

# Zbiory

## Czego się dowiesz

- Czym są zbiory,
- kiedy wykorzystywać klasę `TreeSet`,
- kiedy wykorzystywać klasę `HashSet`.

## Wstęp

Zbiory w Javie (klasy implementujące interfejs `Set`) są swego rodzaju odzwierciedleniem matematycznej ich definicji, zgodnie z którą jest to nieuporządkowany zestaw różnych obiektów.

Najważniejszą cechą zbiorów jest to, że służą one **do przechowywania unikalnych obiektów**. Jeżeli spróbujesz dodać do nich dwa razy tę samą wartość, to w kolekcji nie zobaczysz duplikatów. Ponieważ zbiory, tak jak inne kolekcje, mogą przechowywać tylko obiekty, to musisz zagwarantować, że posiadają one poprawnie zaimplementowaną metodę `equals()` i `hashCode()`. To właśnie one są wykorzystywane do sprawdzania unikalności. Błędna implementacja lub brak tych dwóch metod sprawi, że zbiory mogą zachowywać się w nieoczekiwany sposób i duplikaty będą dopuszczalne.

## Zbiory uporządkowane `TreeSet`

Jeżeli ważne jest dla Ciebie, aby elementy dodawane do zbioru były automatycznie ustawiane w naturalnym porządku (lub zdefiniowanym przez komparator) to `TreeSet` będzie dobrym wyborem. Wewnętrzna reprezentacja `TreeSet` w Javie to **drzewo czerwono-czarne**. Raczej nie musi cię to specjalnie interesować, ale gdy w przyszłości przyjdzie ci zgłębić wiedzę dotyczącą struktur danych, to będziesz musiał do tego zagadnienia wrócić.

Najważniejsze metody, które znajdziesz w interfejsie `Set`, a więc także w klasie `TreeSet`, która ten interfejs implementuje to:

- **`add(E e)`** - dodaje do zbioru obiekt, który jest zgodny ze typem zbioru,
- **`contains(E e)`** - sprawdza, czy w zbiorze znajduje się podany jako argument element,
- **`remove(E e)`** - usuwa ze zbioru element podany jako argument,
- **`size()`** - zwraca ilość elementów w tym zbiorze,
- (pełną listę metod znajdziesz [w dokumentacji](#)).

`TreeSet` posiada też dodatkowe metody, które wynikają z tego, że elementy w nim są posortowane:

- **`first()` / `last()`** - zwraca pierwszy lub ostatni element zbioru (zgodny z porządkiem sortowania).

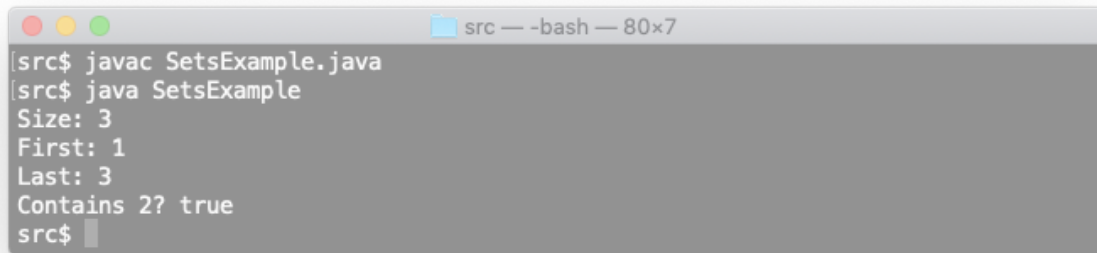
Nie znajdziesz tutaj metody typu `get()` znanej z list, więc iterowanie za pomocą tradycyjnej pętli `for`, czy `while`, nie będzie możliwe tak jak poprzednio. Wynika to z tego, że sposób przechowywania elementów w zbiorze nie jest taki w jakim te elementy umieszczaliśmy. Jeśli chcesz iterować po wszystkich elementach zbioru (abstrahując od kolejności) to musisz skorzystać z `Iteratora`, któremu poświęcimy osobną lekcję, lub z pętli `for each`.

Korzystanie z klasy `TreeSet` jest podobne jak z list:

### *SetsExample.java*

```
import java.util.TreeSet;

class SetsExample {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        set.add(1);
        set.add(1);
        set.add(2);
        set.add(3);
        System.out.println("Size: " + set.size());
        System.out.println("First: " + set.first());
        System.out.println("Last: " + set.last());
        System.out.println("Contains 2? " + set.contains(2));
    }
}
```



```
[src$ javac SetsExample.java]
[src$ java SetsExample]
Size: 3
First: 1
Last: 3
Contains 2? true
[src$ ]
```

Pomimo iż próbowaliśmy dwa razy dodać wartość 1, to zbiór nadal ma rozmiar 3, czyli dodane zostały tylko unikalne obiekty.

Jeżeli chcesz zmienić kolejność w jakiej sortowane są dodawane do `TreeSetu` obiekty, to podczas jego tworzenia możesz przekazać do konstruktora odpowiedni obiekt komparatora.

## HashSet

**HashSet** jest zbiorem nieuporządkowanym i nieposortowanym, nie posiada też nawet specjalnych metod takich jak `first()`, czy `last()`, bo nie mamy żadnej gwarancji co do kolejności w jakiej przechowywane są obiekty. Jedyłą, ale dużą jego zaletą jest to, że operacje dodawania, usuwania i wyszukiwania są w niej bardzo szybkie. Zbiór typu `HashSet` pozwala także w odróżnieniu od `TreeSet` na umieszczenie wartości `null`.

`HashSet` jak sama nazwa wskazuje korzysta pod spodem z tablic mieszających (haszowanych), a one wymagają do poprawnego, wydajnego działania metody `hashCode()`. To między innymi tutaj wymagane jest spełnienie kontraktu pomiędzy metodami `equals()` i `hashCode()`, który dla przypomnienia mówi o tym, że jeśli metoda `equals()` dla dwóch obiektów zwraca `true`, to metoda `hashCode()` wywołana na tych obiektach powinna zwrócić identyczną wartość. Jeżeli kontrakt ten nie będzie spełniony, to `HashSet` może nie działać poprawnie i np. dopuszczać do przechowywania duplikatów.

Stwórzmy klasę `Person` z metodami `equals()` i `hashCode()`, a następnie przetestujmy dodawanie kilku obiektów do zbioru typu `HashSet`.

### *Person.java*

```
import java.util.Objects;

class Person {
    private String firstName;
    private String lastName;

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

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;
        Person person = (Person) o;
        return Objects.equals(firstName, person.firstName) &&
            Objects.equals(lastName, person.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}
```

### *PersonSet.java*

```
import java.util.HashSet;

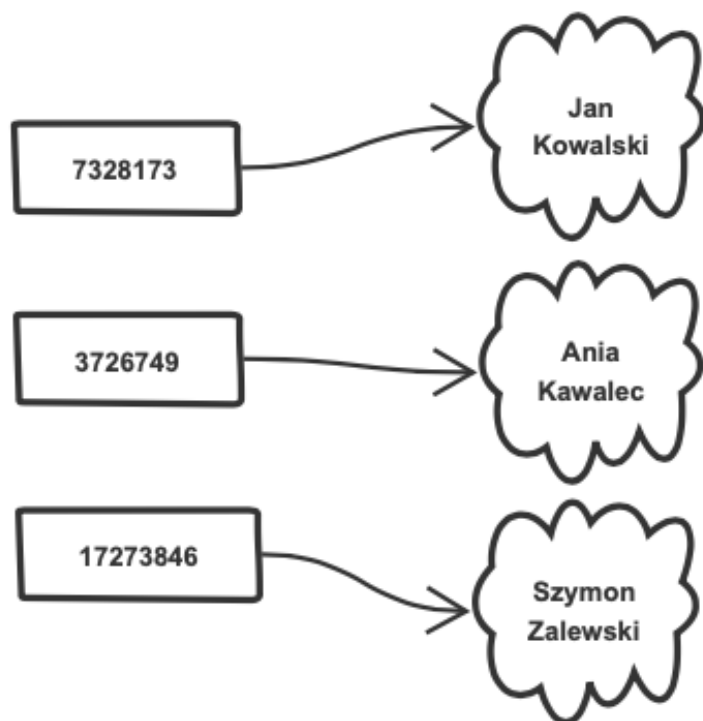
class PersonSet {
    public static void main(String[] args) {
        HashSet<Person> persons = new HashSet<>();

        persons.add(new Person("Jan", "Kowalski"));
        persons.add(new Person("Jan", "Kowalski"));
        persons.add(new Person("Ania", "Kawalec"));
        persons.add(new Person("Ania", "Kawalec"));
        persons.add(new Person("Szymon", "Zalewski"));

        System.out.println("Persons.size() " + persons.size());
        System.out.println("contains Jan Kowalski? " +
            persons.contains(new Person("Jan", "Kowalski")));
        persons.remove(new Person("Ania", "Kawalec"));
        System.out.println("Persons.size() " + persons.size());
    }
}
```

Zauważ jedną ważną rzecz. Przy usuwaniu obiektów w metodzie `remove()` przekazujemy tak naprawdę inny obiekt niż ten, który przechowujemy w kolekcji (przy każdej kreacji obiektu użyliśmy słowa `new`, więc ich porównanie za pomocą operatora `==` zwróciłoby `false`), jednak obiekt zostaje usunięty, ponieważ metoda `equals()` zaimplementowana w klasie `Person` zwróciła `true`.

Wizualnie możesz sobie wyobrazić to w taki sposób, że do wartości wyliczonej przez `hashCode()` przypisywany jest obiekt. Dzięki temu w prosty i wydajny sposób, możemy sprawdzić, czy jakiś obiekt istnieje w naszym zbiorze, bez konieczności przeszukiwania go obiekt po obiekcie.



## LinkedHashSet

**LinkedHashSet** jest zbiorem bardzo podobnym do `HashSet`, jednak dzięki dodatkowej wewnętrznej reprezentacji w postaci listy wiązanej, dodając kolejne elementy do zbioru uzyskujemy pewność, że w tej samej kolejności będziemy mogli po nich iterować. W przypadku zwykłego `HashSet` iteracja odbywa się w przypadkowej kolejności. Tak samo jako w pozostałych zbiorach, do iteracji trzeba jednak wykorzystać iterator lub pętlę `for each`.

## Zbiory i Java 9

W Javie 9 do interfejsu `Set` podobnie jak przy listach dodano metody `of()` pozwalające tworzyć zbiory na podstawie przekazanych argumentów. Również w tym przypadku w wyniku otrzymamy zbiór niemodyfikowalny, czyli nie możemy później do niego dodawać nowych, ani usuwać już istniejących elementów.

```
import java.util.Set;
```

```
class SetTest {  
    public static void main(String[] args) {  
        Set<String> names = Set.of("Marek", "Kasia", "Karol", "Basia");  
        for (String name : names) {  
            System.out.println(name);  
        }  
    }  
}
```

## Podsumowanie zbiorów

- Zbiorów, czyli klas implementujących interfejs `Set` używaj wtedy, kiedy chcesz zapewnić, że przechowujesz tylko **unikalne** obiekty.
- Zbiory najczęściej wykorzystuje się wtedy, kiedy nie obchodzi nas kolejność dodawania obiektów i odwoływanie się do nich po indeksach, ani iterowanie po całym zbiorze, ale liczy się dla nas szybkie dodawanie, usuwanie i wyszukiwanie wartości.
- Klasy `TreeSet` używaj wtedy, kiedy zależy ci na porządkowaniu dodawanych elementów w kolejności naturalnego porządku.
- Klasy `HashSet` używaj wtedy, kiedy liczy się dla Ciebie praktycznie tylko szybkość dodawania, usuwania i wyszukiwania obiektów w zbiorze, ale nie liczy się dla Ciebie kolejność iterowania po nich.
- Klasy `LinkedHashSet` użyj w sytuacji podobnej do `HashSet`, jednak z dodatkową zaletą w postaci możliwości zachowania porządku elementów.
- Do iterowania po zbiorze konieczne jest zastosowanie iteratora lub pętli `for-each`.