

# Kolejki

## Czego się dowiesz

Czym są kolejki,  
do czego wykorzystywać kolejki.

## Kolejki

Każdy kto był kiedykolwiek w sklepie na zakupach z pewnością wie jak wygląda kolejka przy kasie. Zasada działania kolejek w Javie jest bardzo podobna - jeżeli jako pierwsi podeszliśmy do kasy (obiekt został dodany jako pierwszy) to również zostaniemy jako pierwsi obsłużeni (obiekt zostanie pobrany jako pierwszy z kolejki).



Zgodnie z lekcją wprowadzającą, gdzie pokazywaliśmy hierarchię dziedziczenia, kolejki reprezentowane są przez interfejs **Queue** (czytaj "kju"). Istnieją dwie podstawowe implementacje tego interfejsu. Pierwszą już poznaliśmy i jest nią `LinkedList`. Druga to `PriorityQueue`. Oprócz nich znajdziemy jeszcze m.in. `ArrayBlockingQueue`, w której możemy określić maksymalną liczbę elementów w kolejce.

Kolejka priorytetowa pozwala dodatkowo określić priorytet i jeżeli jest on wyższy od elementów, które są już w kolejce, to element ten zostanie przesunięty "wyżej" w kolejce (analogicznie, gdy w sklepie

przepuszczamy matkę z dzieckiem lub osobę niepełnosprawną). Kolejność ta ustalana jest przez naturalny porządek elementów wyznaczany przez interfejs Comparable lub odpowiedni komparator przekazany w konstruktorze kolejki.

Trzy najważniejsze metody, które należy zapamiętać to:

- offer(E e)** - dodaje do kolejki element zgodny z zadeklarowanym typem generycznym (można też wywołać metodę **add()**),
- peek()** - pobiera element z kolejki, ale go nie usuwa,
- poll()** - pobiera element z kolejki i go z niej usuwa. Zwraca null, gdy kolejka jest pusta.

Oprócz tego mamy też metody wynikające z interfejsu Collection takie jak **size()**, **contains()**, **remove()**, czy **clear()**.

Dobrym zastosowaniem kolejki mogłaby być np. lista TODO (rzeczy do zrobienia), gdzie ustalamy sobie plan dnia i dodajemy zadania z konkretnymi priorytetami. Inne dobre zastosowanie to np. panel helpdesk. Jeżeli ktoś zgłasza się do nas o pomoc dodawany jest do kolejki i obsługiwany w pierwszej kolejności, przed osobami, które zgłaszają się później, chyba, że mamy klientów premium, wtedy oni będą obsługiwani w pierwszej kolejności.

Definicja kolejki wygląda następująco:

```
LinkedList<Typ> queue = new LinkedList<>();  
PriorityQueue<Typ> queue = new PriorityQueue<>();
```

Podobnie jak przy innych kolekcjach, warto także tutaj rozważyć posługiwanie się interfejsami do deklaracji typu:

```
Queue<Typ> queue = new LinkedList<>();  
Queue<Typ> queue = new PriorityQueue<>();
```

Prosty przykład obsługi kolejki klientów mógłby wyglądać tak jak poniżej.

Klasa Client definiuje klienta - ceniąc anonimowość potrzebujemy jedynie informację o nicku użytkownika:

*Client.java*

```
class Client {
    private String nickname;

    public Client(String nickname) {
        this.nickname = nickname;
    }

    public String getNickname() {
        return nickname;
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

    @Override
    public String toString() {
        return nickname;
    }
}
```

Definicja metod `hashCode()`, `equals()`, czy implementowanie interfejsu `Comparable` jest zbędne, ponieważ standardowa kolejka nie dba o unikalność obiektów ani ich sortowanie (zachowuje się jak lista, ale z możliwością pobierania elementów tylko z jej początku).

```
import java.util.LinkedList;
import java.util.Queue;

class ClientService {
    public static void main(String[] args) {
        Queue<Client> clientQueue = new LinkedList<>();

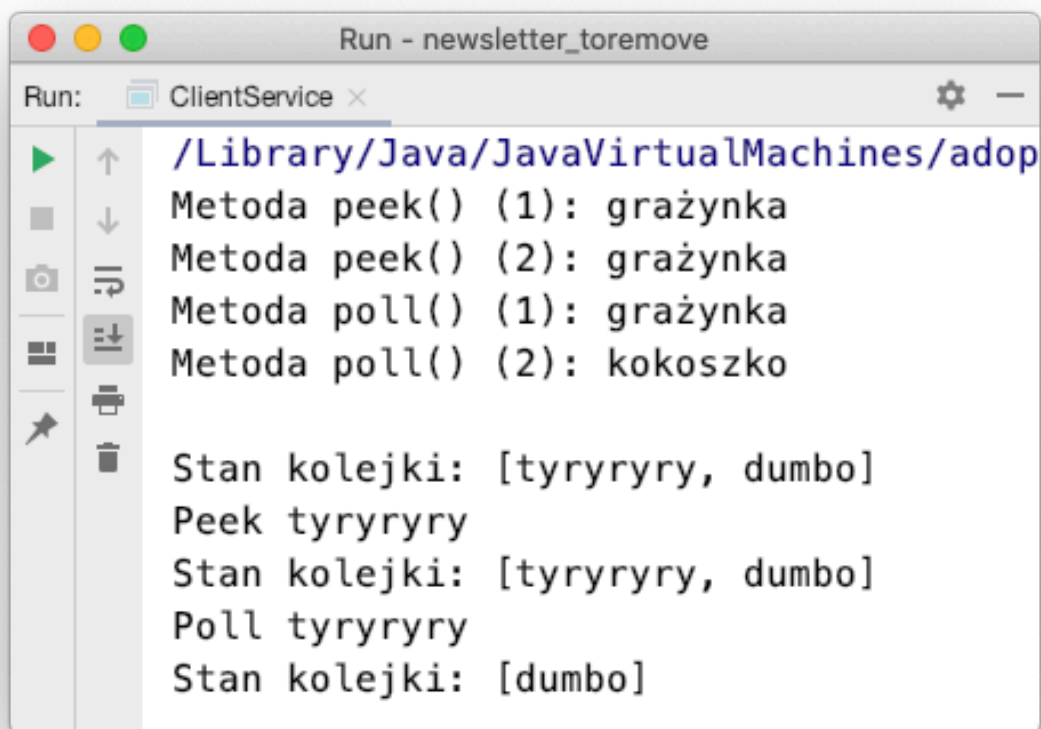
        clientQueue.offer(new Client("grazyinka"));
        clientQueue.offer(new Client("kokoszek"));
        clientQueue.offer(new Client("tyryryry"));
        clientQueue.offer(new Client("dumbo"));

        System.out.println("Metoda peek() (1): " + clientQueue.peek());
        System.out.println("Metoda peek() (2): " + clientQueue.peek());

        System.out.println("Metoda poll() (1): " + clientQueue.poll());
        System.out.println("Metoda poll() (2): " + clientQueue.poll() + "\n");

        System.out.println("Stan kolejki: " + clientQueue);
        System.out.println("Peek " + clientQueue.peek());
        System.out.println("Stan kolejki: " + clientQueue);
        System.out.println("Poll " + clientQueue.poll());
        System.out.println("Stan kolejki: " + clientQueue);
    }
}
```

Po dodaniu 4 obiektów do kolejki przy pomocy metody `offer()`, zaczynamy je pobierać korzystając z metod `peek()` i `poll()`. Po pobraniu za pomocą pierwszych dwóch metod (`peek()`) widzimy, że zwracany jest pierwszy z dodanych obiektów `Client`. Wywołanie kolejnych metod `poll()` powoduje jednocześnie usuwanie obiektów z kolejki zgodnie z kolejnością ich dodawania.



```
Run: ClientService x
/Library/Java/JavaVirtualMachines/adop
Metoda peek() (1): grażynka
Metoda peek() (2): grażynka
Metoda poll() (1): grażynka
Metoda poll() (2): kokoszeko

Stan kolejki: [tyryryry, dumbo]
Peek tyryryry
Stan kolejki: [tyryryry, dumbo]
Poll tyryryry
Stan kolejki: [dumbo]
```

Jeżeli prowadzimy sklep, to możemy jednak wprowadzić pewne usprawnienia. Załóżmy, że w sklepie oferujemy infolinię pomocy, z której korzysta wiele osób. W pierwszej kolejności chcemy obsługiwać zgłoszenia klientów, którzy dokonali u nas zakupu najwięcej razy. W takiej sytuacji możemy wykorzystać kolejkę priorytetową. Do klasy `Client` dodajmy dodatkowe pole, które będzie oznaczało liczbę dokonanych zakupów.

Kolejka priorytetowa opiera się na sortowaniu elementów, więc musimy wyznaczyć porządek dla obiektów. Można to zrobić albo implementując w klasie `Client` interfejs `Comparable`, albo dostarczając do konstruktora `PriorityQueue` odpowiedni komparator. My wykorzystajmy pierwszą z tych metod.

*Client.java*

```
class Client implements Comparable<Client> {
    private String nickname;
    private int ordersNumber;

    public Client(String nickname, int ordersNumber) {
        this.nickname = nickname;
        this.ordersNumber = ordersNumber;
    }

    public String getNickname() {
        return nickname;
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

    public int getOrdersNumber() {
        return ordersNumber;
    }

    public void setOrdersNumber(int ordersNumber) {
        this.ordersNumber = ordersNumber;
    }

    @Override
    public String toString() {
        return "Client{" +
            "nickname='" + nickname + '\'' +
            ", ordersNumber=" + ordersNumber +
            '}';
    }

    @Override
    public int compareTo(Client c) {
        return -Integer.compare(this.ordersNumber, c.ordersNumber);
    }
}
```

W metodzie `compareTo()` zmieniamy znak wartości zwracanej przez metodę `Integer.compare()`, ponieważ chcemy sortować malejąco (klienci z największą liczbą zamówień będą obsługiwani w pierwszej kolejności).

Klasa `ClientService` ulega nieznacznej modyfikacji. Zmieniamy w niej jedynie rodzaj używanej kolejki oraz przy tworzeniu obiektów `Client` dodajemy brakujący argument do konstruktorów.

*ClientService.java*

```
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

class ClientService {
    public static void main(String[] args) {
        Queue<Client> clientQueue = new PriorityQueue<>();

        clientQueue.offer(new Client("grazynka", 3));
        clientQueue.offer(new Client("kokoszek", 1));
        clientQueue.offer(new Client("tyryryry", 5));
        clientQueue.offer(new Client("dumbo", 2));

        System.out.println("Metoda peek() (1): " + clientQueue.peek());
        System.out.println("Metoda peek() (2): " + clientQueue.peek());

        System.out.println("Metoda poll() (1): " + clientQueue.poll());
        System.out.println("Metoda poll() (2): " + clientQueue.poll() + "\n");

        System.out.println("Stan kolejki: " + clientQueue);
        System.out.println("Peek " + clientQueue.peek());
        System.out.println("Stan kolejki: " + clientQueue);
        System.out.println("Poll " + clientQueue.poll());
        System.out.println("Stan kolejki: " + clientQueue);
    }
}
```

```
Run - newsletter_toremove
ClientService
/Library/Java/JavaVirtualMachines/adoptopenjdk-13.jdk/Contents/Home/bin/java "-javaagent:/Applications/
Metoda peek() (1): Client{nickname='tyryryry', ordersNumber=5}
Metoda peek() (2): Client{nickname='tyryryry', ordersNumber=5}
Metoda poll() (1): Client{nickname='tyryryry', ordersNumber=5}
Metoda poll() (2): Client{nickname='grazynka', ordersNumber=3}

Stan kolejki: [Client{nickname='dumbo', ordersNumber=2}, Client{nickname='kokoszek', ordersNumber=1}]
Peek Client{nickname='dumbo', ordersNumber=2}
Stan kolejki: [Client{nickname='dumbo', ordersNumber=2}, Client{nickname='kokoszek', ordersNumber=1}]
Poll Client{nickname='dumbo', ordersNumber=2}
Stan kolejki: [Client{nickname='kokoszek', ordersNumber=1}]
```

Po uruchomieniu programu widzimy, że obiekty w kolejce są posortowane.