

# Iteratory

## Czego się dowiesz

Czym są iteratory,  
kiedy z nich korzystać.

## Wstęp

Iteratory to obiekty, które pozwalają na przeglądanie kolekcji obiektów w określony sposób. Przed Javą 1.5, kiedy nie istniała jeszcze pętla `for each`, był to sposób na wygodne przeglądanie obiektów w listach, a szczególnie w zbiorach, które nie udostępniają metody `get()` i nie można przeglądać ich elementów np. w tradycyjnej pętli. Iterator definiujemy poprzez zaimplementowanie interfejsu `Iterator`, który od Javy 1.5 jest także generyczny.

Nawet pomimo tego, że zbiory nie mają metody takiej jak `get(int index)`, to dzięki iteratorowi możliwe jest przeglądnięcie ich wszystkich elementów w kolejności zależnej od typu kolekcji.

Trzy najważniejsze metody, które ten interfejs definiuje to:

- `hasNext()`** - sprawdza, czy w iteratorze jest jeszcze jakiś kolejny element (najczęściej wykorzystywana w warunku pętli),
- `next()`** - przechodzi do kolejnego elementu iteratora i go zwraca,
- `remove()`** - usuwa element, na który aktualnie wskazuje iterator.

Stwórzmy przykład klasy, w której do zbioru typu `TreeSet` dodamy 5 losowych wartości typu całkowitoliczbowego, a następnie po nich przejdziemy za pomocą iteratora.

*TreeSetIterator.java*

```
import java.util.Iterator;  
import java.util.Set;  
import java.util.TreeSet;
```

```
class TreeSetIterator {  
    public static void main(String[] args) {  
        Set<Integer> numbers = new TreeSet<>();  
        numbers.add(45);  
        numbers.add(3);  
        numbers.add(21);  
        numbers.add(150);  
        numbers.add(1);  
  
        Iterator<Integer> numIterator = numbers.iterator();  
        while (numIterator.hasNext()) {  
            int number = numIterator.next();  
            System.out.println(number);  
        }  
    }  
}
```

Obiekt typu `Iterator` zwracany jest przez metodę `iterator()` dowolnej kolekcji, która implementuje pośrednio lub bezpośrednio interfejs `Iterable`. Ponieważ listy, zbiory i kolejki implementują interfejs `Collection`, a ten interfejs dziedziczy po interfejsie `Iterable`, to metoda `iterator()` jest dostępna we wszystkich tych kolekcjach. Metoda `hasNext()` sprawdza, czy istnieje kolejny element kolekcji, ale do niego nie przechodzi, natomiast metoda `next()` przechodzi do kolejnego elementu i go zwraca, dzięki czemu możemy go przypisać np. do zmiennej.



```
src — -bash — 80x8  
[src$ javac TreeSetIterator.java  
[src$ java TreeSetIterator  
1  
3  
21  
45  
150  
src$
```

## Usuwanie elementów z kolekcji

Przy korzystaniu z iteratorów należy uważać na jeden istotny problem. Dotyczy to zarówno sytuacji, gdy wykorzystujesz iterator do przeglądania kolekcji za pomocą dowolnej pętli, jak i wykorzystywania pętli `for each`, która niejawnie również wykorzystuje iterator.

Przeglądając kolekcję z wykorzystaniem iteratora w pętli, nie możesz wywołać metody `remove()` bezpośrednio na kolekcji. Zostanie wtedy wygenerowany wyjątek `ConcurrentModificationException`, który mówi o tym, że iterator jest nieaktualny względem kolekcji na którą wskazuje. Błąd ten obrazuje poniższy przykład:

### *IteratorRemoverException.java*

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

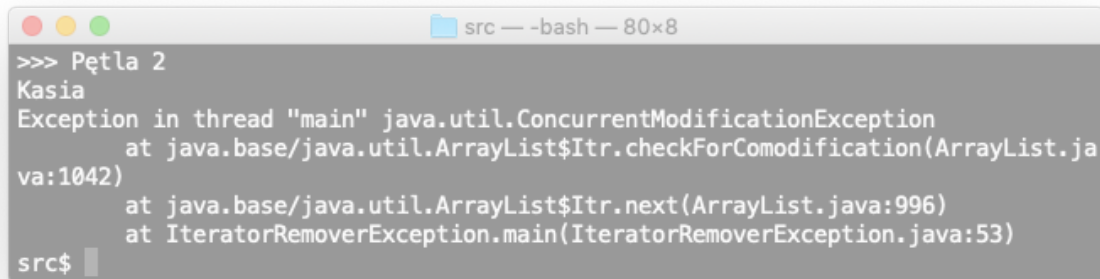
class IteratorRemoverException {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Kasia");
        names.add("Basia");
        names.add("Kajtek");
        names.add("Wojtek");
        names.add("Maniek");

        // pętla ok, bo nie korzystamy z iteratora
        System.out.println(">>> Pętla 1");
        for (int i = 0; i < names.size(); i++) {
            String name = names.get(i);
            System.out.println(name);
            if (name.equals("Basia")) {
                names.remove(name);
                i--;
            }
        }

        Iterator<String> namesIterator = names.iterator();
        System.out.println(">>> Pętla 2");
        while (namesIterator.hasNext()) {
            String name = namesIterator.next();
            System.out.println(name);
            if (name.equals("Kasia")) {
                // błąd – iterujemy po kolekcji za pomocą iteratora, a usuwamy
                // obiekt metodą remove() bezpośrednio z kolekcji
                names.remove(name);
            }
        }

        System.out.println(">>> Pętla 3");
        for (String name : names) {
            System.out.println(name);
            if (name.equals("Maniek")) {
                // błąd – iterujemy po kolekcji za pomocą niejawnego iteratora
                // petli for-each, a usuwamy
                // obiekt metodą remove() bezpośrednio z kolekcji
                names.remove(name);
            }
        }
    }
}
```

Pierwsza z pętli nie spowoduje błędu. Iterujemy w niej po elementach kolekcji, bez wykorzystania iteratora. W 2 i 3 pętli wykorzystujemy jednak jawnie lub niejawnie iteratory, zatem usuwanie elementów bezpośrednio z kolekcji spowoduje błąd. Jeżeli chcemy usuwać elementy korzystając z dobrodziejstw iteratorów, powinniśmy wykorzystywać metodę **remove()** interfejsu **Iterator**.



```
>>> Pętla 2
Kasia
Exception in thread "main" java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1042)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:996)
    at IteratorRemoverException.main(IteratorRemoverException.java:53)
src$
```

Poprawiona fragment powyższej klasy wygląda następująco:

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

class IteratorRemoverException {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Kasia");
        names.add("Basia");
        names.add("Kajtek");
        names.add("Wojtek");
        names.add("Maniek");

        Iterator<String> namesIterator = names.iterator();
        while (namesIterator.hasNext()) {
            String name = namesIterator.next();
            System.out.println(name);
            if (name.equals("Kasia")) {
                namesIterator.remove();
            }
        }
    }
}
```

Tym razem usuwamy element przy pomocy metody `remove()` iteratora, a nie kolekcji - problem zatem nie występuje.

## Kiedy korzystać z iteratorów

Z iteratorów korzystaj zawsze wtedy, kiedy chcesz przeglądać dowolną kolekcję. Nawet jeśli jest to lista udostępniająca metodę `get()`, to w przypadku listy wiązanej wykorzystanie iteratora może podnieść szybkość jej przeglądania. Dodatkowo preferuj pętlę `for each` zamiast tradycyjnej pętli `for`, wtedy wykorzystywany będzie iterator, pomimo że jawnie go nigdzie nie zapiszemy.

Dodatkowo istnieje możliwość zdefiniowania swojego iteratora, co może być przydatne np. w klasach, w których przechowujemy kolekcje obiektów. Klasa taka powinna implementować wtedy interfejs `Iterable`, a klasę iteratora najwygodniej będzie zdefiniować jako klasę wewnętrzną.

W Javie 8 wprowadzono istotne usprawnienia, takie jak strumienie i wyrażenia `lambda`, które sprawiają, że iteratory nie są już tak często potrzebne. Dowiesz się o nich więcej w kolejnych lekcjach.