Wojskowa Akademia Techniczna

WYDZIAŁ ELEKTRONIKI
WOJSKOWA AKADEMIA TECHNICZNA

Zaawansowane programowanie w języku Java

Laboratorium 4

Strumienie, odczyt i zapis danych, serializacja, kolekcje.

2017

LABORATORIUM 4

Zagadnienia:

- ✓ Gromadzenie większej ilości danych - **Kolekcje**
- ✓ Przetwarzanie kolekcji – **Streams**
- ✓ Wczytywanie i zapis danych do pliku

PUNKTY:

1 pkt – obecność
1 pkt - Aplikacja 01 – kolekcje podstawy
2 pkt - Aplikacja 02 – kolekcje zaawansowane
2 pkt - Aplikacja 03 – filtrowanie danych z pliku

Problem

1. Załóżmy, że mamy ***n*** elementów (np. zbiór imion i nazwisk):

Jan Kowalski

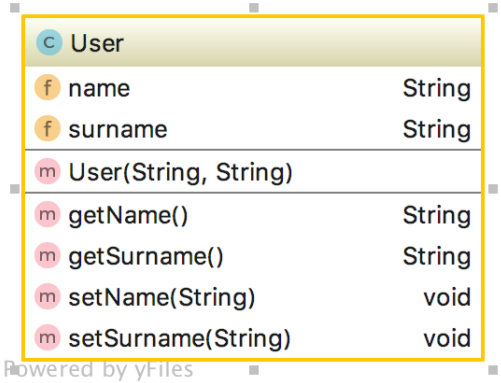
Tomasz Frek

Dominika Olszewska

Karolina Maćkowska

Laura Koliber

...



Problem

2. Możemy je „zgromadzić” w ***n*** zmiennych:

Jan Kowalski

Tomasz Frek

Dominika Olszewska

Karolina Maćkowska

Laura Koliber

...

```
User user1 = new User( name: "Jan",  surname: "Kowalski");
User user2 = new User( name: "Tomasz",  surname: "Frek");
User user3 = new User( name: "Dominika",  surname: "Olszewska");
User user4 = new User( name: "Karolina",  surname: "Maźkowska");
User user5 = new User( name: "Laura",  surname: "Koliber");

//...
```

Problem

3. Niestety takie rozwiązanie stwarza problemy przy przetwarzaniu, np. wyświetlenie wszystkich użytkowników:

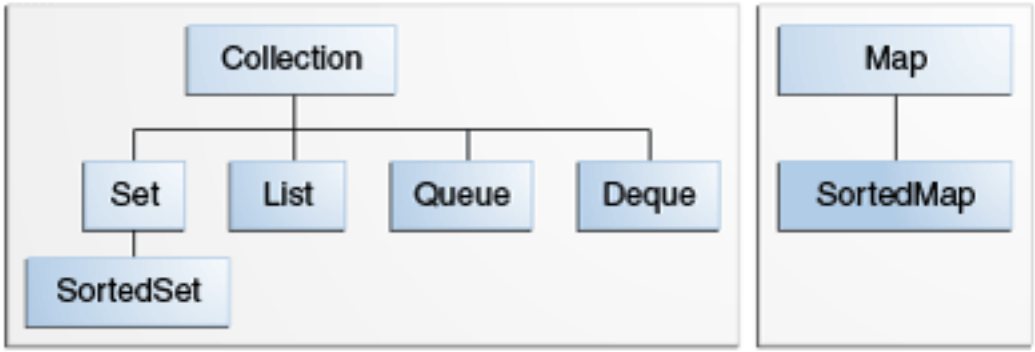
```
System.out.println(user1.getName() + " " + user1.getSurname());
System.out.println(user2.getName() + " " + user2.getSurname());
System.out.println(user3.getName() + " " + user2.getSurname());
System.out.println(user4.getName() + " " + user4.getSurname());
System.out.println(user5.getName() + " " + user5.getSurname());

//...
```

Wady:

1. Łatwo o pomyłkę (np. wyświetlenie usera3)
2. Powielanie kodu
3. Brak możliwości wygenerowania kodu
4. Brak możliwości przetwarzania w pętli
5. itd.

Rozwiązanie

Wykorzystanie Java Collections API:

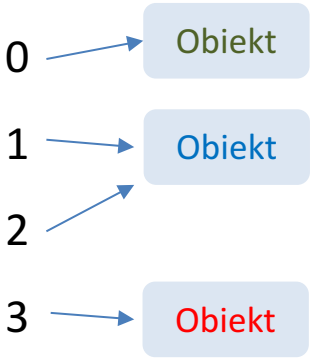


The core collection interfaces.

Rozwiązania problemu

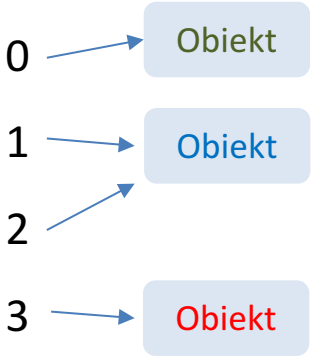
Rozwiązanie 1

Array
(tablica)



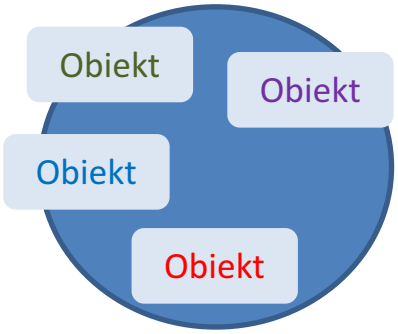
Rozwiązanie 2

List
(lista)



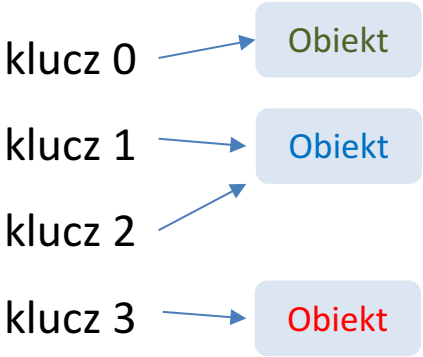
Rozwiązanie 3

Set
(zbiór)



Rozwiązanie 4

Map
(mapa)



Tablice i Mapy nie są stricte kolekcjami (w tym sensie, że nie implementują interfejsu Collection)

Rozwiązanie 1 - Array

1. Tworzenie:

```
User user1 = new User( name: "Jan",  surname: "Kowalski");
User user2 = new User( name: "Tomasz",  surname: "Frek");
User user3 = new User( name: "Dominika",  surname: "Olszewska");
User user4 = new User( name: "Karolina",  surname: "Maźkowska");
User user5 = new User( name: "Laura",  surname: "Koliber");

//...
```

```
User[] users = new User[5];
users[0] = new User( name: "Jan",  surname: "Kowalski");
users[1] = new User( name: "Tomasz",  surname: "Frek");
users[2] = new User( name: "Dominika",  surname: "Olszewska");
users[3] = new User( name: "Karolina",  surname: "Maźkowska");
users[4] = new User( name: "Laura",  surname: "Koliber");

//...
```

inny wariant

```
User[] users = {new User( name: "Jan",  surname: "Kowalski"),
        new User( name: "Tomasz",  surname: "Frek"),
        new User( name: "Dominika",  surname: "Olszewska"),
        new User( name: "Karolina",  surname: "Maźkowska"),
        new User( name: "Laura",  surname: "Koliber")};
```

Rozwiązanie 1 - Array

2. Przetwarzanie - wyświetlanie:

```
System.out.println(user1.getName() + " " + user1.getSurname());
System.out.println(user2.getName() + " " + user2.getSurname());
System.out.println(user3.getName() + " " + user3.getSurname());
System.out.println(user4.getName() + " " + user4.getSurname());
System.out.println(user5.getName() + " " + user5.getSurname());

//...
```

```
for (int i=0; i < users.length; i++) {
    System.out.println(users[i].getName() + " " + users[i].getSurname());
}
```

inny wariant

```
for (User user : users) {
    System.out.println(user.getName() + " " + user.getSurname());
}
```

Rozwiązanie 1 - Array

3. Zalety:

- **Bardzo proste rozwiązanie,**
- Mogą przechowywać typy prymitywne (int, char, itp.)

4. Wady:

- **Musimy** znać dokładną ilość elementów
- itp.

```
User[] users = new User[5];
```

Tej wady pozbawione są kolejne rozwiązania.

Rozwiązanie 2 - List

java.util.List

Lista to kolekcja, w której każdy element ma przyporządkowany numer kolejny (indeks). Podstawowe operacje na liście to dodaj element (***add(E element)***) oraz pobierz element z pozycji i (***E get(int position)***).
Obiekty na liście mogą się powtarzać – tj. ten sam obiekt może się pojawić na liście kilka razy (np. na pozycjach 0, 1 i 2)

Najpopularniejsze implementacje:
ArrayList – przechowuje dane wewnętrznie w tablicy. Opcja optymalna jeśli znamy docelowy rozmiar listy lub operacji dodawania wykonujemy mało w stosunku do operacji odczytu, a operacje odczytu nie są w pętli for-each

LinkedList – przechowuje dane w postaci powiązanej, wydajniejsza w sytuacjach w których dodajemy wiele elementów, a odczyt odbywa się sekwencyjnie (odczytujemy elementy za pomocą iteratora lub pętli for-each)

Rozwiązanie 2 - List

1. Tworzenie:

```
User user1 = new User( name: "Jan",  surname: "Kowalski");
User user2 = new User( name: "Tomasz",  surname: "Frek");
User user3 = new User( name: "Dominika",  surname: "Olszewska");
User user4 = new User( name: "Karolina",  surname: "Maźkowska");
User user5 = new User( name: "Laura",  surname: "Koliber");

//...
```

```
List<User> userList = new ArrayList<>();
userList.add(new User( name: "Jan",  surname: "Kowalski"));
userList.add(new User( name: "Tomasz",  surname: "Frek"));
userList.add(new User( name: "Dominika",  surname: "Olszewska"));
userList.add(new User( name: "Karolina",  surname: "Maźkowska"));
userList.add(new User( name: "Laura",  surname: "Koliber"));
```

Rozwiązanie 2 - List

2. Przetwarzanie - wyświetlanie:

```
System.out.println(user1.getName() + " " + user1.getSurname());
System.out.println(user2.getName() + " " + user2.getSurname());
System.out.println(user3.getName() + " " + user3.getSurname());
System.out.println(user4.getName() + " " + user4.getSurname());
System.out.println(user5.getName() + " " + user5.getSurname());

//...
```

```
for (int i = 0; i < users.size(); i++) {
    System.out.println(users.get(i).getName() + " " + users.get(i).getSurname());
}
```

inny wariant

```
for (User user : users) {
    System.out.println(user.getName() + " " + user.getSurname());
}
```

inny wariant

```
users.forEach(user -> System.out.println(user.getName() + " " + user.getSurname()));
```

Rozwiązanie 3 - Set

java.util.Set

Set (zbiór) to kolekcja, w której elementy **nie mają** przyporządkowanego numeru kolejnego (indeksu), a dostęp do nich odbywa się za pośrednictwem iteratora. Podstawowe operacje na liście to dodaj element (***add(E element)***) oraz pobierz iterator (***Iterator<E> iterator()***). Obiekty w zbiorze **nie mogą się powtarzać** – tj. ten sam obiekt można dodać do zbioru wielokrotnie, ale będzie on w nim przechowywany **tylko jeden raz**.

Najpopularniejsze implementacje:
HashSet – podstawowa implementacja, wykorzystuje mechanizm hashCode() wbudowany w język Java do organizacji przechowywania
TreeList – przechowuje elementy w postaci drzewa, posortowane, gwarantowana złożoność przy wstawianiu elementów (może być to istotne w przypadku dużych zbiorów)

Rozwiązanie 3 - Set

1. Tworzenie:

```
User user1 = new User( name: "Jan",  surname: "Kowalski");
User user2 = new User( name: "Tomasz",  surname: "Frek");
User user3 = new User( name: "Dominika",  surname: "Olszewska");
User user4 = new User( name: "Karolina",  surname: "Maźkowska");
User user5 = new User( name: "Laura",  surname: "Koliber");

//...
```

```
Set<User> users = new HashSet<>();
users.add(new User( name: "Jan",  surname: "Kowalski"));
users.add(new User( name: "Tomasz",  surname: "Frek"));
users.add(new User( name: "Dominika",  surname: "Olszewska"));
users.add(new User( name: "Karolina",  surname: "Maźkowska"));
users.add(new User( name: "Laura",  surname: "Koliber"));
```

Rozwiązanie 3 - Set

2. Przetwarzanie - wyświetlanie:

```
System.out.println(user1.getName() + " " + user1.getSurname());
System.out.println(user2.getName() + " " + user2.getSurname());
System.out.println(user3.getName() + " " + user3.getSurname());
System.out.println(user4.getName() + " " + user4.getSurname());
System.out.println(user5.getName() + " " + user5.getSurname());

//...
```

```
for (int i = 0; i < users.size(); i++) {
    System.out.println(users.get(i).getName() + " " + users.get(i).getSurname());
}
```

```
for (User user : users) {
    System.out.println(user.getName() + " " + user.getSurname());
}
```

inny wariant

```
users.forEach(user -> System.out.println(user.getName() + " " + user.getSurname()));
```

Rozwiązanie 4 - Map

java.util.Map

Mapy, choć nie są formalnie kolekcjami z punktu widzenia języka Java (nie są typu Collection), także służą do przechowywania elementów. Wyróżnia je to, że przechowują nie tyle same elementy, co **mapowanie klucz-wartość**, tzn. jeden obiekt (klucz) wskazuje na inny obiekt (wartość). W przypadku map mamy przede wszystkim operację dodawania (***put(K key, V value)***) oraz pobierania elementu (***V get(K key)***). Obiekty w mapie **mogą się powtarzać**, jeśli są wartościami. **Klucze muszą być unikalne**. Wartości i klucze mogą być dowolnego typu.

Najpopularniejsze implementacje:
HashMap – mapa, której właściwości są podobne do HashSet'a, kolejność oraz przechowywanie wynikają z implementacji funkcji hashCode()
TreeMap – elementy są przechowywane w formie posortowanej (wg klucza)

Rozwiązanie 4 - Map

1. Tworzenie:

```
User user1 = new User( name: "Jan",  surname: "Kowalski");
User user2 = new User( name: "Tomasz",  surname: "Frek");
User user3 = new User( name: "Dominika",  surname: "Olszewska");
User user4 = new User( name: "Karolina",  surname: "Maźkowska");
User user5 = new User( name: "Laura",  surname: "Koliber");

//...
```

```
Map<String, User> users = new HashMap<>();
users.put("pierwszy", new User( name: "Jan",  surname: "Kowalski"));
users.put("drugi", new User( name: "Tomasz",  surname: "Frek"));
users.put("trzeci", new User( name: "Dominika",  surname: "Olszewska"));
users.put("czwarty", new User( name: "Karolina",  surname: "Maźkowska"));
users.put("piąty", new User( name: "Laura",  surname: "Koliber"));
```

Rozwiązanie 4 - Map

2. Przetwarzanie - wyświetlanie:

```
System.out.println(user1.getName() + " " + user1.getSurname());
System.out.println(user2.getName() + " " + user2.getSurname());
System.out.println(user3.getName() + " " + user3.getSurname());
System.out.println(user4.getName() + " " + user4.getSurname());
System.out.println(user5.getName() + " " + user5.getSurname());

//...
```

```
for (Map.Entry<String, User> entry : users.entrySet()) {
    System.out.println(entry.getKey() + "/" + entry.getValue().getName() + entry.getValue().getSurname());
}
```

inny wariant

```
users.forEach((key, user) -> System.out.println("Klucz" + key +"Imię" + user.getName() + " " + user.getSurname()));
```

Streams

Strumień (Steam) reprezentuje **sekwencje elementów** pochodzących od **źródła** na której można **wykonywać operacje agregacji.**

sekwencje elementów – Strumień udostępnia interfejs do zbioru elementów danego typu, strumień **nie przechowuje ich, pozyskuje je na żądanie**

źródłem elementów są **kolekcje, tablice,** I/O

operacje agregujące – operacje z programowania **funkcyjnego**

Operacje te mogą być:
- pośrednie - możemy układać w łańcuchy metod (*filter, map, reduce, find, match, sorted*),
- końcowe - zwracające wynik, lub nie (kończą łańcuch).

Wprowadzone w Javie 8.

Streams - przykłady

Wybrać z listy imiona użytkowników (alfabetycznie), których nazwisko zaczyna się na K.

```
List<User> users = new ArrayList<>();
users.add(new User( name: "Jan",  surname: "Kowalski"));
users.add(new User( name: "Tomasz",  surname: "Frek"));
users.add(new User( name: "Dominika",  surname: "Olszewska"));
users.add(new User( name: "Karolina",  surname: "Maźkowska"));
users.add(new User( name: "Laura",  surname: "Koliber"));
users.add(new User( name: "Ania",  surname: "Kamień"));
```

Streams - przykłady

Wariant zwykły (bez Streams):

```
//Lista tymczasowa
List<User> usersFiltered = new ArrayList<>();

//filtrowanie (userzy z nazwiskiem zaczynającym się na K)
for (User user : users) {
    if (user.getSurname().startsWith("K")) {
        usersFiltered.add(user);
    }
}

//sortowanie po imieniu
usersFiltered.sort(new Comparator<User>() {
    @Override
    public int compare(User o1, User o2) {
        return o1.getName().compareTo(o2.getName());
    }
});

//mapowanie na listę imion
List<String> names = new ArrayList<>();
for(User user: usersFiltered){
    names.add(user.getName());
}
```

vs Streams (wariant długi):

```
List<String> names = users.stream()
        .filter(user -> user.getSurname().startsWith("K"))
        .sorted((u1, u2) -> u1.getName().compareTo(u2.getName()))
        .map(user -> user.getName())
        .collect(Collectors.toList());
```

Streams - przykłady

Wariant zwykły (bez Streams):

```
//Lista tymczasowa
List<User> usersFiltered = new ArrayList<>();

//filtrowanie (userzy z nazwiskiem zaczynającym się na K)
for (User user : users) {
    if (user.getSurname().startsWith("K")) {
        usersFiltered.add(user);
    }
}

//sortowanie po imieniu
usersFiltered.sort(new Comparator<User>() {
    @Override
    public int compare(User o1, User o2) {
        return o1.getName().compareTo(o2.getName());
    }
});

//mapowanie na listę imion
List<String> names = new ArrayList<>();
for(User user: usersFiltered){
    names.add(user.getName());
}
```

vs Streams (wariant krótki):

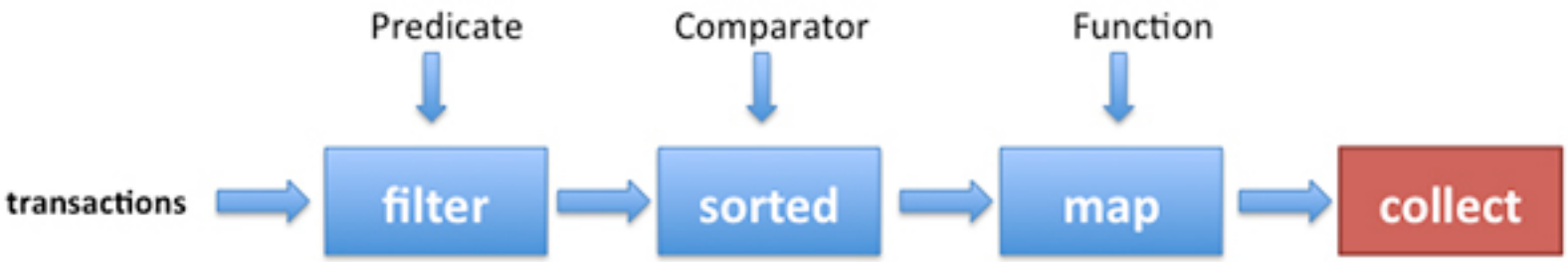
```
List<String> names = users.stream()
        .filter(user -> user.getSurname().startsWith("K"))
        .sorted(Comparator.comparing(User::getName))
        .map(User::getName)
        .collect(toList());
```

Streams - przykłady

Streams

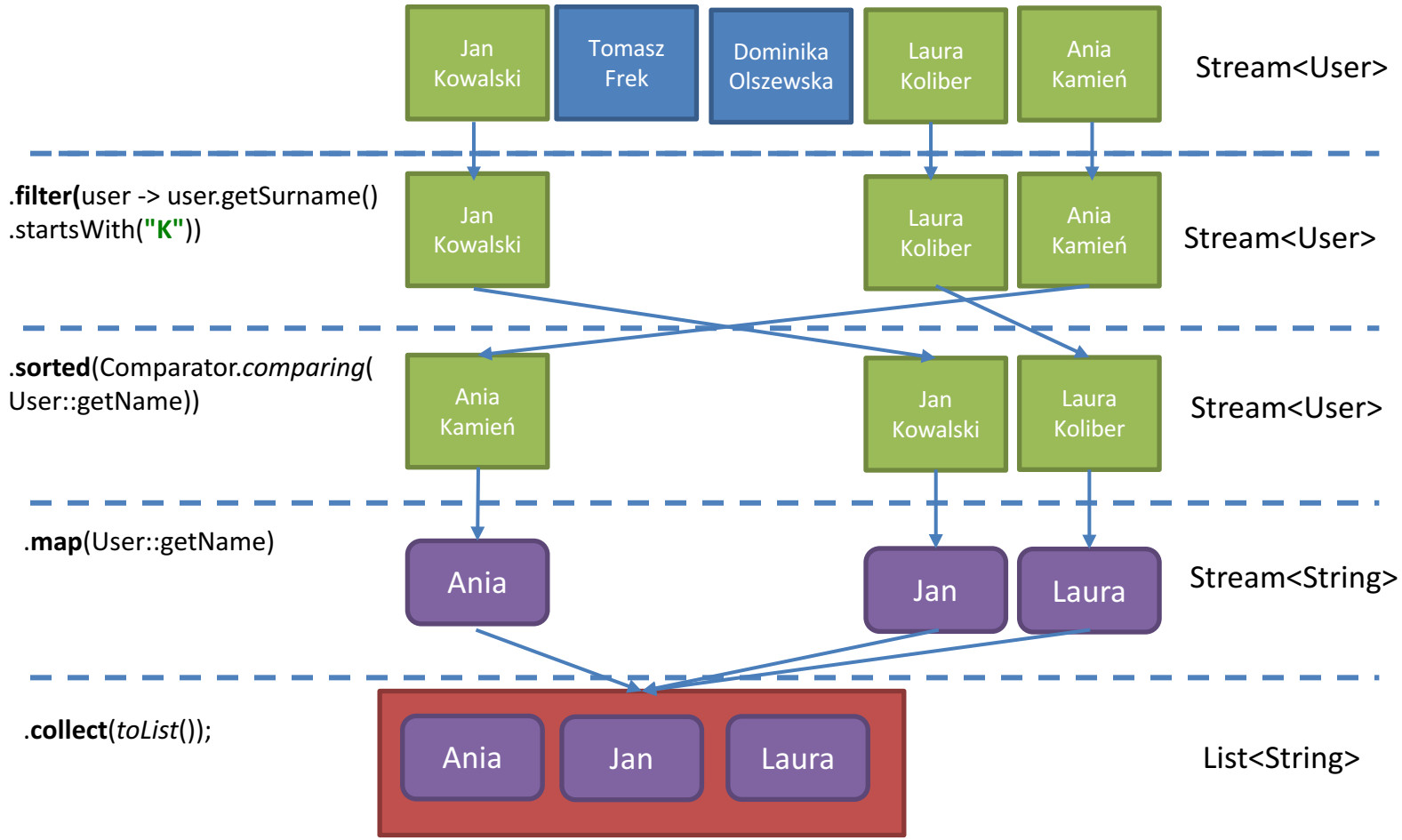
```
List<String> names = users.stream()
        .filter(user -> user.getSurname().startsWith("K"))
        .sorted(Comparator.comparing(User::getName))
        .map(User::getName)
        .collect(toList());
```

=



Streams - przykłady

Transactions Streams



Literatura

http://docs.oracle.com/javase/tutorial/collections/intro/index.html
http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html
http://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html