
    public static void main(String[] args) {
        String[] firstNames = {"Jan", "Karol", "Piotr", "Andrzej"};
        String[] lastNames = {"Abacki", "Kowalski", "Zalewski", "Korzeniewski"};
        int[] ages = {22, 33, 44, 55};
        Random random = new Random();
        Supplier<Person> supplier = () -> {
            String firstName = firstNames[random.nextInt(firstNames.length)];
            String lastName = lastNames[random.nextInt(lastNames.length)];
            int age = ages[random.nextInt(ages.length)];
            return new Person(firstName, lastName, age);
        };
        List<Person> people = generateRandomList(5, supplier);
        consumeList(people, p -> System.out.println(p));
    }

    private static <T> List<T> generateRandomList(int elements, Supplier<T>
supplier) {
        List<T> result = new ArrayList<>();
        for (int i = 0; i < elements; i++) {
            result.add(supplier.get());
        }
        return result;
    }
}

//pozostałe metody z tej lekcji
}
```

i przykładowy wydruk (u ciebie będą inne dane):



```
Run - test
Person [firstName=Karol, lastName=Zalewski, age=55]
Person [firstName=Piotr, lastName=Kowalski, age=44]
Person [firstName=Jan, lastName=Zalewski, age=55]
Person [firstName=Andrzej, lastName=Korzeniewski, age=44]
Person [firstName=Andrzej, lastName=Zalewski, age=33]
```

Ale po co się męczyć z tymi wyrażeniami lambda?

Możliwe, że kolejne pytanie jakie sobie teraz zadajesz, to po co właściwie używać tych wyrażeń lambda, skoro wszystko to co powyżej da się zrobić za pomocą klasycznego kodu z Javy? Otóż zauważ, że wszystkie nasze sparametryzowane metody z przykładu mogą przyjąć tak naprawdę listy dowolnego typu i dowolne wyrażenia lambda, zgodne z typem interfejsu funkcyjnego. Możemy nimi przefiltrować zarówno listę liczb, osób, samochodów, jak i zwierząt. Kod jest uniwersalny i wywoływany w jednej linijce. Więcej zalet elementów programowania funkcyjnego w Javie zobaczysz w innych lekcjach, gdzie kod będzie jeszcze bardziej czytelny, bo jak się okazuje operacje na kolekcjach od Javy 8 są dużo łatwiejsze i wcale nie wymagają definiowania nie do końca czytelnych metod generycznych - są one w zasadzie przygotowane dla nas w ramach JDK.

Korzystanie z wyrażeń lambda w Javie absolutnie nie jest wymogiem i tak naprawdę praktycznie wszystko da się zrobić bez nich, tyle że czasami dużo większym kosztem czasowym, a przede wszystkim większą ilością kodu, który prawdopodobnie będzie mniej czytelny. Aktualnie wyrażenia lambda stały się swego rodzaju standardem i warto próbować je wykorzystywać tam gdzie to możliwe, szczególnie w połączeniu z kolekcjami i strumieniami, które poznasz w innych lekcjach.