

Notatki - Typy generyczne

Spis treści

Po co są typy generyczne?	1
Własna klasa z typem generycznym	2
Pierwsza dygresja	3
Druga dygresja	3
Type erasure	4
Czego z typami generycznymi zrobić się nie da?	4
Generic w metodach statycznych	5
Wildcards	5
Unbounded wildcard	7
Upper-Bounded Wildcards	8
Lower-Bounded Wildcard	10
Podsumowanie	12
PECS	12

W tej części notatek opowiemy o tym czym są **Generic Types**, jakim celu się je stosuje i jakie są dostępne warianty.

Po co są typy generyczne?

Wyobraź sobie, że mamy taki przypadek, że chcemy żeby z klasy którą napisaliśmy mogły korzystać tylko określone typy. Przykładowo, wyobraź sobie, że tworzymy klasę `Bag` i chcemy w niej przechowywać tylko Stringi. Częściej spotkasz się z przykładem, w którym będziemy mieli listę i w niej mamy przechowywać np. tylko Integery.

Spójrzmy na przykład, kod poniżej skompiluje się poprawnie, ale spróbuj go uruchomić ☺.

```
public class Example {
    static void printCities(List cities) {
        for (int i = 0; i < cities.size(); i++) {
            String city = (String) cities.get(i);
            System.out.println(city);
        }
    }

    public static void main(String[] args) {
        List cities = new ArrayList();
        cities.add(1020);
        printCities(cities);
    }
}
```

Kod w trakcie działania wyrzuca `ClassCastException`. Dlaczego? Bo Java nie jest w stanie rzutować `Integera` na `String`. Z drugiej strony zdefiniowaliśmy `Listę`, w której można mieć cokolwiek, `Integery`, `Stringi`, `Gruszki` i `Samochody`. Natomiast w metodzie `printCities()` określamy, że każdy element, próbujemy rzutować na `Stringa`. Czy da się rzutować `Gruszkę` na `Stringa`? No nie da się, `Integera` też nie, stąd błąd.

Typy generyczne rozwiązują ten problem. Dzięki nim programista jest w stanie wymusić, żeby w danej klasie można było użyć tylko określonego typu. Czyli klasa `Bag` mogłaby przechowywać tylko `Gruszki`. A klasa `List` tylko `krzesła`. Wtedy taki przykład kodu wyglądałby w ten sposób:

```
public class Example {
    static void printCities(List<String> cities) {
        for (int i = 0; i < cities.size(); i++) {
            // W ten sposób nie musimy tutaj rzutować na typ String
            String city = cities.get(i);
            System.out.println(city);
        }
    }

    public static void main(String[] args) {
        List<String> cities = new ArrayList<>();
        // W tej linijce dostaniemy błąd kompilacji
        // dzięki wykorzystaniu typów generycznych
        cities.add(1020);
        cities.add("Wrocław");
        printCities(cities);
    }
}
```

Własna klasa z typem generycznym

```
public class Bag<T> {

    private T element;

    public void pack(T element) {
        this.element = element;
    }

    public T empty() {
        // teoretycznie ta metoda mogłaby zwracać void,
        // a w implementacji tylko nullować element
        T tmp = element;
        this.element = null;
        return tmp;
    }
}
```

W przykładzie powyżej mamy klasę `Bag`, w której określamy, że będziemy korzystać z typu generycznego `T`. Jeżeli chcielibyśmy jej użyć to można napisać to tak:

```
public class BagCaller {
```

```

public static void main(String[] args) {
    // W tej linijce określamy, że chcemy żeby typem T był String
    Bag<String> stringBag = new Bag<>();
    stringBag.pack("String element");
    String element = stringBag.empty();
    System.out.println(element);

    Bag<Car> carBag = new Bag<>();
    carBag.pack(new Car());
    Car car = carBag.empty();
    System.out.println(car);
}
}

```

Tutaj dodajmy 2 dygresje:

Pierwsza dygresja

Kiedyś definiowanie generyków wyglądało w ten sposób (zwróć uwagę na String w generyku w wywołaniu konstruktora Bag):

```
Bag<String> stringBag = new Bag<String>();
```

Po Javie 7 nie ma już takiej konieczności i możemy to zapisywać w ten sposób:

```
Bag<String> stringBag = new Bag<>();
```

Oczywiście można to zapisywać tak jak kiedyś, ale nie ma już takiej konieczności. IntelliJ pokaże w takiej sytuacji warning.

Druga dygresja

Istnieje pewna konwencja nazywania typów generycznych, tak jak ja napisałem Bag<T>:

Tabela 1. Przykładowe nazwy typów generycznych

Litera	Użycie
K	Klucz w mapie
V	Wartość w mapie
E	Jakiś element
N	Jakiś numer
T	Typ generyczny
R	Typ zwracany
S, U, V	Gdy mamy kilka typów generycznych w klasie

Type erasure

Wiemy już, że narzucając typ generyczny wymuszamy na użytkowniku naszej klasy określenie konkretnego typu generycznego. Czyli tak jak widzieliśmy, zamiast `T` wstawialiśmy przykładowo `String`. Natomiast to się dzieje na etapie kompilacji.

Pytanie co się dzieje w trakcie działania programu? Kompilator podczas kompilacji zamienia wszystkie typy generyczne `T`, `U`, `R`, `E` itp. na klasę `Object`. Czyli po wykonaniu kompilacji, wszędzie typy generyczne zostają zamienione na typ `Object`. Czyli na konkretnym przykładzie, wyglądałoby to tak:

```
public class Bag {  
  
    private Object element;  
  
    public void pack(Object element) {  
        this.element = element;  
    }  
  
    public Object empty() {  
        Object tmp = element;  
        this.element = null;  
        return tmp;  
    }  
  
}
```

Natomiast w miejscu, w którym odwołujemy się do typu `Bag`, wyglądałoby to w ten sposób:

```
Car car = (Car) bag.empty();
```

Czasem pojawiają się pytania, czy typ generyczny oznacza, że Java stworzy sobie kopię klasy z każdym typem generycznym oddzielnie. Czyli jak zdefiniujemy `Bag` ze `Stringiem` i `Integerem` to mielibyśmy oddzielną kopię klasy ze `Stringiem` i oddzielną z `Integerem`. Nie działa to tak 😊.

To zjawisko, proces, o którym tutaj piszę nazywa się **type erasure**.

Dlaczego zostało to zrobione w ten sposób? Bo Java jest kompatybilna wstecznie. Typy generyczne zostały wprowadzone w wersji 5 Javy (upraszczam tutaj konwencję nazewnictwa). Z racji, że Java jest kompatybilna wstecznie, jej twórcy obeszli problem kompilowania starego kodu nowszym kompilatorem właśnie w ten sposób. Bo założeniem Javy jest bycie kompatybilnym ze starszymi wersjami Javy, które nie wykorzystują typów generycznych.

Czego z typami generycznymi zrobić się nie da?

Większość ograniczeń wynika z procesu type erasure.

Nie możemy zrobić takich rzeczy:

- Wywołać konstruktora typu generycznego, czyli np. `new T()`. Nie jest to możliwe, bo w trakcie działania programu byłoby to `new Object()`,

- Nie możemy stworzyć tablicy typu generycznego, powodem jest to, że w trakcie działania programu otrzymalibyśmy tablicę typu Object,
- Wywołać `instanceof`, nie można tego zrobić bo w trakcie działania programu `Bag<Car>` i `Bag<Apple>` byłyby tym samym przez type erasure,
- Nie możemy używać prymitywów jako typów generycznych, natomiast wiemy już, że istnieje coś takiego jak Autoboxing i mamy klasy takie jak np. Integer,
- Stworzyć zmiennej statycznej typu generycznego. Można to wytłumaczyć w ten sposób, że typ generyczny jest powiązany z instancją klasy, a static nie jest.

Generic w metodach statycznych

Powiedzieliśmy sobie, że nie można stworzyć zmiennej statycznej typu generycznego.

Można to obejść w ten sposób, że napiszemy metodę statyczną, w której określimy typ generyczny na potrzeby tylko tej metody. Oczywiście metody instancyjne (niestatyczne) też możemy zapisać w ten sposób.

```
// Tutaj określamy typ generyczny na potrzeby metody,
// który jest używany w argumencie, a metoda zwraca void
public static <T> void method1(T element) {
    System.out.println(element);
}

// Tutaj określamy typ generyczny na potrzeby metody,
// który jest używany w argumencie, a metoda zwraca String
public static <T> String method2(T element) {
    System.out.println(element);
    return element.toString();
}

// Tutaj określamy typ generyczny na potrzeby metody,
// który jest używany w argumencie i jednocześnie jest typem zwracanym
public static <T> T method3(T element) {
    System.out.println(element);
    method1(element);
    return element;
}
```

Dodam w tym miejscu, że typy generyczne są bardzo ważną tematyką, bo na co dzień są używane bardzo często. Odkąd zostały wprowadzone, w nowym kodzie nie pisze się kolekcji, które mogą trzymać cokolwiek. Zawsze określa się konkretny przewidziany typ.

Wildcards

Tak, wiem, jak to brzmi oraz że ciężko będzie te nazwy spamiętać.

W pewnym momencie zauważysz, że z racji wymazywania typów (type erasure) typy generyczne mają pewien minus, który ogranicza ich stosowanie.

```
public class Bag<T> {
```

```

private T element;

public void run(T element) {
    // Taki zapis nie jest możliwy,
    // bo przecież w trakcie działania programu zamiast <T> będziemy mieli Object.
    // Object nie ma przecież zdefiniowanej metody someMethod().
    // Java nie pozwala też do klasy Object takiej metody dopisać.
    // Wtedy każda klasa w programie miałaby przecież metodę someMethod().
    // Ale są języki, które pozwalają na coś podobnego.
    element.someMethod();
}
}

```

Bounded wildcards (bo tak się tę grupę określa) pozwalają na obejście tego problemu poprzez określenie, że typ generyczny ma być pewnego typu.

Tabela 2. Wildcards podsumowanie

Wildcard	Syntax	Przykład
<i>Unbounded wildcard</i>	?	List<?> l = new ArrayList<>();
<i>Upper-Bounded Wildcard</i>	? extends type	List<? extends Animal> l = new ArrayList<Cat>();
<i>Lower-Bounded Wildcard</i>	? super type	List<? super Animal> l = new ArrayList<Object>();

Znak zapytania w tych zapisach oznacza, nieznany typ generyczny i przechodząc po kolei.

Unbounded wildcard

Unbounded wildcard można rozumieć jako "jakikolwiek typ danych". Czyli pisząc `?` mówimy, "daj mi cokolwiek i OK". Na przykładach:

```
public class Example {  
  
    public static void printCities(List<Object> cities) {  
        for (int i = 0; i < cities.size(); i++) {  
            final Object x = cities.get(i);  
            System.out.println(x);  
        }  
    }  
  
    public static void main(String[] args) {  
        List<String> cities = new ArrayList<>();  
        cities.add("Wrocław");  
        printCities(cities); // Tutaj się nie kompiluje  
    }  
  
}
```

Przykład powyżej się nie kompiluje, bo metoda `printCities()` oczekuje `List<Object>`, a my przekazujemy `List<String>`. Żeby obejść taki problem, możemy zapisać to tak:

```
class Example {  
    public static void printCities(List<?> cities) {  
        for (int i = 0; i < cities.size(); i++) {  
            // Tutaj wtedy należy przypisać wartość do zmiennej Object.  
            // Jeżeli spróbujemy do typu String, dostaniemy błąd kompilacji.  
            Object x = cities.get(i);  
            System.out.println(x);  
        }  
    }  
}
```

Upper-Bounded Wildcards

```
class Animal {}

class Cat extends Animal {}

class Dog extends Animal {}

public class Example {

    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        List<Cat> cats = new ArrayList<>();
        List<Dog> dogs = new ArrayList<>();
        print(animals);
        // Tutaj dostaniemy błąd kompilacji, bo metoda oczekuje listę List<Animal>,
        // a my przekazujemy List<Cat>
        print(cats);
        // Tutaj dostaniemy błąd kompilacji, bo metoda oczekuje listę List<Animal>,
        // a my przekazujemy List<Dog>
        print(dogs);
    }

    public static void print(List<Animal> animals) {
        for (Animal animal : animals) {
            System.out.println(animal);
        }
    }
}
```

W tym momencie, wjeżdża całe na biało Upper-Bounded Wildcards i wystarczy, że metodę print zapiszemy tak:

```
public static void print(List<? extends Animal> animals) {
    for (Animal animal : animals) {
        System.out.println(animal);
    }
}
```

W tym momencie określamy, że pasuje nam każdy typ generyczny w Liście, który dziedziczy z klasy Animal. Tutaj należy tylko pamiętać, że robi się ciekawa kwestia, która wychodzi, jeżeli przykładowo do listy, określonej jako `List<? extends Animal>` będziemy chcieli dodać jakieś elementy (wyobraźmy sobie, że klasy Animal, Cat i Dog są zdefiniowane jak w poprzednim przykładzie):

```
public class Example {

    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        List<Cat> cats = new ArrayList<>();
        List<Dog> dogs = new ArrayList<>();

        // Tutaj jest wszystko w porządku,
        // referencja upperBoundAnimals może wskazywać na listę List<Dog>,
        // to oznacza zapis List<? extends Animal>
    }
}
```



```
// Zwróć też uwagę, że na tym etapie, wiemy też,  
// jaka konkretnie jest zawartość listy dogs i że są tam same psy, nie ma tam kotów.  
List<? extends Animal> upperBoundAnimals = dogs;  
  
// Czegoś takiego zapisać nie możemy,  
// bo nie wiadomo czy finalnie ma to wtedy być lista List<Animal>,  
// czy List<Cat>, czy List<Dog>.  
// Zatem na wszelki wypadek, żaden z poniższych zapisów nie jest dozwolony  
upperBoundAnimals.add(new Dog());  
upperBoundAnimals.add(new Cat());  
upperBoundAnimals.add(new Animal());  
}  
}
```

Lower-Bounded Wildcard

Wyobraźmy sobie teraz taką sytuację:

```
class Animal {}

class Cat extends Animal {}

class Dog extends Animal {}

public class Example {

    public static void main(String[] args) {
        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat());
        List<Animal> animals = new ArrayList<>(cats);

        addCat1(cats);
        addCat1(animals);

        addCat2(cats);
        addCat2(animals);

        addCat3(cats);
        addCat3(animals);
    }

    public static void addCat1(List<?> list) {
        list.add(new Cat());
    }

    public static void addCat2(List<Object> list) {
        list.add(new Cat());
    }

    public static void addCat3(List<? extends Animal> list) {
        list.add(new Cat());
    }
}
```

Próbujemy w metodach addCat1, addCat2, addCat3 dodać Cat do listy i jednocześnie zapisać to w taki sposób, żeby parametr tej metody mógł przyjąć listę Cat i listę Animal jednocześnie. Tutaj uprzedzę, że jest to skrajny przypadek, żeby robić coś takiego w praktyce, ale należy wiedzieć jak podobną sytuację rozwiązać.

I kolejno:

- addCat1 - do listy `List<?>` list nie możemy dodać `new Cat()`, bo dostaniemy błąd kompilacji. Ma sens, w sumie w tej liście może być cokolwiek przez `?`,
- addCat2 - wprawdzie do listy `List<Object>` list możemy dodać `new Cat()`, bo przecież Cat rozszerza klasę Animal, ale do metody `addCat2()` nie możemy przekazać ani listy `List<Cat>`, ani listy `List<Animal>`, bo metoda spodziewa się od nas listy `List<Object>`. Rozwiązaniem byłoby zdefiniowanie w metodzie addCat2 parametru w sposób `List<?> list`, ale wtedy w ciele metody nie moglibyśmy dodać `new Cat()`, jak w przykładzie addCat1,

- addCat3 - jeżeli parametr w metodzie addCat3 określimy jako `List<? extends Animal> list` to możemy do wywołania tej metody przekazać listy `List<Cat>` oraz `List<Animal>`, natomiast dostaniemy błąd kompilacji tak jak wyjaśnione zostało w przykładzie Upper-Bounded Wildcard.

Widać zatem, że żaden z poznanych sposobów nie działa 😊.

I w tym momencie na białym koniu wjeżdża Lower-Bounded Wildcard:

```
public class Example {

    public static void main(String[] args) {
        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat());
        List<Animal> animals = new ArrayList<>(cats);

        addCat(cats);
        addCat(animals);
    }

    public static void addCat(List<? super Cat> list) {
        list.add(new Cat());
    }
}
```

W ten sposób mówimy Javie, że będzie to lista `List<Cat>` albo lista obiektów, które są superklasą klasy `Cat`. Czyli klas, z których `Cat` dziedziczy. Jednocześnie dajemy znać, że jest okej dodać `Cat` do tej listy, bo przecież, będą w tej liście zawsze "przynajmniej" obiekty klasy `Cat`, więc spokojnie można do tej listy dodać kolejnego Kota. Albo obiekty dziedziczące z klasy `Cat` (np. `SuperCat`).

```
class SuperCat extends Cat {}
// gdzieś w kodzie
public static void addCat(List<? super Cat> list) {
    list.add(new Cat());
    list.add(new SuperCat());
    list.add(new Animal()); // Ale tak już nie można, dostaniemy błąd kompilacji
}
```

Nie zadziała wtedy również coś takiego:

```
List<SuperCat> superCats = new ArrayList<>();
addCat(superCats);
```

Nie zadziała dlatego, że zapisem `List<? super Cat>` mówimy, że chcemy tutaj listę z obiektami `Cat` albo z klasami, które są superklasami dla `Cat`. `SuperCat` jest subclassą `Cat`, a nie jego superklasą, dlatego to nie zadziała.

Lower-Bounded Wildcard były dla mnie osobiście najtrudniejsze do przyswojenia, musiałem przez to przejść dobre parę razy. W praktyce używa się ich natomiast bardzo rzadko 😊.

Podsumowanie

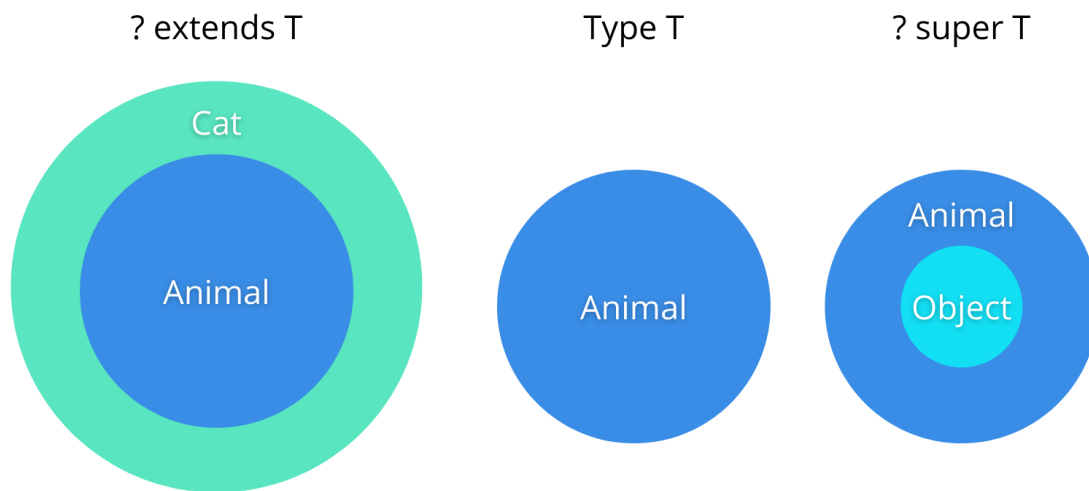
Wiemy już na tym etapie czym są typy generyczne, do czego służą i w jaki sposób mają nam pomóc programować. Powinniśmy już rozumieć, że typ generyczny może być wykorzystany z klasą, interfejsem, a nawet metodą. Na tym etapie powinniśmy mieć również zrozumienie czego z generykami zrobić się nie da. Powinniśmy również rozumieć, że na koniec dnia i tak wszystko jest obiektem ☺.

Dowiedzieliśmy się również, że oprócz "prostego" stosowania generyków, można takie przykłady mocno skomplikować wykorzystując mechanizmy takie jak **Upper-Bounded Wildcards** lub **Lower-Bounded Wildcard**. Jeżeli mielibyśmy problem z zapamiętaniem co i kiedy to z pomocą może przyjść **pecs**.

PECS

Jeżeli będziemy mieli problem z zapamiętaniem, kiedy stosować **upper** bound, a kiedy **lower** bound to pomocne może być przypomnienie sobie zasady **pecs**, której proste wyjaśnienie znajdziemy na [stackoverflow](#). Parafrazując:

Zasada **pecs** oznacza patrzenie z punktu widzenia kolekcji. Jeżeli wyjmujemy generyczne dane z kolekcji - kolekcja działa wtedy jak producent danych i powinniśmy stosować słówko **extends**. Jeżeli natomiast wkładamy dane do kolekcji - kolekcja zachowuj się wtedy jak konsument i stosujemy słówko **super**. Jeżeli robimy obie rzeczy jednocześnie, nie powinniśmy używać ani **extends** ani **super**.



Obraz 1. PECS

Biorąc pod uwagę powyższe wyjaśnienie, możemy teraz łatwo rozwinąć skrót **pecs** - **p**roducer **e**xtends **c**onsumer **s**uper.

Notatki - Comparator i Comparable

Spis treści

W jaki sposób sortować obiekty?	1
Comparator	2
Comparable	3
No to czym one się różnią?	4
Natural Ordering	5
Lexicographical order	5
Natural order	6

W jaki sposób sortować obiekty?

W przypadku liczb, sortowanie jest dosyć naturalne, można rosnąco albo malejąco. W przypadku Stringów, raczej też, możemy najpierw wielkie litery, potem małe litery zgodnie z kolejnością alfabetyczną, a potem np. cyfry. Ale co jeżeli chcemy posortować Koty, Psy, Samochody albo Krzesła?

W tym samym warsztacie poznamy też pewne rodzaje kolekcji, które automatycznie sortują dodane elementy, czyli muszą już wiedzieć w jakiś sposób jak mają posortować kolekcję, jeżeli dodamy do niej np. 4 monitory.

Wtedy z pomocą przychodzą nam 2 mechanizmy, `Comparator` oraz `Comparable`.

Na potrzeby tego wpisu wprowadzę klasę `Cat`, która będzie w tej formie używana później.

```
class Cat {  
  
    private Integer weight;  
  
    public Cat(final Integer weight) {  
        this.weight = weight;  
    }  
  
    public Integer getWeight() {  
        return weight;  
    }  
  
    @Override  
    public String toString() {  
        return "Cat{" +  
            "weight=" + weight +  
            '}';  
    }  
}
```

Comparator

Załóżmy, że mamy Listę kotów `List<Cat>` i chcemy ją w jakiś sposób posortować. Spójrzmy na ten przykład:

```
public class Example {

    public static void main(String[] args) {
        Comparator<Cat> comparator = new Comparator<Cat>() {
            @Override
            public int compare(final Cat o1, final Cat o2) {
                return o1.getWeight() - o2.getWeight();
            }
        };

        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat(2));
        cats.add(new Cat(1));
        cats.add(new Cat(4));
        cats.add(new Cat(3));

        System.out.println("Before: " + cats);
        Collections.sort(cats, comparator);
        System.out.println("After: " + cats);
    }
}
```

Przykład powyżej wykorzystuje mechanizm klasy anonimowej. `Comparator` jest interfejsem funkcyjnym, oznacza to, że można go zaimplementować za pomocą lambdy. (Dokładnie zostanie to wyjaśnione jak dojdziemy do programowania funkcyjnego). Następne przykłady będą zatem implementowane wykorzystując lambdę.

```
public class Example {

    public static void main(String[] args) {
        Comparator<Cat> comparator1 = (o1, o2) -> o1.getWeight() - o2.getWeight();

        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat(2));
        cats.add(new Cat(1));
        cats.add(new Cat(4));
        cats.add(new Cat(3));

        System.out.println("Before: " + cats);
        Collections.sort(cats, comparator1);
        System.out.println("After: " + cats);
    }
}
```

Sortujemy tutaj koty na podstawie ich wagi rosnąco.

Zgodnie z dokumentacją metody `int compare(T o1, T o2)` - metoda ta zwraca integer, który może być odpowiednio ujemny, zerowy lub dodatni. Jeżeli wynik jest:

- ujemny to pierwszy argument jest mniejszy niż drugi,
- zerowy to pierwszy argument jest równy drugiemu,
- dodatni to pierwszy argument jest większy niż drugi.

Biorąc to pod uwagę, jeżeli chcemy posortować elementy rosnąco, metoda `int compare(T o1, T o2)` musi być napisana tak:

```
Comparator<Cat> comparator1 = (o1, o2) -> o1.getWeight() - o2.getWeight();
```

Jeżeli natomiast chcielibyśmy posortować elementy malejąco, metoda `int compare(T o1, T o2)` musi być napisana tak:

```
Comparator<Cat> comparator1 = (o1, o2) -> o2.getWeight() - o1.getWeight();
```

Comparable

Comparable jest natomiast interfejsem, który z założenia ma być implementowany przez klasę, której obiekty chcemy sortować, czyli:

```
class Cat implements Comparable<Cat> {
    private Integer weight;

    public Cat(final Integer weight) {
        this.weight = weight;
    }

    public Integer getWeight() {
        return weight;
    }

    @Override
    public String toString() {
        return "Cat{" +
            "weight=" + weight +
            '}';
    }

    @Override
    public int compareTo(final Cat o) {
        return this.getWeight() - o.getWeight();
    }
}
```

Implementując interfejs `Comparable` musimy zaimplementować metodę `compareTo()`, co do której dokumentacja mówi, że metoda ta zwraca integer, który może być odpowiednio ujemny, zerowy lub dodatni. Jeżeli wynik jest:

- ujemny to pierwszy argument jest mniejszy niż drugi,

- zerowy to pierwszy argument jest równy drugiemu,
- dodatni to pierwszy argument jest większy niż drugi.

Przy czym w tym przypadku porównujemy obiekt, na którym operujemy, poprzez `this`, z obiektem, który przychodzi do nas jako argument metody `compareTo()`.

Biorąc to pod uwagę, jeżeli chcemy posortować elementy rosnąco, metoda `int compareTo(0 o)` musi być napisana tak:

```
@Override
public int compareTo(final Cat o) {
    return this.getWeight() - o.getWeight();
}
```

Jeżeli natomiast chcielibyśmy posortować elementy malejąco, metoda `int compareTo(0 o)` musi być napisana tak:

```
@Override
public int compareTo(final Cat o) {
    return o.getWeight() - this.getWeight();
}
```

W praktyce, możemy wtedy wywołać `Collections.sort(cats);` bez podawania `Comparatora` jako argument wywołania. Tutaj trzeba uważać, bo wywołanie `Collections.sort(cats);` na kolekcji obiektów, które nie implementują interfejsu `Comparable` spowoduje błąd kompilacji.

```
public class Example {

    public static void main(String[] args) {
        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat(2));
        cats.add(new Cat(1));
        cats.add(new Cat(4));
        cats.add(new Cat(3));

        System.out.println("Before: " + cats);
        Collections.sort(cats);
        System.out.println("After: " + cats);
    }
}
```

No to czym one się różnią?

Jeżeli implementujemy `Comparable`, to mamy tylko jeden sposób sortowania obiektów, bo zawsze mamy zaimplementowaną metodą `compareTo(0 o)` w jeden i ten sam sposób. `Comparator` natomiast pozwala nam posortować tę samą kolekcję na kilka różnych sposobów.

Wiedząc już czym jest `Comparator` i czym jest `Comparable` możemy pokusić się o ich porównanie:

Tabela 1. Porównanie `Comparator` vs `Comparable`

Comparator	Comparable
<code>Comparator</code> znajdziemy w paczce <code>java.util</code>	<code>Comparable</code> znajdziemy w paczce <code>java.lang</code>
Stosując <code>Comparator</code> wykorzystamy metodę <code>Collections.sort(list, comparator)</code>	Stosując <code>Comparable</code> wykorzystamy metodę <code>Collections.sort(list)</code>
Stosując <code>Comparator</code> implementujemy metodę <code>compare()</code>	Stosując <code>Comparable</code> implementujemy metodę <code>compareTo()</code>
Stosując <code>Comparator</code> nie modyfikujemy kodu klasy, czyli nie wpływamy na kod źródłowy klasy	Stosując <code>Comparable</code> modyfikujemy kod klasy, czyli wpływamy na kod źródłowy klasy
Stosując <code>Comparator</code> możemy z powodzeniem sortować obiekty klas, których nie napisaliśmy samodzielnie, czyli klasy z bibliotek zewnętrznych (o takich rzeczach będzie potem). Wynika to z tego, że nie możemy modyfikować kodu źródłowego klas z bibliotek zewnętrznych, a <code>Comparator</code> tego nie robi	Stosując <code>Comparable</code> problematyczne okazuje się sortowanie obiektów klas, których nie napisaliśmy samodzielnie, czyli klas z bibliotek zewnętrznych (o takich rzeczach będzie potem). Wynika to z tego, że nie możemy modyfikować kodu źródłowego klas z bibliotek zewnętrznych, a <code>Comparable</code> potrzebuje modyfikować kod klasy do poprawnego działania
Stosując <code>Comparator</code> możemy w prosty sposób określić sekwencję pól, na podstawie których sortujemy, czyli np. <code>name</code> , <code>surname</code> , <code>age</code> itp	Stosując <code>Comparable</code> również możemy to zrobić, natomiast jest to bardziej skomplikowane
Wykorzystując <code>Comparator</code> w jego implementacji nie będziemy stosować <code>Comparable</code>	Wykorzystując <code>Comparable</code> w jego implementacji możemy zastosować <code>Comparator</code>

Natural Ordering

Stosując `Comparator` natkniesz się na taki zapis:

```
Comparator.naturalOrder();
```

Wyjaśnijmy sobie zatem czym jest **natural order**, zanim natomiast to zrobimy, dodajmy jeszcze pojęcie **lexicographical order**.

Lexicographical order

Lexicographical order (porządek leksykograficzny) jest porządkiem alfabetycznym. Żeby lepiej to zrozumieć, przytoczę fragment dokumentacji metody `compareTo()` w klasie `String`.

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string.

Czyli tego rodzaju porównanie będzie się opierało na tablicy **Unicode**. Wartości w tej tablicy będą posortowane zgodnie z kolejnością: cyfry, duże litery, małe litery, duże polskie znaki, małe polskie znaki. Żeby to zobrazować, uruchom poniższy przykład.

```
List<String> list = new ArrayList<>();
list.add("text11");
list.add("text2");
list.add("aa");
list.add("b");
list.add("B");
list.add("Ó");
list.add("ó");
list.add("Ł");
list.add("ł");

System.out.println("Before: " + list);
Collections.sort(list);
System.out.println("After: " + list);
```

Na ekranie zostanie wydrukowane:

```
Before: [text11, text2, aa, b, B, Ó, ó, Ł, ł]
After: [B, aa, b, text11, text2, Ó, ó, Ł, ł]
```

Zgodnie z tym co jesteśmy nauczeni, **2** powinno być przed **11**, więc **text2** powinno być przed **text11**. Zgodnie z kolejnością alfabetyczną **ł** powinno być przed **Ł**, a tutaj widać, że jest za nim. Wszystkie te "dziwne" zachowania wynikają z tego, że domyślne sortowanie **Stringów** w Java odbywa się na podstawie tablicy **Unicode**. Wrócimy do tematu czym jest tablica **Unicode** w warsztacie o operacjach na plikach na dysku komputera. Zgodnie z tą tablicą, przykład **11** i **2** należy traktować analogicznie jak **b** i **aa**, gdzie **2** jest jak **b**, a **11** jest jak **aa**.

Natural order

Zostało wprowadzone również takie pojęcie jak **natural order** [Wikipedia](#). Taka kolejność sortowania powinna być bardziej zrozumiała dla człowieka, czyli z tym podejściem do sortowania powinniśmy zobaczyć kolejność **text2**, a później **text11**.

Java jest tutaj myląca. Jeżeli w przykładzie powyżej, użyjemy do sortowania **Comparator.naturalOrder()**, czyli przepiszemy sortowanie w ten sposób:

```
Collections.sort(list, Comparator.naturalOrder());
```

to na ekranie zostanie wydrukowane:

```
Before: [text11, text2, aa, b, B, Ó, ó, Ł, ł]
After: [B, aa, b, text11, text2, Ó, ó, Ł, ł]
```

Czyli to samo prawda? A przecież **natural order** miał załatwić sprawę. I tutaj właśnie jest ta rozbieżność. Pojęcie **natural order** oznacza sortowanie bardziej naturalne dla człowieka. Natomiast metoda **Comparator.naturalOrder()** pod spodem woła metodę **compareTo()** z klasy, której obiekty mają być sortowane. Jak to działa dla klasy **String** zostało wspomniane wcześniej. Stąd otrzymujemy taki wynik.

Jeżeli ktoś ma ochotę dalej pogrzebać to [tutaj](#) umieszczam link do **Stackoverflow**.

Gdybyśmy chcieli porównać ze sobą sortowanie **alfabetyczne** z **natural order** - wyglądałoby to w ten sposób:

*Alphabetical**VS**Natural**Text 1**Text 1**Text 10**Text 2**Text 11**Text 10**Text 2**Text 11**Text 20**Text 20**Text 21**Text 21*

Notatki - Kolekcje

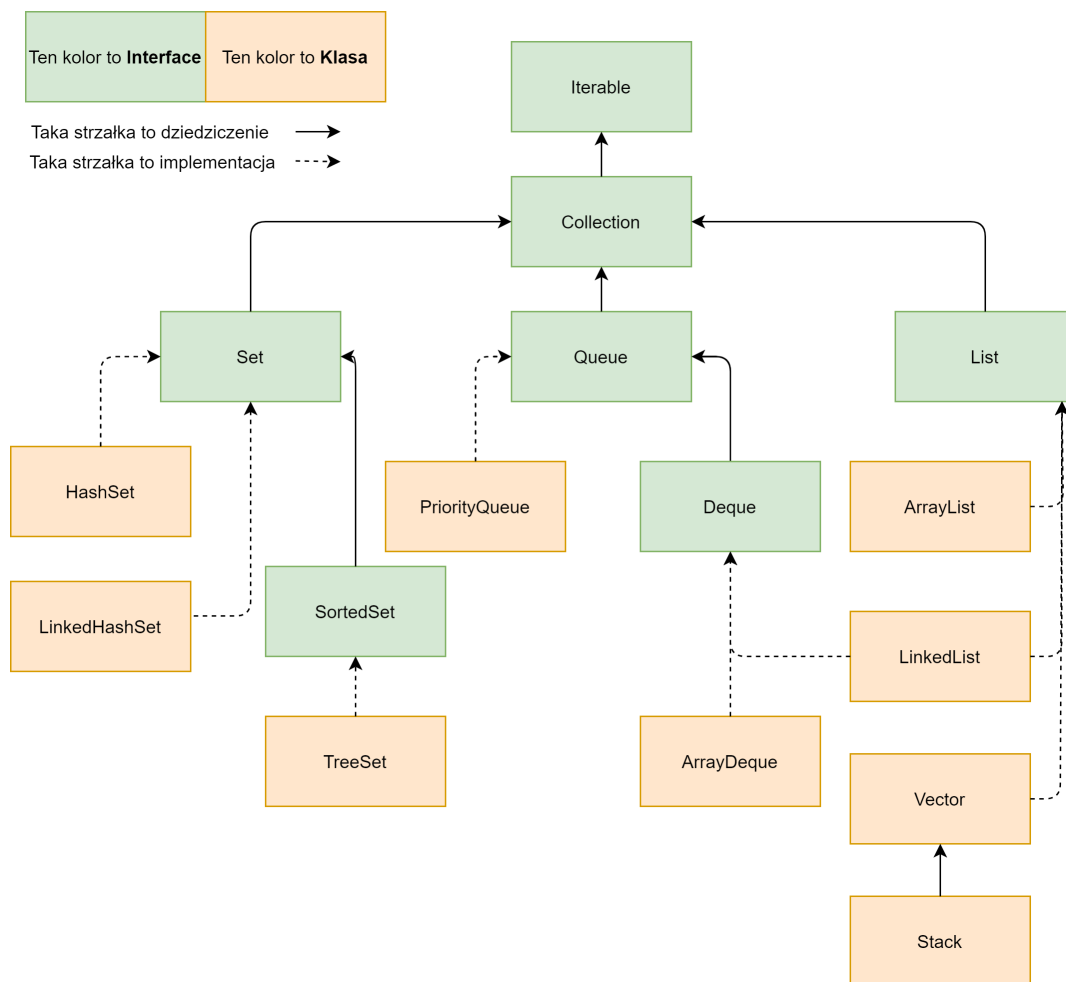
Spis treści

Kolekcje	1
List	2
ArrayList	2
LinkedList	4
Vector	6
Stack	6
Set	6
Struktura HashTable	7
HashSet	8
TreeSet	9
LinkedHashSet	11
Queue i Deque	12
LinkedList	12
ArrayDeque	13
PriorityQueue	15
Map	17
Hashtable	18
HashMap	18
TreeMap	20
LinkedHashMap	20
Podsumowanie	20
Notacja Big O	21
Na co to komu?	21
Przypadki	22

Kolekcje

Kolekcje to określenie na grupę obiektów, które możemy przechowywać w pojedynczym obiekcie (kolekcji). **Java Collections**, to zbiór klas w paczce `java.util`, są tam zdefiniowane m.in. 4 główne interfejsy: `List`, `Set`, `Map` i `Queue`.

W taki sposób wygląda diagram hierarchii klas i interfejsów, które dziedziczą z interfejsu `Collection`, o których będziemy rozmawiać.



Obraz 1. Hierarchia Collections

Tak jak wspomniałem, **List**, **Set**, **Map** i **Queue** to są interfejsy, każdy z nich ma swoje konkretne implementacje, które różnią się pewnymi niuansami. Dlatego przejdźmy po kolei przez 4 omówione w materiałach filmowych rodzaje kolekcji i ich implementacje.

List

Lista to kolekcja elementów zachowująca kolejność. Pozwala na przechowywanie duplikatów. Możliwy jest dostęp do elementów po podanym indeksie, który oznacza miejsce, na którym element się znajduje w kolekcji.

Listy używamy, gdy chcemy przechowywać elementy w określonej kolejności z możliwością posiadania zduplikowanych elementów. Elementy mogą być pobierane i wstawiane w określonych pozycjach na liście w oparciu o indeks - jak w tablicach. Listy są bardzo często używaną kolekcją, gdyż wiele razy przychodzi nam przechowywanie więcej niż jednego elementu w zbiorze. Myślę, że nie będzie to kłamstwo jak napiszę, że jest to jedna z częściej stosowanych kolekcji. Najważniejszą cechą wspólną dla wszystkich implementacji **List** jest to, że mają uporządkowaną kolejność i zezwalają na duplikaty. Poza tym każda implementacja różni się pewnymi cechami.

ArrayList

ArrayList jest jak tablica z automatyczną możliwością zmieniania rozmiaru w trakcie działania. **ArrayList** ma de facto pod spodem zaimplementowaną tablicę, którą automatycznie zarządza. Po

dodaniu elementów do `ArrayList` jest ona w stanie automatycznie zwiększać swój rozmiar.

Główną zaletą `ArrayList` jest to, że możliwe jest wyszukanie dowolnego elementu w stałym czasie, gdyż pod spodem mamy tablicę elementów, która daje nam możliwość dostępu do tych elementów w stałym czasie. Dodawanie lub usuwanie elementu jest natomiast wolniejsze niż uzyskiwanie dostępu do elementu, szczególnie gdy chcemy taki element dodać w środku listy, np. na pozycji 3, wtedy wszystkie elementy za pozycją 3 muszą zostać przesunięte o 1. Tablica pod spodem musi wtedy wymienić indeksy pozostałych elementów, co komplikuje nam taką operację i zwiększa czas.

W najgorszym przypadku, przy dodawaniu elementu do listy, jeżeli skończy nam się miejsce w tablicy pod spodem, `ArrayList` musi stworzyć nową tablicę i przepisać wszystkie elementy ze starej do nowej. Stąd też jeżeli będziemy potrzebowali dodawać bardzo dużo elementów do listy, `ArrayList` może nie być najlepszym wyborem. Jednocześnie `ArrayList` jest dobrym wyborem, gdy więcej czytamy z listy niż do niej zapisujemy (albo mamy tyle samo zapisów co odczytów). Taka operacja została rozrysowana na obrazku poniżej.

Różnica między `ArrayList` a tablicą jest taka, że tablica ma raz określony rozmiar i jak chcemy do niej (tablicy) dodać więcej elementów, niż tablica jest w stanie zmieścić, to musimy zdefiniować nową tablicę z nowym rozmiarem i przepisać do niej elementy ze starej tablicy. `ArrayList` robi to za nas, dlatego też w praktyce tablic używa się bardzo rzadko, najczęściej stosowane są kolekcje.

Często się mówi, że jeśli nie masz pewności, której kolekcji użyć, używaj `ArrayList`.

Implementacja ArrayList

Tworzymy ArrayList, domyślny rozmiar tablicy pod spodem wynosi 10:

```
List<Integer> list = new ArrayList<>();
```

0	1	2	3	4	5	6	7	8	9

Dodajemy pierwszy element:

```
list.add(12);
```

12									
0	1	2	3	4	5	6	7	8	9

Dodajemy resztę elementów:

```
list.add(7); list.add(1); list.add(2); list.add(1); ...
```

12	7	1	2	1	12	8	2	9	11
0	1	2	3	4	5	6	7	8	9

Jeżeli teraz spróbujemy dodać kolejny element, zostanie stworzona nowa tablica, większa np. o 50%, do której przepisujemy elementy ze starej tablicy

12	7	1	2	1	12	8	2	9	11
0	1	2	3	4	5	6	7	8	9

Do nowej tablicy został dodany nowy element, ten na żółto

12	7	1	2	1	12	8	2	9	11	14	...	
0	1	2	3	4	5	6	7	8	9	10	11 ... 14	

Obraz 2. ArrayList działanie

Na obrazku zostało wspomniane, że nowo-utworzona tablica może wzrosnąć np. o 50%. Dlatego napisałem np., bo dokumentacja **ArrayList** nie określa jak bardzo ta tablica rośnie gdy wystąpi taka potrzeba, ale często w internecie można przeczytać, że 50%.

LinkedList

LinkedList jest o tyle specyficzną implementacją, że implementuje jednocześnie interfejsy **List** i **Queue**. Skoro implementuje interfejs **List**, oznacza to, że ma dostępne wszystkie metody **List**. Posiada również dodatkowe metody ułatwiające dodawanie lub usuwanie z początku i / lub końca listy, gdyż implementuje interfejs **Queue**.

Główne zalety **LinkedList** to możliwość uzyskiwania dostępu, dodawania i usuwania plików z początku i końca listy w stałym czasie, wynika to z tego w jaki sposób **LinkedList** jest zaimplementowana pod spodem. Jednocześnie kompromis polega na tym, że znalezienie elementu na dowolnym indeksie wymaga czasu liniowego, czyli im dalej jest element od początku lub końca listy tym dłużej go szukamy.

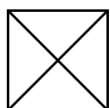
Efekt ten staje się coraz bardziej widoczny im więcej elementów przechowujemy w liście. Spójrz na obrazek poniżej aby zobaczyć jak elementy w liście są ze sobą połączone, wtedy się wyjaśni, czemu aby dostać się do elementu np. na pozycji 3, trzeba zacząć, od któregoś końca i iść przed siebie.

W przypadku `ArrayList` czas odczytu elementu był stały, gdyż tablica pod spodem wiedziała gdzie w pamięci jest każdy z elementów, który był zapisany pod danym indeksem. W przypadku `LinkedList`, szukając elementu na pozycji np. 7, musimy zacząć albo od początku, albo od końca listy i przechodzić przez każdy z elementów na liście, gdyż każdy element wie co następuje przed nim i co nastąpi po nim. W przypadku `LinkedList` nie da się przejść bezpośrednio do elementu na pozycji 7. Musimy zrobić taki sznureczek od początku albo od końca, odpytując każdy z elementów, kto jest następny, aż znajdziemy element, którego szukamy. To zajmuje czas i trwa tym dłużej im więcej elementów na tej liście się znajduje. To wszystko sprawia, że `LinkedList` jest często dobrym wyborem, jeżeli używamy jej jak kolejki, czyli dodajemy i usuwamy elementy na początku lub na końcu. Natomiast nie jest dobrym wyborem, jak mamy dużo elementów w liście i chcemy dostać się do elementu na pozycji np 38.

Implementacja LinkedList

Tworzymy `LinkedList`, pod spodem jest ona pusta:

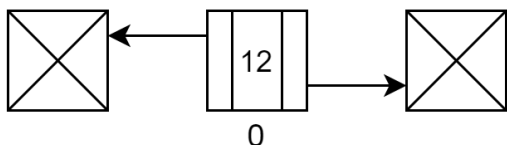
```
List<Integer> list = new LinkedList<>();
```



Przyjmijmy, że to oznacza pustkę ;)

Następnie dodajemy do tej listy element:

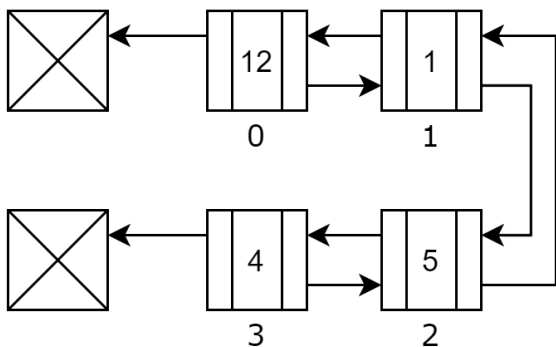
```
list.add(12);
```



Lista ma w tym momencie tylko jeden element, który jest na raz początkiem i końcem listy

Dodajemy do listy kolejne elementy

```
list.add(1); list.add(5); list.add(4);
```



Każdy kolejny element (node) listy, oprócz tego, że przechowuje swoją wartość (czyli 1 albo 12), przechowuje też referencję do node'a, który jest przed nim i node'a, który jest za nim.

Obraz 3. `LinkedList` działanie

To co widać na obrazku powyżej to podwójnie linkowana (*doubly linked*) Lista. Taka implementacja jest stosowana w Javie, czyli każdy element może wskazać namier na element następny i element poprzedzający. Istnieją również implementacje pojedynczo linkowanych (*singly linked*) list, wtedy możemy tylko zacząć od początku listy i iść przed siebie aż do końca. `LinkedList` jest *doubly linked*.

Przykład wykorzystania `ArrayList` i `LinkedList` różni się tylko konkretną klasą implementującą interface

List, dlatego przykłady pokażę na `ArrayList`.

```
List<String> listWithCities = new ArrayList<>();
listWithCities.add("Warszawa");
listWithCities.add("Gdańsk");
listWithCities.add("Łódź");
listWithCities.add("Wrocław");

// Tablica wymagała użycia metody Arrays.toString(),
// lista drukuje się normalnie bez tego
// Cities: [Warszawa, Gdańsk, Łódź, Wrocław]
System.out.println("Cities: " + listWithCities);

// IndexOutOfBoundsException
System.out.println("Get 120: " + listWithCities.get(120));
// Get 0: Warszawa
System.out.println("Get 0: " + listWithCities.get(0));
// isEmpty: false
System.out.println("isEmpty: " + listWithCities.isEmpty());
// size: 4
System.out.println("size: " + listWithCities.size());
// contains: false
System.out.println("contains: " + listWithCities.contains("zajavka"));
```

Istnieją również dwie stare implementacje interfejsu `List`.

Vector

Dawno temu jak chcieliśmy mieć zachowanie analogiczne do `List`, używało się klasy `Vector`. W Javie 1.2 zostało wprowadzone `ArrayList` i od tego momentu należy tej klasy używać zamiast `Vector`. `Vector` robi to samo, co `ArrayList`, tylko wolniej, ze względu na sposób w jaki został zaimplementowany. Implementacja ta powoduje, że `Vector` jest bezpieczniejszy w przypadku programowania wielowątkowego, ale my cały czas rozmawiamy o aplikacjach jednowątkowych, dlatego nie zagłębiam się w ten temat. Pisząc jednowątkowych mam na myśli, że programy, które piszemy wykonują się w jednym wątku. Nic nie dzieje się równolegle, do tego przejdziemy później. Jedyny powód, dla którego musisz wiedzieć o istnieniu klasy `Vector`, jest to, że może odnosić się do niej naprawdę stary kod.

Stack

Stack to struktura danych, w której dodajesz i usuwasz elementy ze szczytu stosu. Stos możemy sobie wyobrazić jako stos faktur na stole. Kolejna faktura jest kładziona na górze stosu, pierwszą fakturą jaką zdejmujemy jest ostatnia jaka została na tym stosie położona. Podobnie jak klasa `Vector`, klasa `Stack` nie jest już dawno używany do tworzenia nowego kodu. W rzeczywistości klasa `Stack` rozszerza klasę `Vector`. Jeśli potrzebujesz zachowania stosu, zamiast klasy `Stack` należy używać klasy `ArrayDeque`.

Nie należy mylić tej klasy ze strukturą Stosu, którą omawialiśmy w modelu pamięci Javy. Tutaj mówimy o klasie, dzięki której możemy zaimplementować zachowanie stosu.

Set

Kolekcja, która nie pozwala na przechowywanie duplikatów i jednocześnie nie daje nam możliwości posiadania indeksów, czyli oznaczania miejsc w których znajdują się elementy. Nie możemy zatem w

przypadku interfejsu `Set` odwołać się do elementu na konkretnej pozycji, elementy mogą być w zbiorze w losowej kolejności. Wiemy natomiast, że dodając te elementy do zbioru nie będziemy mogli w nim mieć elementów zduplikowanych.

I znowu, każda konkretna implementacja oferuje inną funkcjonalność. Najczęściej używane to `HashSet`, `TreeSet` i `LinkedHashSet`.

Zanim natomiast przejdziemy do implementacji `HashSet` należy wspomnieć o strukturze `HashTable`, bo implementacja `HashSet` go wykorzystuje.

Struktura HashTable

`HashTable` to struktura danych, która będzie nam potrzebna do omówienia klas `HashSet` i `HashMap`.

Przechodzimy do praktycznego wykorzystania metody `hashCode()`. Lubię tłumaczyć tę strukturę na drużynach piłkarskich i tak też zrobimy 😊. Działanie metody `hashCode()` wyjaśnialiśmy w sekcji o programowaniu obiektowym.

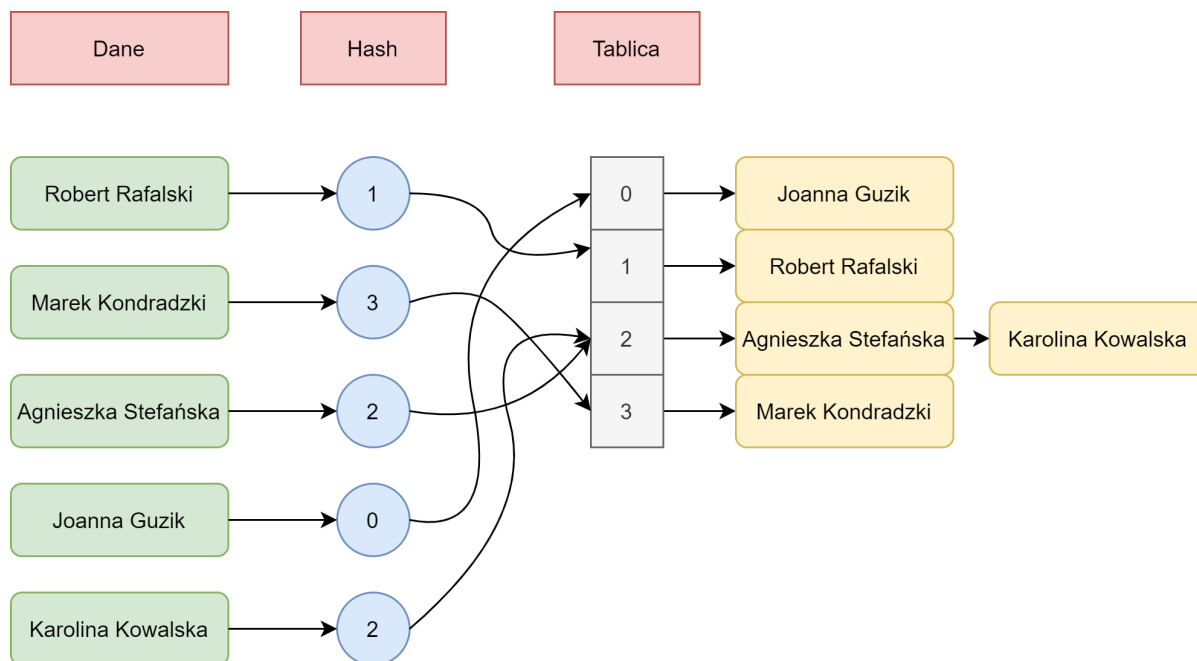
Wyobraź sobie, że masz przed sobą 1000 osób i potrzebujesz wśród nich znaleźć Karolinę Kowalską (patrz obrazek niżej). Ja wiem, że w życiu codziennym można krzyknąć "Hej, szukam Karoliny Kowalskiej" i mogłaby ona podnieść rękę, ale programując to tak nie działa.

Gdybyśmy chcieli w programowaniu znaleźć Panią Karolinę, trzeba by było zapytać każdą z 1000 osób po kolei, czy przypadkiem nie nazywa się "Karolina Kowalska", czyli zrobić "`Karolina Kowalska`".`equals(osoba.getName())` i albo byśmy trafili od razu, albo znaleźli Panią Karolinę po przepytaniu 998 osób.

Żeby zrobić to szybciej, można by najpierw podzielić wszystkie te 1000 osób na 10 drużyn piłkarskich, na podstawie np. metody `hashCode()`. Czyli liczymy `hashCode()` dla każdej osoby, jak wychodzi 3 to trafia do drużyny 3, jak 9 to do 9. Później napiszę co można zrobić jak `hashCode()` zwróci wynik większy niż mamy ilość drużyn.

Skoro każda z osób ma już policzoną wartość `hashCode()` i na tej podstawie trafiła do jakiejś drużyny, a wiemy też, że szukamy "Karolina Kowalska", to możemy policzyć `hashCode()` z "Karolina Kowalska" i na tej podstawie będziemy wiedzieli, w której drużynie tej Pani szukać. Jeżeli mamy napisaną metodę `hashCode()` w taki sposób, że każda drużyna ma mniej więcej tyle samo osób, to z przeszukiwania 1000 osób, zmniejszyliśmy właśnie ilość porównań `"Karolina Kowalska".equals(osoba.getName())` do 100.

Takie właśnie jest założenie struktur, które wykorzystują `HashTable`, stworzyć tablicę (tablica ta często jest nazywana *hash table*) z jakąś cechą grupującą (w tym przypadku wartość `hashCode()`) i na tej podstawie przyspieszyć wyszukiwanie, bo wiemy w której grupie możemy szukać.



Obraz 4. Hashtable działanie

Na obrazku, dla każdej z osób, możemy policzyć `hashCode()` i na tej podstawie umiejscowić tę osobę w konkretnej grupie. Grupa ma konkretne miejsce w tablicy. Jeżeli w danej grupie pojawia się więcej niż jedna osoba, możemy je dodawać do Listy, którą poznaliśmy wcześniej. Sytuacja taka nazywa się w praktyce *hash collision* - kolizją hashy. Dla kilku osób wyliczyliśmy tę samą wartość `hashCode()` i muszą one trafić do tego samego indeksu w *hash table*, zatem naturalnym jest dodanie pod takim indeksem listy, aby pod tym samym indeksem mogło znaleźć się kilka osób.

I znowu, jeżeli szukalibyśmy takiej osoby "Karolina Kowalska", należałoby policzyć dla kogoś takiego `hashCode()`, znaleźć na tej podstawie odpowiednie miejsce w tablicy *hash table*, zostanie nam wtedy wykonanie `"Karolina Kowalska".equals(osoba.getName())` dla ludzi, którzy znajdują się na liście osób pod danym indeksem tablicy *hash table*, nie będziemy tego musieli robić dla wszystkich ludzi znajdujących się w zbiorze.

Co zrobić jeżeli `hashCode()` wyjdzie większy niż rozmiar tablicy? W praktyce stosuje się modulo. Czyli `hashCode() % rozmiarTablicy`, wtedy otrzymamy wartości w zakresie `0 - rozmiarTablicy` i problem załatwiony.

Dodam tutaj też, że często pojawiają się stwierdzenia o dobrej metodzie `hashCode()`. Patrząc na przykład powyżej, dobra metoda `hashCode()` to taka, która zapewnia bardzo równomierne rozłożenie danych w konkretnych grupach (miejscach w tablicy). Innymi słowy, najgorszy `hashCode()`, to taki, który dla każdego obiektu zwróciłby 1 (a w zasadzie to każdą liczbę, która byłaby stała), bo wtedy nie osiągnęlibyśmy żadnego grupowania, wszyscy byłiby w tej samej grupie.

HashSet

`HashSet` przechowuje swoje elementy zgodnie ze strukturą `HashTable`. Oznacza to, że używa metody `hashCode()` dla przetrzymywanych obiektów, aby wydajniej je pobierać. Należy pamiętać o tym, że przez to, że stosujemy tę strukturę, tracimy informację o kolejności elementów w jakiej były one dodawane. W większości przypadków używania `Set` i tak okazuje się, że nie przyjmujemy określonej kolejności elementów, co sprawia, że `HashSet` jest najpopularniejszą implementacją.

```
Set<String> setWithCities = new HashSet<>();
setWithCities.add("Warszawa");
setWithCities.add("Gdańsk");
setWithCities.add("Łódź");
setWithCities.add("Wrocław");
setWithCities.add("Warszawa");
setWithCities.add("Wrocław");

// Tablica wymagała użycia metody Arrays.toString(),
// set drukuje się normalnie bez tego
// Cities size: [Gdańsk, Warszawa, Łódź, Wrocław]
// Set pozbywa się duplikatów, dlatego mamy tylko unikalne miasta
System.out.println("Cities size: " + setWithCities);

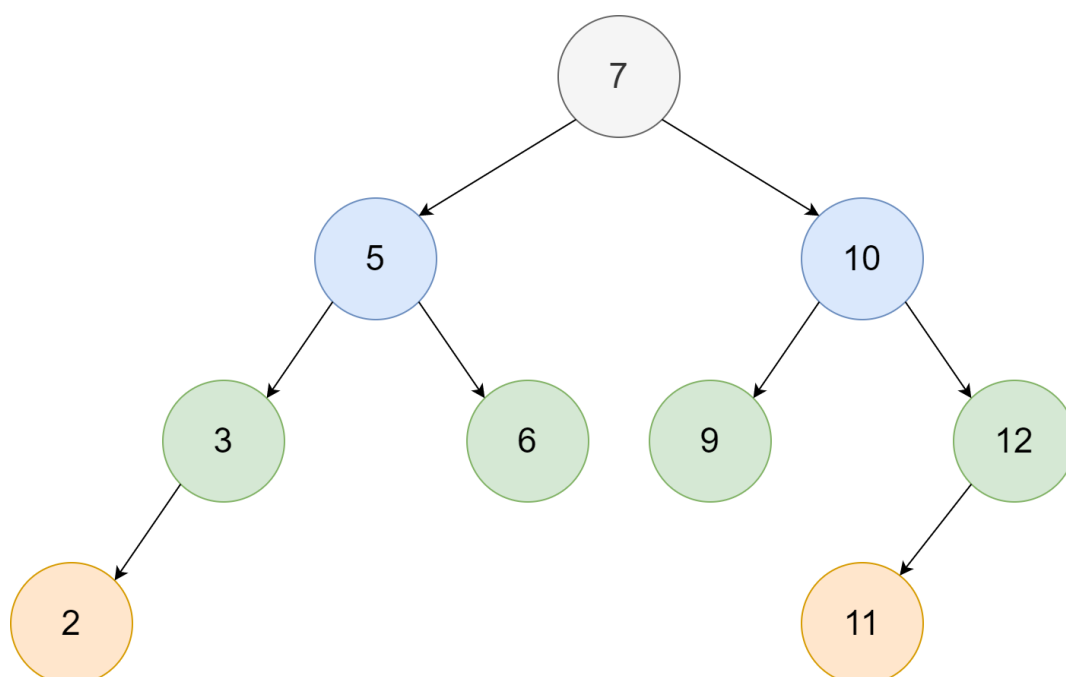
// nie kompiluje się gdyż Set nie ma metody get
// System.out.println("Get 120: " + setWithCities.get(120));

// isEmpty: false
System.out.println("isEmpty: " + setWithCities.isEmpty());
// size: 4
System.out.println("size: " + setWithCities.size());
// contains: false
System.out.println("contains: " + setWithCities.contains("zajavka"));
```

TreeSet

Czym jest drzewo

Zanim przejdziemy do omówienia **TreeSet** (czyli jakiś zbiór z drzewem), omówmy sobie czym jest drzewo. Jest to struktura, która w naturalny sposób jest w stanie reprezentować hierarchię danych i często jest stosowane do tego celu. Drzewa są stosowane gdy potrzebujemy ułatwić wyszukiwanie danych, ze względu na to, że możemy w prosty sposób przedstawić zależność między danymi jeżeli są one posortowane.



Obraz 5. Przykład drzewa

W opisie poniżej będziemy odnosić się do rysunku powyżej. Drzewa składają się z **wierzchołków (węzłów)** oraz **krawędzi** łączących te wierzchołki. Wyróżniamy również określenie takie jak **korzeń**, które oznacza najwyższy wierzchołek (na rys. wartość 7). Jeżeli zaczynamy łączyć ze sobą węzły za pomocą krawędzi, to ciąg takich krawędzi nazywa się **ścieżką**. Drzewo może mieć tylko jedną ścieżkę, która doprowadzi nas z korzenia do każdego wierzchołka. Możemy wyróżnić również **poziom węzła**, który można określić zliczając ilość krawędzi od korzenia do danego węzła. Drzewo może mieć również **wysokość**, która określa największy istniejący poziom. Czyli na rysunku, wartość 7 jest na poziomie 0, wartości 5 i 10 są na poziomie 1, wartości 3, 6, 9 i 12 są na poziomie 2, a wartości 2 i 11 są na poziomie 3. Wysokość drzewa to 3. Wyróżniamy też określenie takie jak **dzieci**, czyli węzły połączone z danym wierzchołkiem. Dziećmi wierzchołka z wartością 5 są wartości 3 i 6. Wierzchołek może mieć dowolną liczbę dzieci, a jeżeli nie ma ich wcale to jest nazywany **liściem**. Czyli liście w przykładzie to 2, 6, 9, 11. Wierzchołek jest też rodzicem, jeżeli ma dzieci. Wtedy jest rodzicem swojego dziecka. Każdy węzeł ma jednego rodzica, oprócz korzenia, który nie ma rodziców.

Rodzajów drzew jest dużo, natomiast na ten moment wyjaśnijmy sobie takie drzewo jak **BST (Binary Search Tree)**.

Czym jest drzewo binarne (binary tree)

Drzewo binarne to specjalny rodzaj drzewa, w którym liczba dzieci każdego węzła wynosi nie więcej niż 2. Stąd binarne. Czyli mamy maksymalnie prawe dziecko i lewe dziecko. Mówi się wtedy o prawym synu i lewym synu.

Czym jest BST

BST (*Binary Search Tree*) jest specjalnym rodzajem drzewa, w którym każde lewe poddrzewo danego węzła zawiera elementy o wartościach mniejszych niż wartość w danym węźle, a prawe poddrzewo danego węzła zawiera elementy o wartościach nie mniejszych niż wartość w węźle. Wartości, o których wspominam często są kluczami (np. jeżeli będziemy rozmawiali o mapie, wtedy klucze w lewym poddrzewie będą mniejsze niż klucz w węźle, a klucze w prawym poddrzewie będą nie mniejsze). W węzłach oprócz klucza jest też przetrzymywana informacja o rodzicu i o dzieciach.

Po co stosuje się takie struktury? Żeby było szybciej. Wyobraź sobie teraz, że mając drzewo jak w przykładzie na obrazku chcesz znaleźć w takim drzewie wartość 11. Nie musisz przeszukiwać wszystkich wartości w zbiorze, wystarczy, że wiesz, że zbiór jest ułożony w BST. Wtedy zaczynasz od góry i patrzysz, że $11 > 7$, więc idziesz do prawego poddrzewa (bo pamiętasz jaka jest struktura BST). Następnie sprawdzasz, że $11 > 10$, więc znowu idziesz do prawego poddrzewa. Potem sprawdzasz, że $11 < 12$, więc idziesz do lewego poddrzewa i znajdujesz szukaną wartość. Dzięki temu nie musieliśmy porównywać wszystkich elementów w zbiorze, dokonaliśmy o wiele mniej operacji, niż gdybyśmy musieli przejść przez każdy element w zbiorze w pętli i dla każdego wołać metodę `equals()`.

Wracając do wyjaśnienia czym jest TreeSet

TreeSet przechowuje swoje elementy w posortowanej strukturze drzewa. Główną zaletą jest to, że ta implementacja `Set` jest zawsze posortowana. `TreeSet` implementuje specjalny interfejs o nazwie `NavigableSet`, który umożliwia szukanie elementów w secie szukając elementu większego/mniejszego niż inny element. Jeżeli chcemy korzystać z `TreeSet` z obiektami klasy napisanej przez nas, należy pamiętać, żeby albo określić `Comparator`, którego chcemy używać, albo żeby nasza klasa, której obiekty będziemy przechowywać w `TreeSet` implementowała interfejs `Comparable`. Inaczej dostaniemy błąd w trakcie działania programu.

```

class Cat {

    private String name;

    public Cat(final String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Cat: " + name;
    }
}

// Gdzieś w kodzie
// Klasa Cat nie implementuje interfejsu Comparable, więc musimy podać
// Comparator na etapie tworzenia kolekcji,
// inaczej dostaniemy błąd w trakcie działania programu
Set<Cat> catSet = new TreeSet<>((o1, o2) -> o1.name.compareTo(o2.name));
catSet.add(new Cat("cat3"));
catSet.add(new Cat("cat2"));
catSet.add(new Cat("cat4"));
catSet.add(new Cat("cat1"));
catSet.add(new Cat("cat2"));
// [Cat: cat1, Cat: cat2, Cat: cat3, Cat: cat4]
System.out.println(catSet);

```

LinkedHashSet

LinkedHashSet używamy gdy zachodzi potrzeba utrzymania kolejności iteracji. Czyli jednak jest możliwość eliminowania duplikatów i jednocześnie zachowania kolejności dodawania elementów 😊. Iterując po **HashSet**, kolejność elementów jest nieprzewidywalna, podczas gdy **LinkedHashSet** pozwala nam iterować po elementach w kolejności, w jakiej zostały wstawione. **LinkedHashSet** jest zaimplementowany pod spodem poprzez podwójnie linkowaną **LinkedList**, dzięki której możemy zapamiętać kolejność, w jakiej elementy zostały wstawione. Tutaj należy zwrócić uwagę, że możemy iterować po elementach w kolejności ich wstawiania, ale nie mamy indeksów, na których zostały te elementy wstawione, tak jak w przypadku listy. Możemy natomiast stworzyć listę, przekazując do jej konstruktora nasz set, wtedy dostaniemy indeksy. Ponownie należy zwrócić uwagę na to, że z racji, że ta implementacja ma *hash* w nazwie, to w przypadku przetrzymywania w kolekcji naszych własnych klas/obiektów, należy im zaimplementować metody `equals()` oraz `hashCode()`.

```

class Dog {

    private String name;

    public Dog(final String name) {
        this.name = name;
    }

    @Override
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        final Dog dog = (Dog) o;
        return Objects.equals(name, dog.name);
    }
}

```

```

    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }

    @Override
    public String toString() {
        return "Dog: " + name;
    }
}

// Gdzieś w kodzie
Set<Dog> dogs = new LinkedHashSet<>();
dogs.add(new Dog("doggo3"));
dogs.add(new Dog("doggo2"));
dogs.add(new Dog("doggo1"));
dogs.add(new Dog("doggo3"));
dogs.add(new Dog("doggo2"));
dogs.add(new Dog("doggo4"));

// [Dog: doggo3, Dog: doggo2, Dog: doggo1, Dog: doggo4]
System.out.println(dogs);

```

Natomiast indeksów nie ma, nawet jak użyjemy referencji typu `LinkedHashSet`:

```

LinkedHashSet<Dog> dogs = new LinkedHashSet<>();
dogs.get(0); // błąd kompilacji, bo nie ma takiej metody

```

Queue i Deque

Kolekcja, która porządkuje swoje elementy w określonej kolejności do przetworzenia. Na przykład, gdy chcesz kupić bilet, a ktoś czeka w kolejce, ustawiasz się za tą osobą. Typowa kolejka przetwarza swoje elementy w kolejności first-in-first-out (pierwszy przyszedł, pierwszy wyszedł). Niektóre implementacje kolejki zmieniają to, aby używać innej kolejności. Do tego należy dodać, że jeżeli normalnie elementy są dodawane na końcu kolejki, to możliwe jest też dodanie elementu na początku, coś jak przepuszczenie kobiety w ciąży na początek kolejki.

Poznaliśmy następujące metody implementacji kolejek:

LinkedList

`LinkedList`, pojawiła się o niej wzmianka już wcześniej, oprócz tego, że jest to lista, jest to też podwójna kolejka. Wcześniej wspominaliśmy, że `LinkedList` w Javie jest zaimplementowana jako doubly linked (podwójnie linkowana) lista. Kolejka z podwójnym zakończeniem różni się od zwykłej kolejki tym, że można wstawiać i usuwać elementy zarówno z przodu, jak i z tyłu kolejki. Główną zaletą `LinkedList` jest to, że implementuje zarówno interfejs `List`, jak i `Queue`. Wadą jest to, że nie jest tak wydajna jak "czysta" kolejka. Innym popularnym formatem jest **LIFO** (ostatni na wejściu, pierwszy na wyjściu).

ArrayDeque

ArrayDeque to "czysta" kolejka dwustronna. Przechowuje swoje elementy w tablicy o zmiennym rozmiarze (znowu jakaś klasa ma `Array` w nazwie i pod spodem jest realizowana przez tablicę). Główną zaletą `ArrayDeque` jest to, że jest bardziej wydajna niż `LinkedList`. Filozofia co do wymowy jest ciekawa, bo jedni mówią **ArrayDeck** a inni **ArrayDekju**. Ja na żywo słyszałem chyba tylko deck, ale mój mózg podpowiada mi d-queue.

`LinkedList` i `ArrayDeque` obydwie implementują interface `Deque`, który natomiast dziedziczy z interfejsu `Queue`. Dlatego o tym mówię, gdyż ostatnia poruszana kolejna `PriorityQueue` implementuje interface `Queue` bezpośrednio.

Do `ArrayDeque` możemy również podchodzić jak do stosu, w zależności od tego, jakich metod będziemy używać. I to właśnie `ArrayDeque` jest rekomendowana do użycia jeżeli potrzebujemy osiągnąć zachowanie Stosu (nie stosujemy starej klasy `Stack`). `ArrayDeque` możemy używać jako:

- **stos** - osiągniemy zachowanie **LIFO** (*last-in-first-out*), jeżeli będziemy używać następujących metod: `push/pop/peek`,
- **kolejkę** - osiągniemy zachowanie **FIFO** (*first-in-first-out*), jeżeli będziemy używać następujących metod: `offer/poll/peek`.

Oczywiście jedno i drugie możemy w praktyce ze sobą łączyć i osiągniemy wtedy kolejkę, która od czasu do czasu jest stosem. Znowu, wszystkiego należy używać z głową.

Parę przydatnych metod z interfejsu `Deque`:

metoda	opis	używamy przy kolejce?	używamy przy stosie?
<code>boolean add(E e)</code>	Dodaje element na końcu kolejki i zwraca true albo rzuca wyjątek (np. bo jest za mało miejsca)	Tak	Nie
<code>E element()</code>	Zwraca następny element lub rzuca wyjątek jeżeli kolejka jest pusta	Tak	Nie
<code>boolean offer(E e)</code>	Dodaje element na końcu kolejki i zwraca true jeżeli się udało	Tak	Nie
<code>E remove()</code>	Usuwa i zwraca następny element lub rzuca wyjątek jeżeli kolejka jest pusta	Tak	Nie
<code>void push(E e)</code>	Dodaje element na początku kolejki	Tak	Tak
<code>E poll()</code>	Usuwa i zwraca następny element lub zwraca null jeżeli kolejka jest pusta	Tak	Nie
<code>E peek()</code>	Zwraca następny element lub zwraca null jeżeli kolejka jest pusta	Tak	Tak
<code>E pop()</code>	Zwraca następny element lub rzuca wyjątek jeżeli kolejka jest pusta	Nie	Tak

```
List<Integer> init = Arrays.asList(3, 9, 1, 7);
```



```

Queue<Integer> queue = new ArrayDeque<>();
for (Integer element : init) {
    System.out.println("queue.offer element: " + element + ": " + queue.offer(element));
    System.out.println("queue: " + queue);
}
System.out.println();

// musimy wyjąć rozmiar do zmiennej,
// bo będzie się on zmieniał w trakcie działania programu
// możesz spróbować i zobaczyć co dziwnego się stanie
int size = queue.size();
for (int i = 0; i <= size; i++) {
    System.out.println("queue.peek: " + queue.peek());
    System.out.println("queue: " + queue);
    System.out.println("queue.poll: " + queue.poll());
    System.out.println("queue: " + queue);
}

```

Na ekranie wydrukuje się wtedy coś takiego:

```

queue.offer element: 3: true
queue: [3]
queue.offer element: 9: true
queue: [3, 9]
queue.offer element: 1: true
queue: [3, 9, 1]
queue.offer element: 7: true
queue: [3, 9, 1, 7]

queue.peek: 3
queue: [3, 9, 1, 7]
queue.poll: 3
queue: [9, 1, 7]
queue.peek: 9
queue: [9, 1, 7]
queue.poll: 9
queue: [1, 7]
queue.peek: 1
queue: [1, 7]
queue.poll: 1
queue: [7]
queue.peek: 7
queue: [7]
queue.poll: 7
queue: []
queue.peek: null
queue: []
queue.poll: null
queue: []

```

Patrząc na przykład wyżej możesz zwrócić uwagę, że elementy są dodawane w kolejności przychodzenia, i w tej samej kolejności są z kolejki wyjmowane. Czyli 3, 9, 1, 7. Kolejność zdejmowania jest taka sama, czyli 3, 9, 1, 7.

Ta sama klasa może być używana jak Stos, czyli ostatni położony na stosie element, będzie z niego zdjęty jako pierwszy.

```

List<Integer> init = Arrays.asList(3, 9, 1, 7);

Deque<Integer> stack = new ArrayDeque<>();
for (Integer element : init) {
    stack.push(element);
    System.out.println("stack: " + stack);
}
System.out.println();

int limit = stack.size();
for (int i = 0; i <= limit; i++) {
    System.out.println("stack.peek: " + stack.peek());
    System.out.println("stack: " + stack);
    System.out.println("stack.poll: " + stack.poll());
    System.out.println("stack: " + stack);
}

```

Na ekranie wydrukuje się wtedy coś takiego:

```

stack: [3]
stack: [9, 3]
stack: [1, 9, 3]
stack: [7, 1, 9, 3]

stack.peek: 7
stack: [7, 1, 9, 3]
stack.poll: 7
stack: [1, 9, 3]
stack.peek: 1
stack: [1, 9, 3]
stack.poll: 1
stack: [9, 3]
stack.peek: 9
stack: [9, 3]
stack.poll: 9
stack: [3]
stack.peek: 3
stack: [3]
stack.poll: 3
stack: []
stack.peek: null
stack: []
stack.poll: null
stack: []

```

Patrząc na przykład wyżej można zauważyć, że elementy są dodawane jak faktury kładzione na stole. Czyli na dole jest 3, potem kładziemy 9, potem 1, potem 7. Gdy zdejmujemy te liczby, zdejmujemy je w kolejności 7, 1, 9, 3.

PriorityQueue

PriorityQueue jest używana, gdy obiekty mają być przetwarzane na podstawie priorytetu. Wiadomo, że kolejka działa zgodnie z algorytmem first-in-first-out, ale czasami elementy kolejki muszą zostać przetworzone zgodnie z priorytetem, wtedy, w grę wchodzi PriorityQueue. Przykładowo kobieta w ciąży może mieć pierwszeństwo w kolejce. PriorityQueue jest oparta na zestawie priorytetów. Elementy

kolejki priorytetowej są uporządkowane zgodnie z implementowaną metodą `compareTo()` z interfejsu `Comparable` lub przez `Comparator` dostarczany w czasie budowy kolejki, w zależności od używanego konstruktora.

Elementy, przekazane na kolejkę, będą z niej zdejmowane zgodnie z tym jakbyśmy je posortowali.

```
List<Integer> init = Arrays.asList(3, 9, 1, 7);

Queue<Integer> queue = new PriorityQueue<>();
for (Integer element : init) {
    System.out.println("queue.offer " + element + ": " + queue.offer(element));
    System.out.println("queue: " + queue);
}
System.out.println();

int limit = queue.size();
for (int i = 0; i <= limit; i++) {
    System.out.println("queue.peak: " + queue.peak());
    System.out.println("queue: " + queue);
    System.out.println("queue.poll: " + queue.poll());
    System.out.println("queue: " + queue);
}
```

Na ekranie wydrukuje się wtedy coś takiego:

```
queue.offer 3: true
queue: [3]
queue.offer 9: true
queue: [3, 9]
queue.offer 1: true
queue: [1, 9, 3]
queue.offer 7: true
queue: [1, 7, 3, 9]

queue.peak: 1
queue: [1, 7, 3, 9]
queue.poll: 1
queue: [3, 7, 9]
queue.peak: 3
queue: [3, 7, 9]
queue.poll: 3
queue: [7, 9]
queue.peak: 7
queue: [7, 9]
queue.poll: 7
queue: [9]
queue.peak: 9
queue: [9]
queue.poll: 9
queue: []
queue.peak: null
queue: []
queue.poll: null
queue: []
```

Zwróć uwagę, że elementy są zdejmowane z kolejki zgodnie z tym jakbyśmy je posortowali, czyli 1, 3, 7,

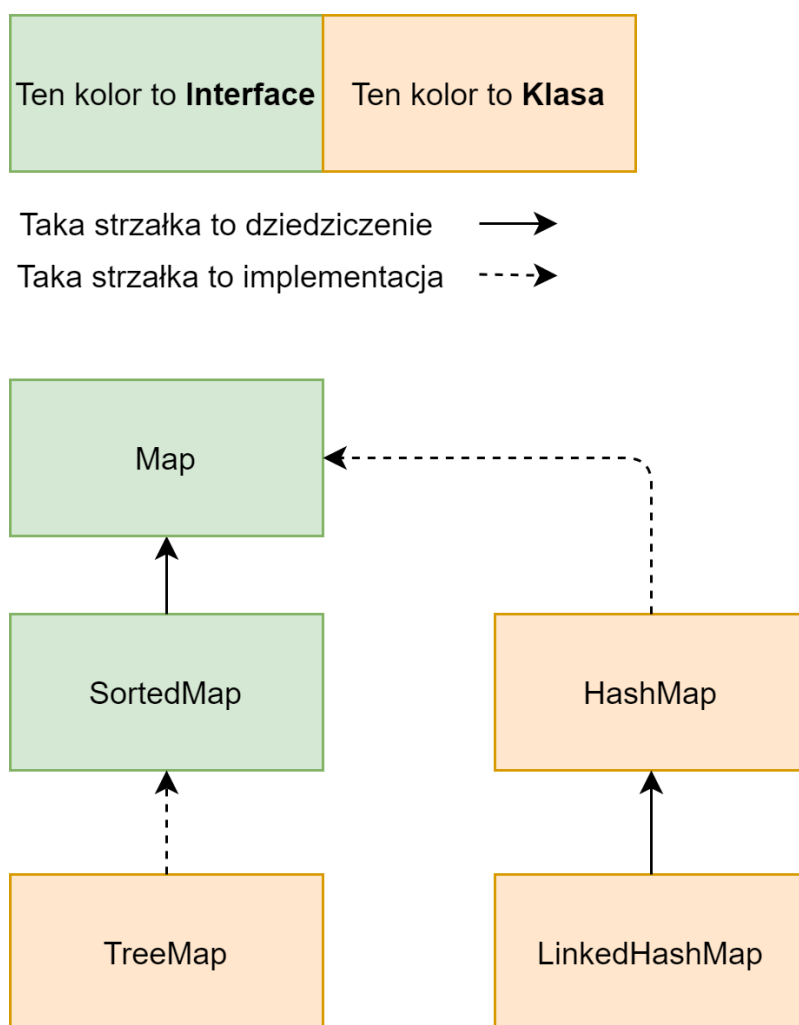
9. Ciekawe natomiast jest to, że na etapie przetwarzania elementów na kolejce, wcale nie są one posortowane tak jak będą z niej zdejmowane. Taka ciekawostka.

Możemy również przekazać do takiej kolejki własny komparator, który np. będzie zdejmował elementy z kolejki w porządku malejącym.

Należy wtedy zdefiniować **Comparator** w ten sposób:

```
Queue<Integer> queue = new PriorityQueue<>((o1, o2) -> o2 - o1);
```

Map



Obraz 6. Hierarchia Map

Map również jest interfejsem, który posiada różne implementacje, które różnią się pewnymi niuansami. Najczęściej używane to **HashMap**, **TreeMap** i **LinkedHashMap**. Wspomnimy też o **Hashtable** (zwróć uwagę na różnicę w wielkiej literze 't' poprzednio dla struktury **HashTable** była ona z wielkiej litery, a dla klasy **Hashtable** jest z małej)

Mapa to zbiór, który odwzorowuje klucze na wartości, bez możliwości przetrzymywania duplikatów kluczy. Elementy mapy to pary **klucz:wartość**.

Mapy należy używać, gdy chcemy identyfikować wartości za pomocą klucza. Na przykład, gdy

korzystamy z listy kontaktów w telefonie, wyszukujemy "Roman", zamiast przeglądać każdy numer telefonu po kolei.

Hashtable

Hashtable jest jak klasa **Vector**, ponieważ jest naprawdę stary i napisany z myślą o używaniu wielowątkowym. W nazwie pojawia się litera 't', która zgodnie z nazwą struktury powinna być z wielkiej litery, ale to błąd starych czasów. W formie analogii do starych klas można powiedzieć, że **ArrayList** do **Vector** ma się tak samo, jak **HashMap** do **Hashtable**. **Hashtable** to klasa w Javie. **HashTable** to struktura danych, na której ta klasa bazuje. Taka ciekawostka 😊.

HashMap

HashMap przechowuje klucze zgodnie ze strukturą **HashTable**. Oznacza to, że ponownie, używamy metody **hashCode()** do przechowywania danych i wydajniejszego pobierania wartości. Konsekwencją używania tej struktury jest utrata kolejności wstawiania elementów.

W większości przypadków i tak się tym nie przejmujemy. Jeżeli natomiast nam na tym zależy, możemy wykorzystać **LinkedHashMap**.

Biorąc pod uwagę, że **Map** nie rozszerza **Collection**, w interfejsie **Map** określono więcej metod. Ponieważ w **Map** istnieją zarówno klucze, jak i wartości, definiując Mapę określamy typ generyczny zarówno dla klucza jak i dla wartości.

Przykład użycia:

```
Map<String, String> cars = new HashMap<>();
cars.put("Volvo", "XC60");
cars.put("Fiat", "Panda");
cars.put("Volkswagen", "Golf");

// Get key: XC60
System.out.println("Get key: " + cars.get("Volvo"));
// Remove key: Panda
System.out.println("Remove key: " + cars.remove("Fiat"));
// Size: 2
System.out.println("Size: " + cars.size());
// isEmpty: false
System.out.println("isEmpty: " + cars.isEmpty());
// containsKey: true
System.out.println("containsKey: " + cars.containsKey("Volkswagen"));
// containsValue: true
System.out.println("containsValue: " + cars.containsValue("Golf"));
```

HashMap posiada taką klasę jak **Entry**, która jest klasą zagnieżdżoną (o których dowiemy się już niebawem). Stąd też żeby dostać się do klasy **Entry**, można napisać to w ten sposób:

```
Set<Map.Entry<String, String>> entries = cars.entrySet();
```

Otrzymujemy dzięki temu **Set** "wystąpień", czyli **Entriesów**, czyli wszystkich par 'klucz:wartość' dodanych do Mapy.

Należy pamiętać o klasie `Entry`, gdyż implementacja `HashMap` wylicza `hashCode()` z klucza, który jest zdefiniowany w `Entry`, a nie z jej całości. Na tej podstawie następnie następuje przetwarzanie tak jak w pokazanej strukturze `HashTable`.

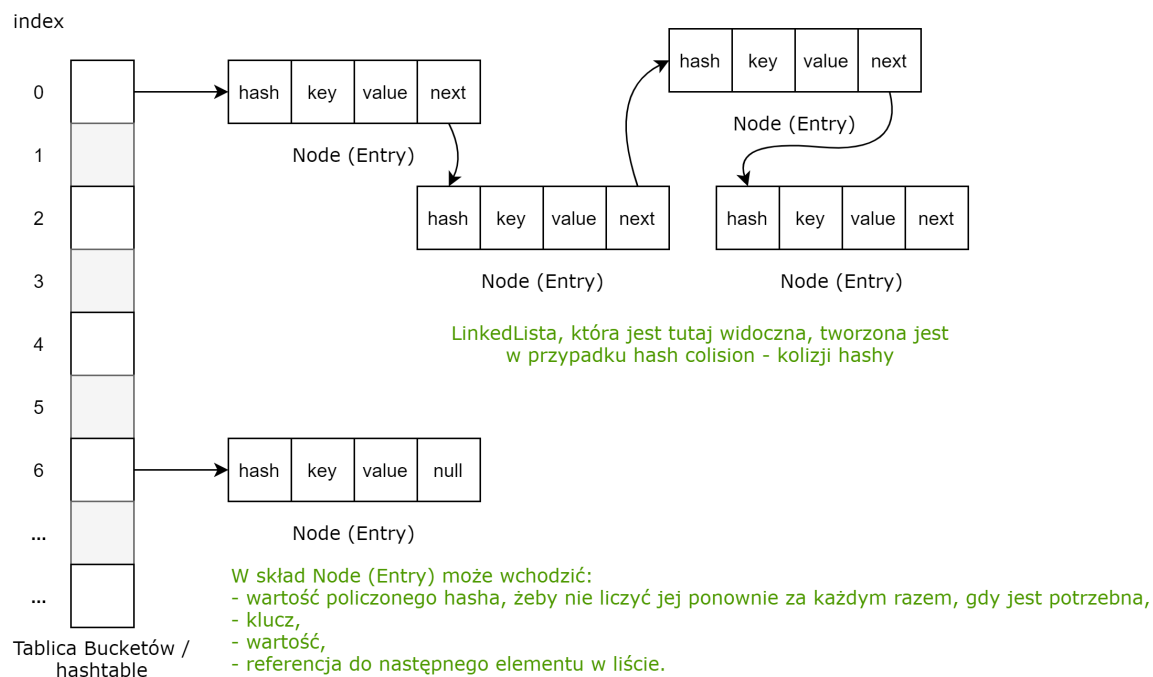
Czyli wyliczamy `hashCode()` z klucza tej pary i `Entry` umieszczamy w odpowiednim miejscu w tablicy. Jeżeli wystąpi sytuacja, w której kilka `Entries`ów ma taką samą wartość `hashCode()`, umieszczamy je w tym samym miejscu w tablicy, natomiast przechowujemy je wtedy w liście.

Konkretnie miejsce w tablicy *hash table*, która przetrzymuje nam elementy, nazywane jest bucketem (wiaderkiem). W jednym wiaderku możemy mieć listę elementów, które mają taką samą wartość `hashCode()`.

Jeszcze raz wspomnę o przypadku, gdy mamy więcej różnych obliczonych wartości `hashCode()` niż wiaderek - należy wtedy zastosować `hashCode() % LiczbaWiaderek`.

W praktyce może to wyglądać w ten sposób:

Jeżeli obliczona wartość hashCode() jest większa niż ilość bucketów, możemy zastosować modulo, czyli hashCode() % ilośćBucketów. Wtedy gwarantujemy, że index w tablicy zawsze będzie w zakresie 0 - ilośćBucketów.



Obraz 7. HashMap działanie

Kolizja hashy to taka sytuacja, w której, do jednego bucketa trafia kilka nodów (entriesów), bo mają tę samą wartość `hashCode()`. Można ją rozwiązać dodając elementy wpadające do tego samego bucketa do listy. W praktyce może to być np. `LinkedList`, gdyż częściej będziemy do takiej listy zapisywać elementy na jej końcu niż te elementy odczytywać.

Na końcu jeszcze raz napiszmy, że gdy szukamy jakiegoś klucza w mapie, w pierwszej kolejności policzymy dla szukanego klucza wartość `hashCode()`, następnie mając tę wartość znajdziemy wiaderko (bucket), w którym szukany klucz może się znajdować i jeżeli w danym wiaderku będzie kilka elementów na liście, to z każdym kluczem na liście będziemy musieli dokonać porównania poprzez wykorzystanie metody `equals()`. Jeżeli taki klucz w mapie występuje, to przyspieszamy jego znalezienie szukając tylko w konkretnym wiaderku. To jednocześnie pokazuje dlaczego ważny jest kontrakt między

`equals()` a `hashCode()`.

Kolejny niuans jest taki, że jeżeli szukanego klucza w mapie nie ma, to możemy to wykryć na wczesnym etapie, bo np. dane wiaderko może być puste i wtedy szybko skończyliśmy przeszukiwanie elementów.

Java 8

Należy tutaj również dołożyć kolejną cegielkę. Przykładowy sposób działania `HashMap` pokazuje, że w jednym wiaderku możemy mieć listę elementów. W Java 8 została wprowadzona pewna modyfikacja mająca na celu poprawę wydajności działania. Jeżeli ilość elementów w danym wiaderku przekroczy pewien próg, który jest określany parametrem `TREEIFY_THRESHOLD`, wspomniana lista jest zamieniana na drzewo binarne. Gdy zaczniesz szukać w internecie ile wynosi wartość `TREEIFY_THRESHOLD` znajdziesz informację, że jest to 8. Wyjaśnieniem czemu struktura zostaje zamieniona z listy na drzewo jest poprawa optymalności przeszukiwań elementu. Różne struktury pozwalają znaleźć elementy w innym czasie. W przypadku przekroczenia pewnej ilości elementów, drzewo pozwoli znaleźć szukany element szybciej niż lista. Będzie bardziej zrozumiałe gdy wyjaśnimy sobie czym jest notacja **Big O**, o czym możesz przeczytać na końcu tej notatki.

Są to natomiast szczegóły implementacyjne, najważniejsze jest to, żeby pamiętać logikę działania mapy "pod maską" ☺.

Podsumowując, wiem, że wypisane tutaj zostało dużo wiedzy, ale jest ona ważna. Powodzenia w jej przyswojeniu ☺.

TreeMap

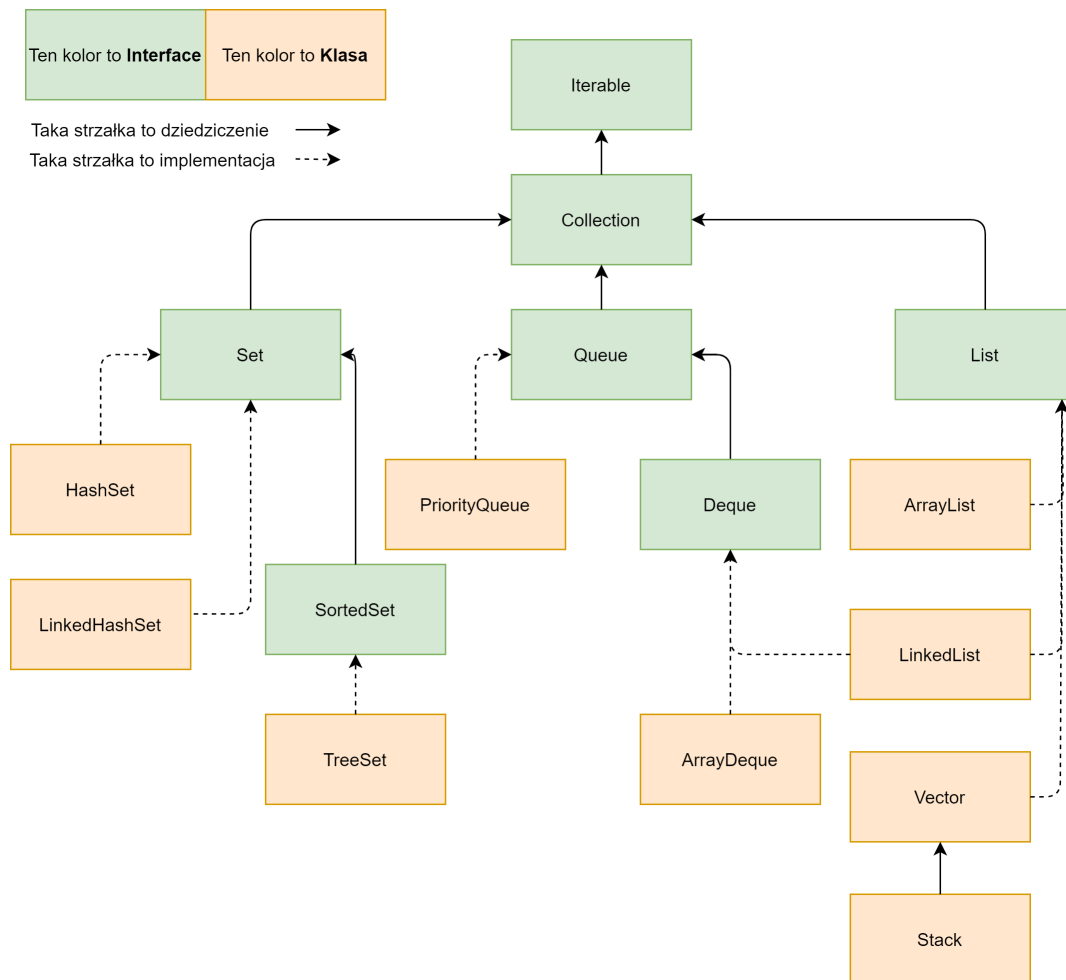
`TreeMap` przechowuje swoje elementy w posortowanej strukturze drzewa. Główną zaletą jest to, że zestaw jest zawsze posortowany na podstawie kluczy Mapy. Jeżeli chcemy korzystać z `TreeMap` z klasą napisaną przez nas, należy pamiętać, żeby albo określić `Comparator`, którego chcemy używać, albo żeby nasza klasa, która będzie nam służyła za klucz w `TreeMap` implementowała interfejs `Comparable`. Inaczej dostaniemy błąd w trakcie działania programu.

LinkedHashMap

`LinkedHashMap` używamy gdy zachodzi potrzeba utrzymania kolejności iteracji elementów. Iterując po `HashMap`, kolejność elementów jest nieprzewidywalna, podczas gdy `LinkedHashMap` pozwala nam iterować po elementach w kolejności, w jakiej zostały wstawione.

Podsumowanie

Postarajmy się w tym momencie jeszcze raz ułożyć sobie w głowie wszystkie kolekcje, które zostały dotychczas pokazane. Rzuć zatem okiem jeszcze raz na poniższą grafikę.



Obraz 8. Hierarchia Collections

Notacja Big O

Chcę lekko podrapać po powierzchni temat notacji **Big O**, czyli np. $O(n)$. W momencie gdy zaczniesz czytać chociażby o różnych algorytmach sortujących, zaczną się tam pojawiać zapisy podobne do $O(n)$, $O(n^2)$ itp. Poniżej chcę ogólnie wyjaśnić, co te zapisy oznaczają.

Na co to komu?

W programowaniu komputerowym używamy notacji **Big O**, aby mówić o wydajności algorytmów. Gdyby się nad tym zastanowić, to jest całkiem jasne, że wykonanie tej samej czynności przez może zająć 1 sekundę albo 10 sekund. Lepiej jest stosować algorytmy, które dokładnie to samo mogą osiągnąć w krótszym niż dłuższym czasie.

Jeżeli chodzi o wydajność algorytmów, nie można odnosić się do wydajności algorytmów w sekundach, bo każdy komputer wykona ten sam algorytm w innym czasie, ze względu na różnicę parametrów, RAM, CPU, obecną zajętość zasobów. Notacja ta daje 'uniwersalną' metodę porównania poziomu złożoności zadań.

Notacja **Big O** pozwala porównać rząd wielkości wydajności, a nie dokładną wydajność. Zakłada również najgorszy czas trwania. Jeśli napiszesz algorytm, który może zająć trochę czasu lub być natychmiastowy, notacja odnosi się do tego dłuższego. Oznaczenie małe **n** jest używane, aby odzwierciedlić liczbę elementów lub rozmiar danych, o których mówisz. Często w przypadku notacji **Big**

O będziemy odnosić się do czasu wykonania algorytmu, ale nie w rozumieniu sekund tylko zależności czasu trwania od ilości elementów, które trzeba przetworzyć. Poniżej wymieniam najczęstsze wartości notacji duże O, które zobaczysz, i ich znaczenie:

- **O(1)** - stały czas - nie ma znaczenia, jak duża jest ilość elementów do przetworzenia przez algorytm. Przykładowo, zwróć ostatni element listy. Za każdym razem, wykonanie takiej operacji powinno zająć stały czas.
- **O(log n)** - logarytmiczny czas - logarytm to funkcja matematyczna, która rośnie znacznie wolniej niż rozmiar danych. Można sobie to zapamiętać w ten sposób, że jak zobaczymy algorytm z czasem logarytmicznym to powinno być szybciej niż z liniowym. Wyszukiwanie binarne (binary search) działa w czasie logarytmicznym, ponieważ nie obejmuje większości elementów dużych kolekcji.
- **O(n)** - liniowy czas - wydajność będzie rosła liniowo w stosunku do wielkości kolekcji. Zapętlenie listy i zwrócenie liczby elementów pasujących do słowa "Java" zajmie liniowy czas, bo musimy sprawdzić wszystkie elementy w kolekcji.
- **O(n²)** - kwadratowy czas - kod zawierający pętlę w pętli, w których każda pętla przechodzi przez dane, zajmuje n^2 . Przykładem może być wydrukowanie każdego elementu z listy jako para z każdym innym elementem z listy. Czyli jeżeli mamy listę [1, 5, 8], to możemy mieć przykładowo 3 takie pary: [1, 5], [1, 8], [5, 8]. Aby stworzyć taką parę musimy przeiterować listę 2 razy, czyli mieć 2 zagnieżdżone pętle, stąd $O(n^2)$.

Jeżeli teraz zobaczysz **tabelę**, która przykładowo porównuje ze sobą algorytmy sortujące, będziesz w stanie zrozumieć, w czym te oznaczenia pomagają 😊.

Przypadki

Gdy mówimy o wydajności algorytmów możemy rozróżniać przypadki. W praktyce spotkamy się z takimi stwierdzeniami:

- **worst-case** - najgorszy możliwy czas trwania danego zadania.
- **average-case** - uśredniony czas trwania, biorąc pod uwagę wszystkie możliwe rodzaje danych wejściowych.
- **amortized analysis** - analiza zamortyzowana określa czas niezbędny do wykonania ciągu operacji, bo może się okazać, że wykonanie ciągu operacji jest mniej kosztowne niż wskazuje na to złożoność pojedynczej operacji.

Projekt

Napisz program, w którym zdefiniujesz klasę **Osoba** posiadającą pola **name**, **surname** oraz **counter**. **Osoba** może przyjść zrobić zakupy i zderzyć się z kolejką w sklepie. Taka osoba może się w kolejce ustawić i czekać. Jeżeli poczeka, to może przejść jej kolej i może wejść do sklepu. Może też dojść do wniosku, że ma dość i sobie pójść. Pozwól użytkownikowi swojego programu na zabawę taką kolejką, dając możliwość wykonania operacji, które zostały wspomniane, czyli dodanie osoby do kolejki, kolejka przesuwa się do przodu, osoba odpuszcza i opuszcza kolejkę.

Jeżeli do kolejki dodasz osobę o tym samym imieniu i nazwisku parę razy, to w każdej z nich ma się zwiększać **counter**, po to aby móc zdefiniować 2 osoby np. **TOMASZ_ROMAŃSKI_1** oraz **TOMASZ_ROMAŃSKI_2**. Potrzebne jest to do dalszej logiki w programie. Jeżeli **TOMASZ_ROMAŃSKI_1** nie będzie już stał w kolejce, i przyjdzie kolejny **TOMASZ_ROMAŃSKI**, to ma on otrzymać numer **3**, a nie **1**.

Wykorzystaj klasę **Scanner** od odczytu danych od użytkownika. Dane takie będziemy wprowadzać w terminalu. Napisz też wariant z danymi zapisanymi na sztywno w liście.

Program będzie przyjmował od użytkownika komendy tekstowe, musimy więc ustalić na jakie zachowania ma reagować. Zatem jeżeli wpiszesz:

- **ADD PERSON(Tomasz_Romański)** - taka osoba ustawi się w kolejce, podłogą oddzielamy imię i nazwisko. Jeżeli zduplikujesz osobę, zostanie zwiększony **counter**. Osoba może mieć tylko jedno imię i nazwisko.
- **PROCESS** - kolejka przesuwa się o jedną osobę do przodu, czyli każda osoba czekająca w kolejce przesuwa się o jedno miejsce do przodu
- **LEAVE PERSON(Tomasz_Romański_2)** - osoba o takim imieniu i nazwisku (tutaj widzisz po co jest to pole **counter**, żeby jednoznacznie identyfikować, o której osobie mówimy) odpuszcza stanie w kolejce i sobie idzie. Pozostałe osoby wtedy przesuwać się w kolejce do przodu.

Jeżeli program nie rozpozna komendy, ma wypisać na ekranie: "O co Ci chodzi?". Zapewnij, aby przy każdym kroku na ekranie drukowała się zawartość kolejki.

Nie zostaje narzucona obsługa przypadków skrajnych, czyli np, jak ma się zachować program jeżeli powiemy, że kolejkę musi opuścić osoba, której na tej kolejce nie ma. Pozwalam Ci wymyślić samodzielnie co zrobić w takich sytuacjach 😊.

Przykład sekwencji wejściowej:

```
ADD PERSON(Tomasz_Romański)
ADD PERSON(Rafał_Adamczuk)
ADD PERSON(Tomasz_Romański)
ADD PERSON(Mariusz_Wilczek)
ADD PERSON(Zbigniew_Ratownik)
PROCESS
LEAVE PERSON(Tomasz_Romański_2)
LEAVE PERSON(Mariusz_Wilczek)
PROCESS
PROCESS
PROCESS
```

Przykład co może być wydrukowane na ekranie:

```
ADD PERSON(Tomasz_Romański)
Tomasz_Romański_1 came to the queue: true
Queue: [Tomasz_Romański_1]

ADD PERSON(Rafał_Adamczuk)
Rafał_Adamczuk_1 came to the queue: true
Queue: [Tomasz_Romański_1, Rafał_Adamczuk_1]

ADD PERSON(Tomasz_Romański)
Tomasz_Romański_2 came to the queue: true
Queue: [Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2]

ADD PERSON(Mariusz_Wilczek)
Mariusz_Wilczek_1 came to the queue: true
Queue: [Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2, Mariusz_Wilczek_1]

ADD PERSON(Zbigniew_Ratownik)
Zbigniew_Ratownik_1 came to the queue: true
Queue: [Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2, Mariusz_Wilczek_1, Zbigniew_Ratownik_1]

PROCESS
Processing queue: Tomasz_Romański_1 arrived at the store
Queue: [Rafał_Adamczuk_1, Tomasz_Romański_2, Mariusz_Wilczek_1, Zbigniew_Ratownik_1]

LEAVE PERSON(Tomasz_Romański_2)
Leaving queue: Tomasz_Romański_2
Queue: [Rafał_Adamczuk_1, Mariusz_Wilczek_1, Zbigniew_Ratownik_1]

PROCESS
Processing queue: Rafał_Adamczuk_1 arrived at the store
Queue: [Mariusz_Wilczek_1, Zbigniew_Ratownik_1]

PROCESS
Processing queue: Mariusz_Wilczek_1 arrived at the store
Queue: [Zbigniew_Ratownik_1]

PROCESS
Processing queue: Zbigniew_Ratownik_1 arrived at the store
Queue: []
```

Następnie dodaj opcję VIPów, czyli ludzi, którzy jak przyjdą to trafiają na początek kolejki. Nie narzucam kryteriów ustalania kto ma być pierwszy w kolejce jak przyjdzie 2 VIPów, możesz to ustalić jak chcesz. Jeżeli jeszcze nie rozumiesz czemu o tym piszę, to zrozumiesz pisząc kod 😊. Do wiadomości wysyłanych do Twojego programu dodaj takie parametry:

- `ADD PERSON(Tomasz_Romański,VIP)` - taki gość jest VIPem,
- `ADD PERSON(Tomasz_Romański)` - a taki nie jest,
- `PROCESS` oraz `LEAVE PERSON(Tomasz_Romański_2)` zostają tak jak były wcześniej.

Spróbuj wymyślić jak można to zrealizować, nie daję więcej wskazówek. Przykład sekwencji wejściowej:

```
ADD PERSON(Tomasz_Romański)
ADD PERSON(Rafał_Adamczuk)
```

```
ADD PERSON(Tomasz_Romański)
ADD PERSON(Mariusz_Wilczek,VIP)
ADD PERSON(Zbigniew_Ratownik)
ADD PERSON(Roman_Tomaszewski,VIP)
PROCESS
LEAVE PERSON(Zbigniew_Ratownik)
LEAVE PERSON(Tomasz_Romański_2)
PROCESS
PROCESS
PROCESS
```

Przykład co może być wydrukowane na ekranie:

```
ADD PERSON(Tomasz_Romański)
Tomasz_Romański_1 came to the queue: true
Queue: [Tomasz_Romański_1]

ADD PERSON(Rafał_Adamczuk)
Rafał_Adamczuk_1 came to the queue: true
Queue: [Tomasz_Romański_1, Rafał_Adamczuk_1]

ADD PERSON(Tomasz_Romański)
Tomasz_Romański_2 came to the queue: true
Queue: [Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2]

ADD PERSON(Mariusz_Wilczek,VIP)
Mariusz_Wilczek_1_VIP came to the queue: true
Queue: [Mariusz_Wilczek_1_VIP, Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2]

ADD PERSON(Zbigniew_Ratownik)
Zbigniew_Ratownik_1 came to the queue: true
Queue: [Mariusz_Wilczek_1_VIP, Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2,
Zbigniew_Ratownik_1]

ADD PERSON(Roman_Tomaszewski,VIP)
Roman_Tomaszewski_1_VIP came to the queue: true
Queue: [Mariusz_Wilczek_1_VIP, Roman_Tomaszewski_1_VIP, Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2, Zbigniew_Ratownik_1]

PROCESS
Processing queue: Mariusz_Wilczek_1_VIP arrived at the store
Queue: [Roman_Tomaszewski_1_VIP, Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2, Zbigniew_Ratownik_1]

LEAVE PERSON(Zbigniew_Ratownik)
Leaving queue: Zbigniew_Ratownik_1
Queue: [Roman_Tomaszewski_1_VIP, Tomasz_Romański_1, Rafał_Adamczuk_1, Tomasz_Romański_2]

LEAVE PERSON(Tomasz_Romański_2)
Leaving queue: Tomasz_Romański_2
Queue: [Roman_Tomaszewski_1_VIP, Tomasz_Romański_1, Rafał_Adamczuk_1]

PROCESS
Processing queue: Roman_Tomaszewski_1_VIP arrived at the store
Queue: [Tomasz_Romański_1, Rafał_Adamczuk_1]

PROCESS
Processing queue: Tomasz_Romański_1 arrived at the store
Queue: [Rafał_Adamczuk_1]
```

PROCESS

Processing queue: Rafał_Adamczuk_1 arrived at the store

Queue: []

Programowanie funkcyjne, interfejsy funkcyjne - zadania

1. Napisz implementację interfejsu funkcyjnego `java.util.function.BiFunction`, który przyjmie `Integer` oraz `Double`, a zwróci `String`. Implementacja w trakcie wywołania wydrukuj na ekranie 2 przekazane argumenty, a na koniec zwróci dodane do siebie `Integer` oraz `Double` jako `String` stosując konkatenację. Pisząc implementację tego interfejsu użyj mechanizmu `method reference`.
2. Napisz interfejs funkcyjny z metodą, która przyjmie od Ciebie `String`, `Integer` oraz Dowolny obiekt. Typem zwracanym z takiej metody będzie `Integer`. Napisz 2 implementacje funkcyjne takiej metody:
 - Pierwsza ma zwracać sumę długości przekazanego `Stringa`, przekazanego `Integera` oraz długości `Stringa` zwróconego z wywołanej metody `toString()` na przekazanym ostatnim obiekcie do metody.
 - Druga ma dodać do siebie liczbę wystąpień litery 'a' w pierwszym `Stringu`, wartość przekazaną jako `Integer` oraz ilość wystąpień przecinków zsumowaną z ilością wystąpień '=' w metodzie `toString()` wywołanej na przekazanym ostatnim obiekcie.
3. Napisz implementację interfejsu `BinaryOperator`, która z 2 przekazanych do niej obiektów zwróci większy. Jeżeli przekazane 2 obiekty są sobie równe, zwróć `null`, a potem obsłuż go wykorzystując `Optional`, który wyrzuci wyjątek, w przypadku gdy obiekty są sobie równe.

Zaawansowane kolekcje - zadania

Spis treści

Zadanie 1	1
Zadanie 2	1
Zadanie 3	1

Zadanie 1

Wyobraź sobie, że idziesz do muzeum motoryzacji i wypisujesz sobie samochody, które chcesz tam zobaczyć. Wypisujesz informacje takie jak: marka, model oraz rocznik. Idziesz tam z inną osobą i ona też wypisuje swoje. Przedstaw to co napisaliście w postaci 2 list zawierających obiekty klasy `Car`. Klasa `Car` ma posiadać pola `brand`, `model` oraz `year`. Następnie sklej ze sobą te 2 listy, tak aby wyeliminować powtarzające się samochody, które dodaliście i przedstaw zawartość na ekranie posortowaną najpierw po roku rosnąco, potem po marce rosnąco oraz po modelu rosnąco. Załóżmy, że taka byłaby kolejność oglądania eksponatów w muzeum.

Zadanie 2

Napisz implementację książki telefonicznej, która umożliwi wprowadzenie przez użytkownika osoby wraz z numerem telefonu, w sposób pokazany poniżej:

- Jan Kowalski +48000192871
- Stefan Nawrocki +48000125847
- Anna Tomaszewska +48000421458

Dodaj te osoby do Mapy w taki sposób, żeby kluczem było nazwisko, a wartością zbiór osób występujących pod takim nazwiskiem, posortowanych w kolejności nazwisko, a potem imię. Po każdej dodanej osobie, wyświetl zawartość mapy na ekranie. Spraw, żeby osoby były posortowane w książce po nazwisku. Dla uproszczenia załóż, że osoby mogą mieć tylko jedno imię i tylko jedno nazwisko.

Zadanie 3

Napisz program, w którym stworzysz klasę `Invoice`, zawierającą pola `id` oraz `amount`. Wykorzystaj klasę `Scanner` do odczytu danych od użytkownika (wiem, że jeszcze jej nie omawialiśmy, w internecie znajdziesz dużo przykładów). Pozwól użytkownikom dodawać faktury, które będą odkładane na stosie do przetworzenia. W swoim programie zdefiniuj 2 osoby, jedną która będzie faktury dodawała (Prezes - `CEO`) oraz drugą która będzie te faktury przetwarzała (Księgowy - `ACCOUNTANT`). `CEO` korzystając z tego programu może dodać nową fakturę, a księgowy fakturę już istniejącą przetworzyć i zdjąć ze stosu. Księgowy może zdjąć i przetworzyć tylko pierwszą fakturę z góry stosu. Z racji, że program będzie przyjmował od użytkownika komendy tekstowe, musimy ustalić na jakie zachowania ma reagować. Załóżmy zatem, że jeżeli wpiszemy:

- "CEO: ADD INVOICE(190.20)" - oznacza to, że **CEO** dodaje fakturę na kwotę **190.20** i trafia ona na stos do rozliczenia. Identyfikator faktury to może być np. **1/2020**, gdzie **1** jest counterem dodawanym automatycznie przez program, natomiast rok jest dodawany automatycznie na podstawie obecnego roku.
- "ACCOUNTANT: PROCESS INVOICE" - oznacza to, że księgowy rozlicza fakturę z identyfikatorem **1/2020** i zostaje ona zdjęta ze stosu do rozliczania.

Jeżeli program nie rozpoznaje komendy, ma wypisać na ekranie: "O co Ci chodzi?". Zapewnij, aby przy każdym kroku na ekranie drukowała się zawartość stosu wraz z sumą kwot faktur jeszcze nierozliczonych. Nie narzucam obsługi przypadków skrajnych, czyli np, jak ma się zachować księgowy jak wpisze mu komendę "rozlicz fakturę", a nie ma żadnych faktur. Pozwalam Ci wymyślić co zrobić w takich sytuacjach ☺. Przykład danych wejściowych:

```
CEO: ADD INVOICE(19.20)
CEO: ADD INVOICE(31.40)
ACCOUNTANT: PROCESS INVOICE
CEO: ADD INVOICE(21.20)
CEO: ADD INVOICE(120.99)
ACCOUNTANT: PROCESS INVOICE
ACCOUNTANT: PROCESS INVOICE
ACCOUNTANT: PROCESS INVOICE
```

Przykład co może zostać wydrukowane na ekranie:

```
CEO: ADD INVOICE(123.40)
CEO added invoice: ID: 1/2021, amount: 123.40
Current invoices amount: 123.40. Invoices: [{ID: 1/2021, AMOUNT: 123.40}]

CEO: ADD INVOICE(12.41)
CEO added invoice: {ID: 2/2021, AMOUNT: 12.41}
Current invoices amount: 135.81. Invoices: [{ID: 2/2021, AMOUNT: 12.41}, {ID: 1/2021, AMOUNT: 123.40}]

CEO: ADD INVOICE(492.10)
CEO added invoice: {ID: 3/2021, AMOUNT: 492.10}
Current invoices amount: 627.91. Invoices: [{ID: 3/2021, AMOUNT: 492.10}, {ID: 2/2021, AMOUNT: 12.41}, {ID: 1/2021, AMOUNT: 123.40}]

ACCOUNTANT: PROCESS INVOICE
ACCOUNTANT processed invoice: {ID: 3/2021, AMOUNT: 492.10}
Current invoices amount: 135.81. Invoices: [{ID: 2/2021, AMOUNT: 12.41}, {ID: 1/2021, AMOUNT: 123.40}]

CEO: ADD INVOICE(121.40)
CEO added invoice: {ID: 4/2021, AMOUNT: 121.40}
Current invoices amount: 257.21. Invoices: [{ID: 4/2021, AMOUNT: 121.40}, {ID: 2/2021, AMOUNT: 12.41}, {ID: 1/2021, AMOUNT: 123.40}]

CEO: ADD INVOICE(812.41)
CEO added invoice: {ID: 5/2021, AMOUNT: 812.41}
Current invoices amount: 1069.62. Invoices: [{ID: 5/2021, AMOUNT: 812.41}, {ID: 4/2021, AMOUNT: 121.40}, {ID: 2/2021, AMOUNT: 12.41}, {ID: 1/2021, AMOUNT: 123.40}]
```

```
ACCOUNTANT: PROCESS INVOICE
ACCOUNTANT processed invoice: {ID: 5/2021, AMOUNT: 812.41}
```


Current invoices amount: 257.21. Invoices: [{ID: 4/2021, AMOUNT: 121.40}, {ID: 2/2021, AMOUNT: 12.41}, {ID: 1/2021, AMOUNT: 123.40}]

ACCOUNTANT: PROCESS INVOICE

ACCOUNTANT processed invoice: {ID: 4/2021, AMOUNT: 121.40}

Current invoices amount: 135.81. Invoices: [{ID: 2/2021, AMOUNT: 12.41}, {ID: 1/2021, AMOUNT: 123.40}]

ACCOUNTANT: PROCESS INVOICE

ACCOUNTANT processed invoice: {ID: 2/2021, AMOUNT: 12.41}

Current invoices amount: 123.40. Invoices: [{ID: 1/2021, AMOUNT: 123.40}]

ACCOUNTANT: PROCESS INVOICE

ACCOUNTANT processed invoice: {ID: 1/2021, AMOUNT: 123.40}

Current invoices amount: 0. Invoices: []