Java zaawansowana

v3.1

Plan

1. Kolekcje
2. Wyrażenia regularne

Kolekcje

Kolekcje

Kolekcje – są to klasy, które służą do przechowywania innych obiektów. Należą do pakietu **java.util**.

Posiadają mechanizmy umożliwiające wykonywanie operacji:

- na pojedynczych elementach, np.:
  - pobieranie obiektów,
  - wstawianie obiektów,
  - usuwanie obiektów,
- zbiorowych operacji na grupach obiektów.

Najważniejsze z nich to:

- **`List`** – znają indeks (położenie) elementu.
- **`Set`** – nie pozwalają na przechowywanie duplikatów.
- **`Map`** – przechowują pary klucz-wartość.

UWAGA: Gdybyśmy chcieli być bardzo precyzyjni, to mapy nie są prawdziwymi kolekcjami, ponieważ implementują inny interfejs – natomiast ich przeznaczenie jest zbliżone.

Kolekcje vs. tablice

Kolekcje:

- Rozmiar nieokreślony – nie trzeba go deklarować przy tworzeniu.
- Przechowują tylko referencje do obiektów.
- Posiadają specjalne metody do obsługi przechowywanych elementów.

Tablice:

- Stały rozmiar – deklarowany już przy tworzeniu.
- Przechowują zmienne typu prostego lub referencje do obiektów.
- Nie posiadają dodatkowych metod do obsługi przechowywanych elementów.

Co to jest interfejs?

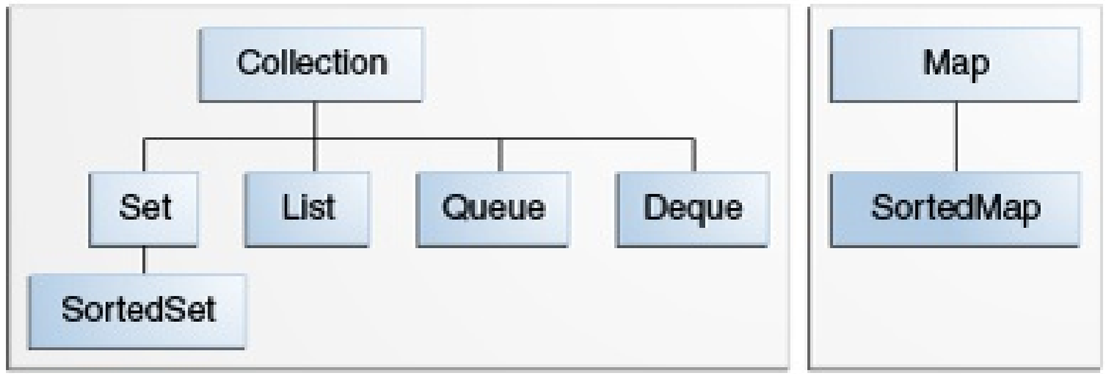
- Interfejs posiada prostszą budowę niż klasa. Przeważnie zawiera tylko definicje metod (lub stałych), ale bez ich implementacji (wyjątek stanowią metody domyślne – więcej o nich dowiemy się podczas omawiania ósmej wersji Javy, w której zostały wprowadzone).

- Interfejs pozwala określić, jakie metody musi zaimplementować klasa. Nie określa jednak, jak te metody powinny działać (podobnie jak klasa abstrakcyjna).

- Dzięki interfejsom różne klasy mogą wykorzystywać te same funkcjonalności bez konieczności dziedziczenia.

Hierarchia kolekcji

Hierarchię kolekcji obrazuje poniższy schemat – poszczególne klasy implementują określone interfejsy, co z kolei determinuje zestaw ich możliwości.



Klasa implementująca interfejs musi zawierać wszystkie metody tego interfejsu.

O tym jak tworzyć własne interfejsy, a na ich podstawie konkretne implementacje, dowiemy się w jednym z kolejnych modułów.

Interfejs **Collection** – metody

Interfejs **Collection** stanowi podstawę dla wszystkich kolekcji.

Zawiera m.in. następujące metody:

- `boolean add(E e)` – dodaje element do kolekcji – jeżeli ta operacja się uda, metoda zwraca `true`.
- `boolean addAll(Collection<? extends E> c)` – dodaje do kolekcji wszystkie elementy z kolekcji `c` – jeżeli uległa ona zmianie, zwraca `true`.
- `void clear()` – usuwa wszystkie elementy z kolekcji.
- `boolean contains(Object o)` – zwraca `true` jeśli element `o` jest w kolekcji.
- `boolean containsAll(Collection<?> c)` – zwraca `true` jeśli wszystkie elementy z kolekcji `c` są w kolekcji, na której wywołujemy metodę.
- `boolean isEmpty()` – zwraca `true`, gdy kolekcja jest pusta.

Interfejs **Collection** – metody c.d.

- `Iterator <E> iterator()` – zwraca iterator, którego możemy użyć do poruszania się po kolekcji.
- `boolean remove(Object o)` – usuwa element kolekcji, zwraca `true` jeśli się udało.
- `boolean removeAll(Collection<?> c)` – usuwa wszystkie elementy znajdujące się w kolekcji `c`, zwraca `true` jeśli usunął chociaż jeden element.
- `boolean retainAll(Collection<?> c)` – zachowuje tylko wspólne elementy kolekcji, pozostałe usuwa – zwraca `true` jeśli kolekcja się zmieniła.
- `int size()` – zwraca liczbę elementów w kolekcji.
- `Object[] toArray()` – przekształca kolekcję na tablicę.

Interfejs **List** – metody

Interfejs **`List`** zawiera dodatkowe metody, m.in.:

- **`void add(int index, E element)`** – dodaje obiekt pozycji na określonej przez indeks.
- **`E get(int index)`** – pobiera element z określonej pozycji.
- **`int indexOf(Object o)`** – zwraca indeks pierwszego wystąpienia obiektu **`o`** lub **`-1`**, gdy obiekt nie występuje w kolekcji.
- **`int lastIndexOf(Object o)`** – zwraca indeks ostatniego wystąpienia na liście obiektu **`o`**, lub -1 gdy nie występuje.
- **`E remove(int index)`** – usuwa z listy element o określonej pozycji, a następnie go zwraca.
- **`E set(int index, E element)`** – zamienia obiekt na pozycji określonej przez indeks.
- **`List<E> subList(int start, int end)`** – zwraca listę utworzoną z elementów listy wyjściowej o indeksach od start do end-1.

Klasa ArrayList

Klasa **`ArrayList`** implementuje interfejs **`List`**. Jest jedną z najpopularniejszych kolekcji.

Elementy na liście mogą się powtarzać.

Listę tworzymy w taki sam sposób jak znane nam dotychczas obiekty.

```
List myList = new ArrayList();
```

Klasa ArrayList

Klasa **`ArrayList`** implementuje interfejs **`List`**. Jest jedną z najpopularniejszych kolekcji.

Elementy na liście mogą się powtarzać.

Listę tworzymy w taki sam sposób jak znane nam dotychczas obiekty.

```
List myList = new ArrayList();
```

- **`List`** – deklarujemy typ.

Klasa ArrayList

Klasa **`ArrayList`** implementuje interfejs **`List`**. Jest jedną z najpopularniejszych kolekcji.

Elementy na liście mogą się powtarzać.

Listę tworzymy w taki sam sposób jak znane nam dotychczas obiekty.

```
List myList = new ArrayList();
```

- **`List`** – deklarujemy typ.
- **`new ArrayList()`** – tworzymy nowy obiekt klasy **`ArrayList`**.

ArrayList – elementy

Na schemacie obrazującym hierarchię kolekcji widzimy, że lista implementuje interfejs **`List`**, a ten z kolei dziedziczy z **`Collection`**, dzięki czemu lista posiada określoną w nim metodę **`add`**.

```
List myList = new ArrayList();
myList.add("Element 1");
myList.add("Element 2");
```

Elementy pobieramy korzystając z metody **`get`**, podając w parametrze metody indeks pod jakim znajduje się nasz obiekt.

```
Object elementList = myList.get(0);
```

Jednakże otrzymujemy **`Object`**, a przecież dodawaliśmy do listy elementy typu **`String`**. Natomiast jeśli chcemy uzyskać dodany wcześniej obiekt typu **`String`**, musimy dokonać rzutowania.

```
String elementList = (String) myList.get(0);
```

Typy parametryzowane

Aby uniknąć problemów związanych z rzutowaniem – stosujemy typy sparametryzowane (tzw. generyki).

Tworząc listę możemy jawnie zadeklarować jakiego typu obiekty będą w niej przechowywane.

Przykład

```
List<String> myList = new ArrayList<String>();
```

Typy parametryzowane

Aby uniknąć problemów związanych z rzutowaniem – stosujemy typy sparametryzowane (tzw. generyki).

Tworząc listę możemy jawnie zadeklarować jakiego typu obiekty będą w niej przechowywane.

Przykład

```
List<String> myList = new ArrayList<String>();
```

- **`List<String>`** – deklarujemy typ i określamy od razu jakie wartości będą przechowywane w naszej liście.

Typy parametryzowane

Aby uniknąć problemów związanych z rzutowaniem – stosujemy typy sparametryzowane (tzw. generyki).

Tworząc listę możemy jawnie zadeklarować jakiego typu obiekty będą w niej przechowywane.

Przykład

```
List<String> myList = new ArrayList<String>();
```

- **`List<String>`** – deklarujemy typ i określamy od razu jakie wartości będą przechowywane w naszej liście.
- **`new ArrayList<String>()`** – od wersji 1.7 Javy deklarację **`<String>`** przy tworzeniu obiektu możemy zastąpić pustymi nawiasami: **`<>`**

Typy parametryzowane

Pobierając element tak utworzonej listy nie musimy wykonywać rzutowania, co więcej, nasza lista nie przyjmie elementu innego typu niż zadeklarowany.

```
String elementList = myList.get(0);
```

Więcej o zastosowanej tutaj koncepcji typów generycznych możesz przeczytać tutaj:

https://docs.oracle.com/javase/tutorial/java/generics/

Iterator

Iterator to obiekt, którego zadaniem jest przemieszczanie się po wybranej kolekcji.

Możemy go pobrać od dowolnej kolekcji implementującej interfejs **Collection**.

Schematyczny zapis jest następujący:

```
Iterator<E> iterator = collection.iterator();
```

<E> – oznacza typ elementów znajdujących się w kolekcji.

Przykład

```
List<String> stringsList = new ArrayList<>();
Iterator<String> iterator = stringsList.iterator();
```

Dokumentacja: https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Iterator

Iterator zawiera poniższe metody:

- `boolean hasNext()` – zwraca wartość `true`, jeżeli kolekcja posiada następny element.
- `E next()` – zwraca następny element na liście i przesuwa iterator.
- `void remove()` – usuwa element kolekcji.

Iterator – przykład

Tworzymy listę, uzupełniamy ją wartościami, następnie pobieramy iterator od kolekcji:

```
List<Integer> arrayList = new ArrayList<>();
arrayList.add(5);
arrayList.add(4);
arrayList.add(3);
arrayList.add(new Integer(2));
arrayList.add(new Integer(1));

Iterator<Integer> it = arrayList.iterator();
```

Zwróć uwagę, że do listy dodajemy wartości typów prostych (prymitywnych) – znane już nam typy **`int`**.

Jak to możliwe skoro wcześniej powiedzieliśmy że kolekcje przechowują tylko referencje do obiektów? Spójrz na kolejny slajd.

Autoboxing

Dodanie do kolekcji wartości typu prostego było możliwe dzięki zastosowanej w Javie idei tzw. **autoboxingu**.

Polega ona na automatycznym przekształcaniu pomiędzy typami prostymi a typami obiektów klas, opakowujących dany typ prosty.

Przekształcenia mają miejsce między następującymi typami:

Typ prymitywny	Typ obiektowy
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`
`double`	`Double`

Autoboxing

W kontekście przekształceń mówimy o typach opakowujących typy proste.

Oprócz automatycznej konwersji możemy również wykonać ją w sposób ręczny (przekształcając typy proste na obiektowe i odwrotnie).

Możemy to zrobić wykorzystując metody: **`intValue`**, **`floatValue`** itd. lub stosując rzutowanie typów.

```
Integer integerObject = new Integer(12);
int intPrimitive = integerObject.intValue();
Integer newIntegerObject = new Integer(intPrimitive);
Integer castToInteger = (Integer) intPrimitive;
int castToInt = (int) integerObject;
```

Więcej o **autoboxingu** przeczytasz w dokumentacji:

https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html

Iterator – przykład

Wróćmy do utworzonego wcześniej, sparametryzowanego iteratora:

```
Iterator<Integer> iterator = arrayList.iterator();
```

1. Wypisywanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```

2. Usuwanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()) {
    if (iterator.next() == 4) {
        iterator.remove();
    }
}
```

Iterator – przykład

Wróćmy do utworzonego wcześniej, sparametryzowanego iteratora:

```
Iterator<Integer> iterator = arrayList.iterator();
```

1. Wypisywanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```

Dopóki kolekcja posiada elementy...

2. Usuwanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()) {
    if (iterator.next() == 4) {
        iterator.remove();
    }
}
```

Iterator – przykład

Wróćmy do utworzonego wcześniej, sparametryzowanego iteratora:

```
Iterator<Integer> iterator = arrayList.iterator();
```

1. Wypisywanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```

...wyświetl element listy.

2. Usuwanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()) {
    if (iterator.next() == 4) {
        iterator.remove();
    }
}
```

Iterator – przykład

Wróćmy do utworzonego wcześniej, sparametryzowanego iteratora:

```
Iterator<Integer> iterator = arrayList.iterator();
```

1. Wypisywanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```

2. Usuwanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()) {
    if (iterator.next() == 4) {
        iterator.remove();
    }
}
```

Dopóki kolekcja posiada elementy...

Iterator – przykład

Wróćmy do utworzonego wcześniej, sparametryzowanego iteratora:

```
Iterator<Integer> iterator = arrayList.iterator();
```

1. Wypisywanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```

2. Usuwanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()) {
    if (iterator.next() == 4) {
        iterator.remove();
    }
}
```

...sprawdź warunek...

Iterator – przykład

Wróćmy do utworzonego wcześniej, sparametryzowanego iteratora:

```
Iterator<Integer> iterator = arrayList.iterator();
```

1. Wypisywanie elementów kolekcji przy użyciu iteratora:

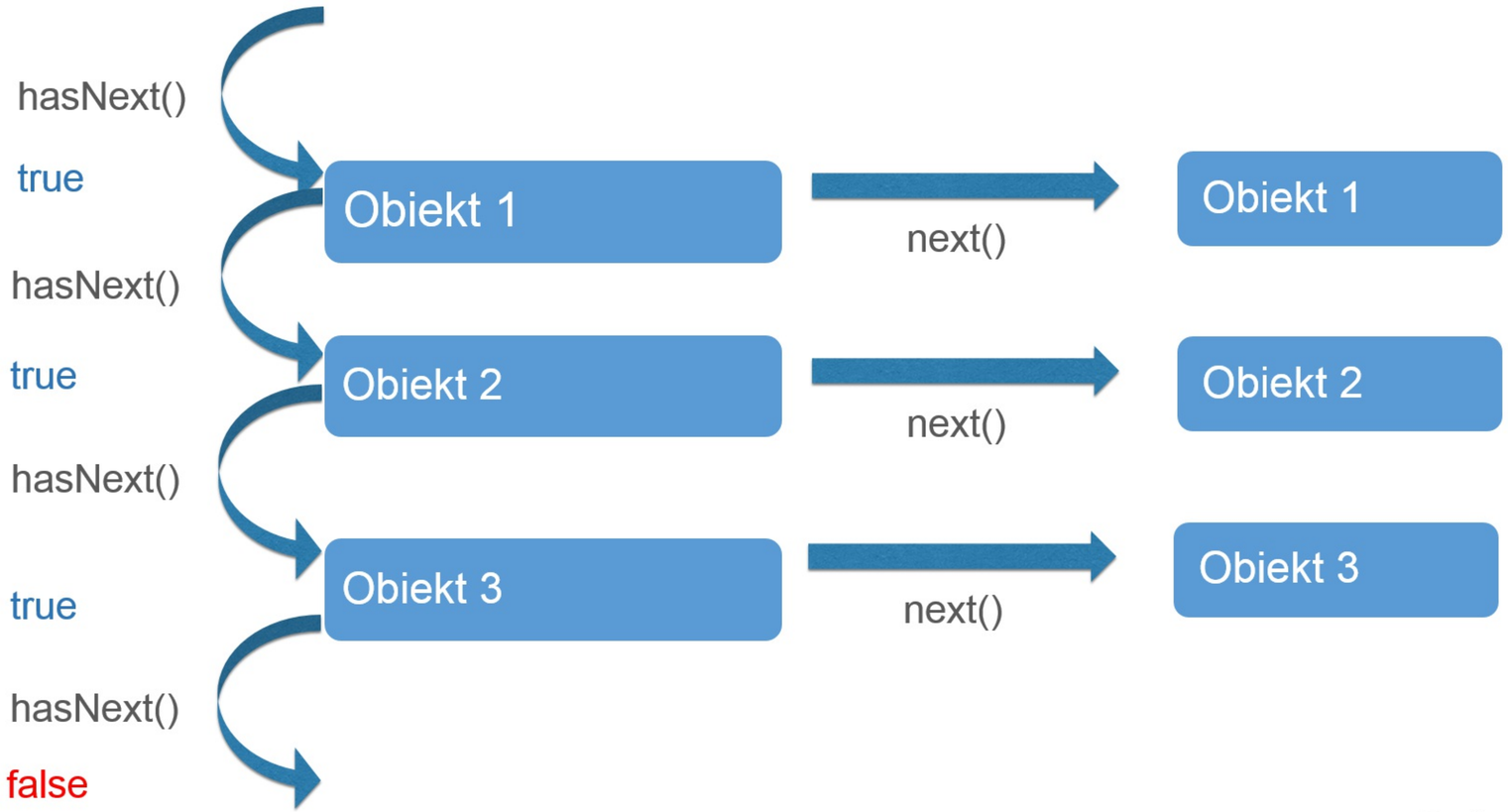
```
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```

2. Usuwanie elementów kolekcji przy użyciu iteratora:

```
while (iterator.hasNext()) {
    if (iterator.next() == 4) {
        iterator.remove();
    }
}
```

...wykonaj operację – w tym przypadku – usunięcia elementu.

Iterator – przykład



Iterator

Przykład

```
List<String> list = new ArrayList<>();
list.add("first");
list.add("second");
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String string = it.next();
    System.out.println(string);
}

while (it.hasNext()) {
    String string = it.next();
    System.out.println(string);
}
System.out.println("---------");
```

Pamiętaj, że **iteratora** można użyć tylko raz. Nie posiada on metod które pozwalały by go resetować.

Mimo dwukrotnego wywołania, jako wynik uruchomienia przykładowego kodu otrzymamy:

```
first
second
---------
```

Iterator

Przykład

```
List<String> list = new ArrayList<>();
list.add("first");
list.add("second");
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String string = it.next();
    System.out.println(string);
}

it = list.iterator();//ponowne pobranie
while (it.hasNext()) {
    String string = it.next();
    System.out.println(string);
}
System.out.println("---------");
```

Aby ponownie przejść przez elementy listy wykorzystując **iterator**, należy go ponownie pobrać za pomocą wywołania:

```
it = list.iterator();
```

Teraz jako wynik uruchomienia przykładowego kodu otrzymamy:

```
first
second
first
second
---------
```

ListIterator

Dla list dostępny jest specjalny typ iteratora – **ListIterator**, zawiera on dodatkowe metody:

- **`boolean hasPrevious()`** – zwraca wartość **`true`**, jeżeli kolekcja posiada wcześniejszy element.
- **`Object previous()`** – zwraca poprzedni element na liście i przesuwa iterator.
- **`void add(T o)`** – dodaje element do listy.
- **`void set(T o)`** – zmienia ostatni element zwrócony za pomocą metody **`next()`** lub **`previous()`**.
- **`int nextIndex()`** – zwraca indeks elementu, który byłby zwrócony przez wywołanie metody **`next()`**.
- **`int previousIndex()`** – zwraca indeks elementu, który byłby zwrócony przez wywołanie metody **`previous()`** .

ListIterator – przykład

ListIterator pobieramy od kolekcji za pomocą metody **listIterator()**:

```
List<String> list = new ArrayList<>();
list.add("first");
list.add("second");
ListIterator<String> listIterator = list.listIterator();

while (listIterator.hasNext()) {
        String string = listIterator.next();
        System.out.println(string);
}
while (listIterator.hasPrevious()) {
        String string = listIterator.previous();
        System.out.println(string);
}
```

Jak widać na przykładzie, za pomocą tego iteratora możemy przechodzić po elementach kolekcji również od końca do początku.

Pętla for w liście

Do przejścia po elementach listy możemy używać znanej już nam pętli **`for`**. Tworzymy pętlę w następujący sposób:

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

Jeśli używamy **iteratora** do przechodzenia po kolekcji w pętli, to nie wykorzystujemy indeksu. Pętla **`for`** ma wtedy postać:

```
Iterator<Integer> iterator = arrayList.iterator();
for (iterator; iterator.hasNext(); ) {
    System.out.println(iterator.next());
}
```

Pętla for w liście

Do przejścia po elementach listy możemy używać znanej już nam pętli **`for`**. Tworzymy pętlę w następujący sposób:

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

Jeśli używamy **iteratora** do przechodzenia po kolekcji w pętli, to nie wykorzystujemy indeksu. Pętla **`for`** ma wtedy postać:

```
Iterator<Integer> iterator = arrayList.iterator();
for (iterator; iterator.hasNext(); ) {
    System.out.println(iterator.next());
}
```

- Zwróć uwagę że pętla **`for`** ma tylko dwa parametry.

Konstrukcja for-each

Do przejścia po elementach kolekcji możemy wykorzystać również poniższą konstrukcję:

```
for (Object o : collection){
    System.out.println(o);
}
```

Przykład:

```
for (Integer value : arrayList){
    System.out.println(value);
}
```

Jeżeli chcemy usunąć określone elementy z kolekcji, musimy w tym celu użyć **`iteratora`**.

Próba usunięcia elementu w pętli **for-each** spowoduje wystąpienie wyjątku:

```
Exception in thread "main"
java.util.
ConcurrentModificationException
```

Zadania

HashSet

Klasa **`HashSet`** implementuje interfejs **`Set`**, czyli zbiór. Ten interfejs nie posiada dodatkowych metod względem interfejsu **`Collection`**.

Elementy w tej kolekcji nie mogą się powtarzać.

```
Set<String> myHashSet = new HashSet<>();
myHashSet.add("circle");
myHashSet.add("square");
myHashSet.add("circle");

System.out.println(myHashSet.size());
```

HashSet

Klasa **`HashSet`** implementuje interfejs **`Set`**, czyli zbiór. Ten interfejs nie posiada dodatkowych metod względem interfejsu **`Collection`**.

Elementy w tej kolekcji nie mogą się powtarzać.

```
Set<String> myHashSet = new HashSet<>();
myHashSet.add("circle");
myHashSet.add("square");
myHashSet.add("circle");

System.out.println(myHashSet.size());
```

- Dodajemy elementy do zbioru.

HashSet

Klasa **`HashSet`** implementuje interfejs **`Set`**, czyli zbiór. Ten interfejs nie posiada dodatkowych metod względem interfejsu **`Collection`**.

Elementy w tej kolekcji nie mogą się powtarzać.

```
Set<String> myHashSet = new HashSet<>();
myHashSet.add("circle");
myHashSet.add("square");
myHashSet.add("circle");

System.out.println(myHashSet.size());
```

- Dodajemy elementy do zbioru.
- Zwróci 2, ponieważ element **`"circle"`** istniał już w zbiorze i nie został dodany po raz drugi.

Interfejs Map

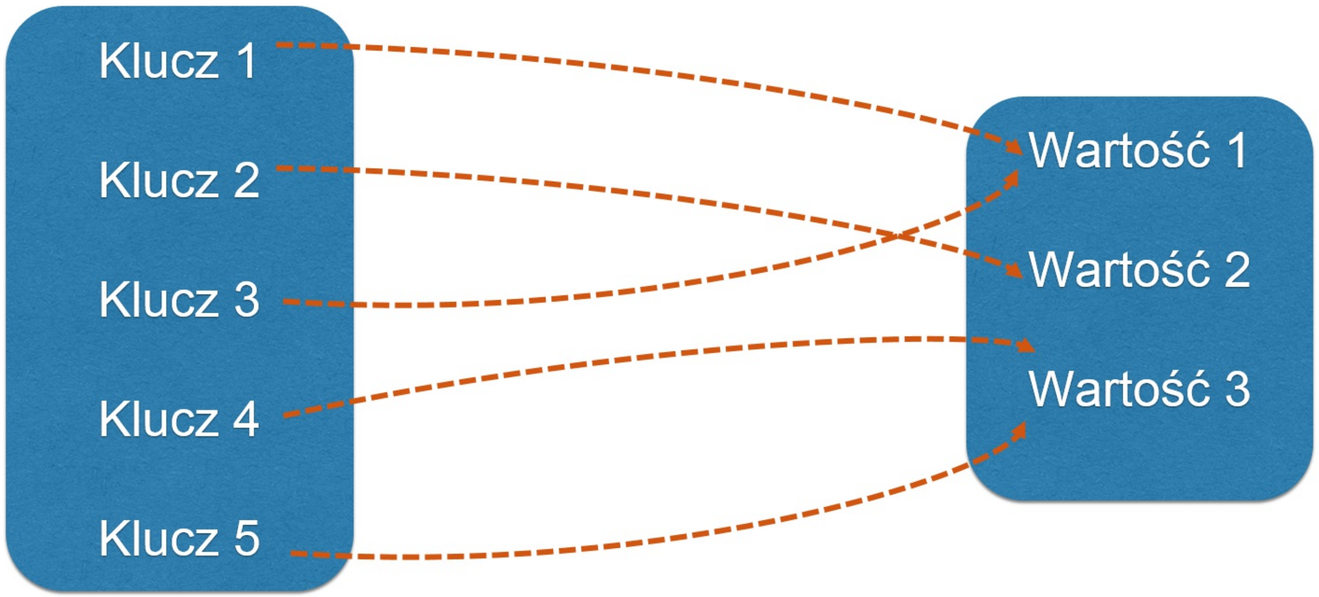
Mapy przechowują zestawy par obiektów klucz–wartość.

Posiadają następujące metody:

- `boolean containsKey(Object key)` – zwraca `true`, gdy dany klucz istnieje.
- `boolean containsValue(Object value)` – zwraca `true`, gdy dana wartość istnieje.
- `V get(Object key)` – zwraca wartość z określonego klucza.
- `Set<K> keySet()` – zwraca zbiór kluczy.
- `V put(K key, V value)` – wstawia pod konkretny klucz żądaną wartość.
- `V remove(Object key)` – usuwa element oraz zwraca wartość znajdującą się pod określonym kluczem.

Mapa

Wizualnie mapę moglibyśmy przedstawić następująco:



Zwróć uwagę, że kilka kluczy może wskazywać na tę samą wartość, natomiast klucze muszą być **unikalne**.

HashMap

Znając klucz, możemy pobrać wartość.

Przykład

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Black");
map.put(2, "Black");
map.put(3, "Black");
map.put(3, "White");

String mapValue = map.get(1);
System.out.println(map);
```

HashMap

Znając klucz, możemy pobrać wartość.

Przykład

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Black");
map.put(2, "Black");
map.put(3, "Black");
map.put(3, "White");

String mapValue = map.get(1);
System.out.println(map);
```

- Dodajemy elementy do mapy.

HashMap

Znając klucz, możemy pobrać wartość.

Przykład

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Black");
map.put(2, "Black");
map.put(3, "Black");
map.put(3, "White");

String mapValue = map.get(1);
System.out.println(map);
```

- Dodajemy elementy do mapy.
- Klucz o wartości "**3**" już istnieje, więc zostanie nadpisana wartość pod kluczem **3**.

HashMap

Znając klucz, możemy pobrać wartość.

Przykład

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Black");
map.put(2, "Black");
map.put(3, "Black");
map.put(3, "White");

String mapValue = map.get(1);
System.out.println(map);
```

- Dodajemy elementy do mapy.
- Klucz o wartości "**3**" już istnieje, więc zostanie nadpisana wartość pod kluczem **3**.
- Pobierze wartość, która znajduje się pod kluczem **1**.

HashMap

Znając klucz, możemy pobrać wartość.

Przykład

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Black");
map.put(2, "Black");
map.put(3, "Black");
map.put(3, "White");

String mapValue = map.get(1);
System.out.println(map);
```

- Dodajemy elementy do mapy.
- Klucz o wartości "**3**" już istnieje, więc zostanie nadpisana wartość pod kluczem **3**.
- Pobierze wartość, która znajduje się pod kluczem **1**.
- Wyświetli: **`{1=Black, 2=Black, 3=White}`**

Zadania

Wyrażenia regularne

Wyrażenia regularne – definicja

- Wyrażenia regularne (regex) są to wzorce opisujące łańcuchy symboli.
- W informatyce teoretycznej to ciągi znaków, pozwalające opisywać języki regularne.
- W praktyce znalazły bardzo szerokie zastosowanie, pozwalają bowiem w łatwy sposób opisywać wzorce tekstu.

- Dwie najpopularniejsze składnie wyrażeń regularnych to **składnia uniksowa** i **składnia perlowa**.
- W większości zastosowań stosuje się **składnię perlową**.

Wyrażenia regularne – podstawowe zasady

- Każdy znak (oprócz znaków specjalnych) określa sam siebie, np. **a** oznacza po prostu znak **a**.
- Zapis symboli oznacza, że w łańcuchu muszą wystąpić dokładnie takie symbole i w dokładnie takiej samej kolejności

Przykład

Zapis: **ab** oznacza, że łańcuch musi składać się ze znaku **a** poprzedzającego znak **b**.

Znaki specjalne

ZNAK	ZNACZENIE
`.`	Dowolny znak z wyjątkiem znaku nowego wiersza.
`[]`	Jeden dowolny znak ze znaków znajdujących się między nawiasami. Przykład: **`[abc]`** – oznacza **`a`**, **`b`** lub **`c`**.
`[a-c]`	Jeden znak z przedziału od **`a`** do **`c`**
`[^...]`	Jeden dowolny znak nieznajdujący się między nawiasami. Przykład: **`[^abc]`** – oznacza jeden znak z wyjątkiem **`a`**, **`b`** i **`c`**.
`()`	To grupa symboli do późniejszego wykorzystania (przechwytywanie).
`|`	To odpowiednik słowa lub. Oznacza wystąpienie jednego z podanych wyrażeń. Przykład: **`a|b|c`** oznacza **`a`** lub **`b`** lub **`c`**.
`^`	Oznacza początek wiersza.
`$`	Oznacza koniec wiersza.

Znaki specjalne c.d.

ZNAK	ZNACZENIE
*	To zero lub więcej wystąpień poprzedzającego wyrażenia. Przykład: **[abc]*** – oznacza zero lub więcej znaków ze zbioru **a**, **b**, **c**.
+	To jedno lub więcej wystąpień poprzedzającego wyrażenia.
?	To najwyżej jedno wystąpienie (może być zero) poprzedzającego wyrażenia.

Znaki specjalne jako zwykłe znaki:

Jeżeli chcemy, aby znak specjalny został potraktowany jako zwykły, to musimy go poprzedzić ukośnikiem wstecznym "\".

Na przykład:

- `\.` oznacza kropkę, a nie dowolny znak.
- `\.+` oznacza jedną lub więcek kropek.

Rozszerzenia

KOD	ZNACZENIE
`\d`	Dowolna cyfra
`\D`	Dowolny znak niebędący cyfrą
`\s`	Dowolny znak biały (np. spacja, tabulator)
`\S`	Dowolny znak niebędący znakiem białym
`\w`	Dowolny znak typu: litera, cyfra, podkreślnik
`\W`	Dowolny znak inny niż: litera, cyfra, podkreślnik
`{N}`	Dokładnie N wystąpień
`{N,}`	Co najmniej N wystąpień
`{N,M}`	Od N do M wystąpień

Odwołania i przechwytywanie

Odwołania wsteczne

Umieszczając grupę symboli w nawiasach okrągłych "`()`", umożliwiamy jej przechwycenie i wykorzystanie później w wyrażeniu – przy pomocy odwołania wstecznego.

Do kolejnych fragmentów odwołujemy się poprzez `\1 \2 \3` itd.

Przykład

`(\w+)=\1` pasuje do napisu: `Ala=Ala`

Wyrażenie regularne `(\w+)=\1` sprawdza, czy przed znakiem równości w napisie znajduje się słowo, określone wyrażeniem: `\w+` – przechwytuje je, a następnie weryfikuje, czy za znakiem równości jest to samo słowo – `\1`.

Przykłady prostych wyrażeń regularnych

- Cztery cyfry:
  `\d{4}`
- Mała lub duża litera użyta przynajmniej raz:
  `[A-Za-z]+`
- Litera z, a następnie zero lub więcej xy:
  `z(xy)*`
- Od 2 do 3 wystąpień napisu java:
  `(java){2,3}`
- Dowolny symbol użyty co najmniej raz:
  `.+`
- Dowolny symbol użyty zero lub więcej razy:
  `.*`

Przydatne wzory wyrażeń regularnych

Wartość HEX:

```
#?([a-f0-9]{6}|[a-f0-9]{3})
```

Adres IP:

```
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
```

Tag HTML:

```
<([a-z]+)[^<>\/]*(?:>(.*)<\/\1>|\/>)
```

Kod pocztowy:

```
[0-9][0-9]-[0-9][0-9][0-9]
```

Adres email:

```
[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@[a-zA-Z0-9-]+(\.[a-zA-Z0-9-]+)*\.([a-zA-Z]{2,}){1}
```

Godzina w formacie 24h:

```
([01]?[0-9]|2[0-3]):[0-5][0-9]
```

Uwaga

W Javie wyrażenia regularne piszemy wykorzystując obiekty typu **`String`** (ciągi znaków).

Ponieważ **backslash** (\) jest używany w ciągach znaków np. **`\n`** wpisując wyrażenie regularne dodajemy dodatkowy znak `\`.

Tzn. zamiast wpisać `\.` wpiszemy `\\.` - jako wskazanie że w naszym wyrażeniu ma wystąpić znak kropki `.`

```
String pat1 = "abc\\.";
String pat2 = "^\\d{2}-\\d{3}$";
```

Przykłady wyrażeń podawane w prezentacji oprócz przykładów kodu Javy są podawane w notacji ogólnej, tzn. bez dodatkowego znaku `\.`

Testowanie wyrażeń regularnych

Do testowania wyrażeń regularnych możemy użyć stworzonych w tym celu serwisów, np.:

- https://regex101.com/
- http://www.freeformatter.com/regex-tester.html

Przydatne materiały o regex znajdziesz na stronach:

- http://docs.oracle.com/javase/tutorial/essential/regex/
- https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Istnieją również wtyczki do przeglądarek, które udostępniają taką funkcjonalność

- Chrome: https://chrome.google.com/webstore/detail/regexp-tester/fekbbmalpajhfifodaakkfeodkpigjbk
- Firefox: https://addons.mozilla.org/pl/firefox/addon/rext/

Wyrażenia regularne w Javie

Do obsługi wyrażeń regularnych mamy dwie klasy z pakietu **`java.util.regex`**:

- **`Pattern`**
- **`Matcher`**

Klasa **`Pattern`** – reprezentuje wyrażenie regularne.

Klasa **`Matcher`** – reprezentuje dopasowanie do wyrażenia.

Warto zapoznać się z ich dokumentacją:

https://docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html

Przykład wykorzystania

```
Pattern compiledPattern = Pattern.compile("reg.*");
Matcher matcher = compiledPattern.matcher("Wyrażenia regularne w Javie");
System.out.println(matcher.find());
System.out.println(matcher.matches());
```

Przykład wykorzystania

```
Pattern compiledPattern = Pattern.compile("reg.*");
Matcher matcher = compiledPattern.matcher("Wyrażenia regularne w Javie");
System.out.println(matcher.find());
System.out.println(matcher.matches());
```

Wyrażenie opisuje ciąg znaków zaczynający się od **`reg`**, po którym występuje **`.*`** – dowolny symbol użyty zero lub więcej razy

Przykład wykorzystania

```
Pattern compiledPattern = Pattern.compile("reg.*");
Matcher matcher = compiledPattern.matcher("Wyrażenia regularne w Javie");
System.out.println(matcher.find());
System.out.println(matcher.matches());
```

Wyrażenie opisuje ciąg znaków zaczynający się od **`reg`**, po którym występuje **`.*`** – dowolny symbol użyty zero lub więcej razy

Metoda **`compile(String regex)`** – przetwarza wyrażenie regularne, do dalszego wykorzystania.

Przykład wykorzystania

```
Pattern compiledPattern = Pattern.compile("reg.*");
Matcher matcher = compiledPattern.matcher("Wyrażenia regularne w Javie");
System.out.println(matcher.find());
System.out.println(matcher.matches());
```

Wyrażenie opisuje ciąg znaków zaczynający się od **`reg`**, po którym występuje **`.*`** – dowolny symbol użyty zero lub więcej razy

Metoda **`compile(String regex)`** – przetwarza wyrażenie regularne, do dalszego wykorzystania.

Metoda **`find()`** zwraca **`true`** jeśli w napisie znajdzie dopasowanie do wyrażenia regularnego.

Przykład wykorzystania

```
Pattern compiledPattern = Pattern.compile("reg.*");
Matcher matcher = compiledPattern.matcher("Wyrażenia regularne w Javie");
System.out.println(matcher.find());
System.out.println(matcher.matches());
```

Wyrażenie opisuje ciąg znaków zaczynający się od **`reg`**, po którym występuje **`.*`** – dowolny symbol użyty zero lub więcej razy

Metoda **`compile(String regex)`** – przetwarza wyrażenie regularne, do dalszego wykorzystania.

Metoda **`find()`** zwraca **`true`** jeśli w napisie znajdzie dopasowanie do wyrażenia regularnego.

Metoda **`matches()`** zwraca **`true`** jeśli napis pasuje w całości do wyrażenia regularnego.

Przykład wykorzystania

```
Pattern pattern = Pattern.compile("java", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(
  "Java jest technologią wykorzystywaną do\n" +
  "tworzenia aplikacji, które czynią Internet bardziej atrakcyjnym.\n" +
  "Java to nie to samo co JavaScript. Więcej informacji o oprogramowaniu Java");

while (matcher.find()) {
    System.out.print("start: " + matcher.start());
    System.out.println(" end: " + matcher.end() + " ");
}
```

Przykład wykorzystania

```
Pattern pattern = Pattern.compile("java", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(
  "Java jest technologią wykorzystywaną do\n" +
  "tworzenia aplikacji, które czynią Internet bardziej atrakcyjnym.\n" +
  "Java to nie to samo co JavaScript. Więcej informacji o oprogramowaniu Java");

while (matcher.find()) {
    System.out.print("start: " + matcher.start());
    System.out.println(" end: " + matcher.end() + " ");
}
```

Definiujemy wyrażenie regularne, dodatkowo określając, że dopasowania mają być niezależne od wielkości znaków.

Przykład wykorzystania

```
Pattern pattern = Pattern.compile("java", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(
  "Java jest technologią wykorzystywaną do\n" +
  "tworzenia aplikacji, które czynią Internet bardziej atrakcyjnym.\n" +
  "Java to nie to samo co JavaScript. Więcej informacji o oprogramowaniu Java");

while (matcher.find()) {
    System.out.print("start: " + matcher.start());
    System.out.println(" end: " + matcher.end() + " ");
}
```

Definiujemy wyrażenie regularne, dodatkowo określając, że dopasowania mają być niezależne od wielkości znaków.

Dopóki znajdujemy dopasowania...

Przykład wykorzystania

```
Pattern pattern = Pattern.compile("java", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(
  "Java jest technologią wykorzystywaną do\n" +
  "tworzenia aplikacji, które czynią Internet bardziej atrakcyjnym.\n" +
  "Java to nie to samo co JavaScript. Więcej informacji o oprogramowaniu Java");

while (matcher.find()) {
    System.out.print("start: " + matcher.start());
    System.out.println(" end: " + matcher.end() + " ");
}
```

Definiujemy wyrażenie regularne, dodatkowo określając, że dopasowania mają być niezależne od wielkości znaków.

Dopóki znajdujemy dopasowania...

...wyświetlamy na konsoli index początku i końca dopasowania w ciągu znaków.

Przykład wykorzystania

Jako wynik wykonania otrzymamy na konsoli:

```
start: 0 end: 4
start: 118 end: 122
start: 141 end: 145
start: 188 end: 192
```

Przykład zastosowania znaków: "^" i "$"

- Jeżeli zmodyfikujemy poprzednie wyrażenie, dodając na jego początku znak "**^**", np.:

```
Pattern pattern = Pattern.compile("^java", Pattern.CASE_INSENSITIVE);
```

Oznacza to, że dopasowanie musi rozpoczynać się od **java**.

Jako wynik wywołania wcześniejszego kodu otrzymamy:

```
start: 0 end: 4
```

- Jeżeli zmodyfikujemy wyrażenie dodając na jego końcu znak "**$**", np.:

```
Pattern pattern = Pattern.compile("java$", Pattern.CASE_INSENSITIVE);
```

Dopasowanie musi kończyć się na **java**.

Jako wynik wywołania wcześniejszego kodu otrzymamy:

```
start: 188 end: 192
```

Wyrażenia regularne w Javie

- Możemy również skorzystać z uproszczonej formy:

```
boolean regex = Pattern.matches("[0-9]", "1");
```

Forma ta jest zalecana przy pojedynczych sprawdzeniach wyrażenia.

- Również klasa **`String`** posiada metodę **`matches(String regex)`** – za jej pomocą możemy sprawdzić dopasowanie do wzorca np:

```
String checkDigits = "112233";
System.out.println(checkDigits.matches("\\d+"));  //jedna lub więcej cyfr
```

  - Metoda klasy **`String`** wywołuje metodę klasy **`Pattern`**:

```
public boolean matches(String regex) {
    return Pattern.matches(regex, this);
}
```

Metody klasy String wykorzystujące regex

Warto również wspomnieć o poznanych wcześniej metodach klasy **`String`**, które przyjmują jako parametr wyrażenie regularne np.:

- **`replaceAll()`**:

```
String textToReplace = "Kurs Euro to 4.12, a dolara to 3.33";
String replacedText = textToReplace.replaceAll("[0-9]", "X");
System.out.println(replacedText);
```

Jako wynik otrzymamy:

`Kurs Euro to X.XX a dolara to X.XX`

- **`split()`**:

```
String[] splitedText = textToReplace.split("[0-9]\\.[0-9]*");
System.out.println(Arrays.toString(splitedText));
```

Jako wynik otrzymamy:

`[Kurs Euro to , a dolara to ]`

Zadania