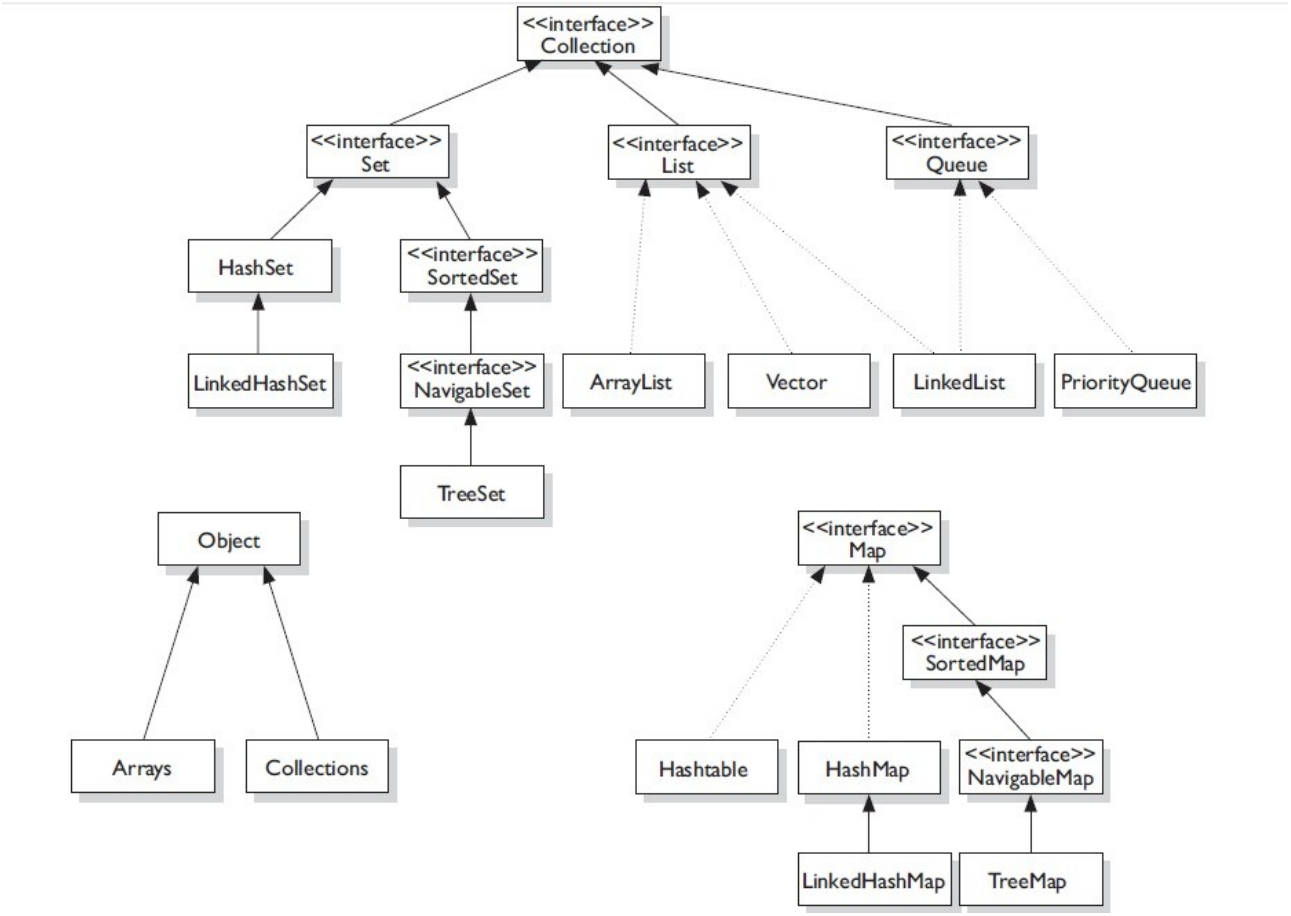
Kolekcje

Wstęp

Struktury danych to jest coś na czym opiera się logikę wszystkich programów. Nie da się optymalnie napisać programu korzystając jedynie z tablic. W każdym języku istnieją struktury danych, które charakteryzują się pewnymi cechami przydatnymi do pewnej klasy problemów programistycznych. Są na tyle ważne, że warto na chwilę się zatrzymać i jeszcze raz spojrzeć na ten temat dokładniej. Dlatego też spróbujmy zgłębić wiedzę o kolekcjach po raz kolejny, z większą uwagą.
Zanim jednak zaczniemy o nich rozmawiać zobaczmy podstawowy wykres pokazujący zależności między nimi:



Jak widzimy - wyróżniamy podstawowe typy danych: **Listy (List), Zbiory (Set), Kolejki (Queue), Mapy (Map)**. Każda ma zastosowanie do konkretnego problemu a sprawny programista automatycznie potrafi dopasować odpowiednią strukturę do aktualnego problemu z którym się spotkał.

Skoro 3 z nich wywodzą się z interfejsu Collection to przyjrzyjmy się jakie są tam dostępne metody:

- boolean add(T obj) - dodaje element do kolekcji
- boolean remove(T obj) - usuwa pierwsze wystąpienie podanego obiektu z kolekcji
- boolean remove(int index) - usuwa z kolekcji element o wskazanym indeksie
- boolean contains(T obj) - sprawdza czy dany obiekt wystepuje już w kolekcji

- int size() - odpowiednik własności length w przypadku tablic - zwraca rozmiar kolekcji
- boolean isEmpty() - zwraca true jeśli kolekcja jest pusta
- void clear() - czyści kolekcję
- Iterator iterator() - zwraca tzw. Iterator - o tym niżej.
- inne: addAll(Collection coll), removeAll(Collection coll), containsAll(Collection coll), retainAll(Collection coll)

Wszystkie omawiane dalej struktury danych (oprócz Mapy) implementują ten interfejs więc na pewno posiadają te metody. Nie będę więc do nich wracał.

Listy

Jest to podstawowa struktura danych. Możemy sobie ją wyobrazić jako zwykła tablica jaką znamy (np. String[]) tyle, że nie musimy się martwić o rozmiar tablicy i zestaw wielu pożytecznych metod. Interfejs ten nie decyduje o możliwości przechowywania duplikatów, ale wszystkie implementacje znajdujące się w JDK pozwalają na ich istnienie. List to kolekcja w której liczy się kolejność. Podobnie jak w tablicy każdy element ma w liście swoje miejsce, ma przed sobą i za sobą konkretny element wynikający z kolejności w jakiej zostały one do tej listy wstawione.

Charakterystyczne operacje w List (nie występujące w Collection):

- int indexOf(T obiekt) - zwraca indeks pierwszego wystąpienia elementu obj
- int lastIndexOf(T obiekt) - zwraca indeks ostatniego takiego elementu
- T set(int indeks, T obj) - wstawia w miejsce “indeks” element obj i zwraca obiekt zastąpiony
- ListIterator listIterator() - zwraca ListIterator - o tym niżej
- List subList(int poczatek, int koniec) - zwraca listę, która zawiera elementy od poczatek do koniec

Powszechnie znane są dwie implementacje interfejsu List:

```
List<String> listaString = new ArrayList<>(); // arraylist
List<String> listaString2 = new LinkedList<>(); // linkedlist
```

każda z nich różni się sposobem zaimplementowania funkcjonalności jaką wymusił na nich interfejs List. W większości klasycznych zastosowań korzysta się raczej z ArrayList - jednakże są sytuacje kiedy to lepiej zrobić to nieco inaczej.

```
List<Person> people = new ArrayList<>();
people.add(new Person("Arek", "Nowak", 25)); // Arek Nowak, lat 25
people.add(new Person("Natalia", "Kowalska", 33));
people.add(new Person("Łukasz", "Komorowski", 63));
people.add(new Person("Lucjan", "Janosz", 44));

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

// albo prościej:
for(Person person: people){
    System.out.println(person);
}
```

Jak sprawić, aby w konsoli pojawił się ładny tekst przedstawiajacy daną osobę przy wydruku **System.out.println(person)**;. Domyślnie każdy obiekt, przedstawia się dość mało ciekawie (drukuje się nazwa klasy a potem jakieś cyferki). Co za to odpowiada? Każda klasa dziedziczy po **Object** a tam jest kilka zaimplementowanych metod, które są znamienne. Jedną z nich jest metoda **toString()**. Jeśli tej metody nie nadpiszemy - obiekty będą się przedstawiać - tak jak im implementacja w Object każe (czyli brzydko).

A co jeśli byśmy chcieli umieć posortować listę elementów? Nic bardziej trudnego. Wystarczy powiedzieć javie co to znaczy, że jakiś element jest większy od innego. Resztę zrobi sam algorytm sortowania. Za całą funkcjonalność odpowiada interfejs **Comparable** wystarczy, że dowolny obiekt go implementuje oraz my napiszemy jego implementację - i wtedy będziemy w domu. Posortujmy więc listę People wg. wieku:

```
public class People implements Comparable<People> {

   ... // pola, metody

   public int compareTo(People o) {
       if(this.age == o.age){
           return 0;
       } else if(this.age > o.age) {
           return 1;
       } else {
           return -1;
       }
   }

}
```

Co tu się wydarzyło? Powiedziałem w klasie, że potrafię porównywać się z elementami People (tj. Comparable<People>). Musiałem więc zaimplementować metodę, którą ten interfejs mi nakazał (tj. metoda compareTo). Jak mówi dokumentacja muszę zwrócić: 0 gdy elementy są równe, 1 jeśli element jest większy, -1 jeśli element jest mniejszy. Tak też zrobiłem. Od tej chwili List<People> mogę posortować za pomocą statycznej metody w klasie **narzędziowej Collections** dla interfejsu Collection:

```
Collections.sort(people);
```

Mogę zastosować sortowanie dla dowolnej listy obiektów. Wystarczy, że owy obiekt implementuje Comparable. To jest właśnie zaleta z abstrakcji!

A co jeśli bym chciał posortować tą listę malejąco? Wystarczy kosmetyczna zmiana (pomyśl jaka). Z resztą jak sprawny obserwator zauważy - klasa Integer ma zaimplementowaną metodę **compareTo** więc mogłem po prostu napisać:

```
public class People implements Comparable<People> {

   ... // pola, metody

   public int compareTo(People o) {
       return this.age.compareTo(o.age));
   }

}
```

Co jednak jeśli raz w kodzie chciałbym posortować po wieku a w innym miejscu chciałbym posortować po nazwisku? Nie mogę przecież nagle zmienić implementacji metody compareTo. Z pomocą

przychodzi nam interfejs o nazwie Comparator. Posiada on tylko jedną metodę: **int compare(T object1, T object2)** i nie trudno się domyśleć co powinno być w jego implementacji. Natomiast w Collections jest metoda statyczna: **public static <T> void sort(List<T> list, Comparator<? super T> c)** (po ostatnich zajęciach nawet wiemy co ten zapis oznacza!). Przykład sorotowania po nazwisku i przy okazji powtórzenie o klasach anonimowych:

```
Collections.sort(people, new Comparator<People>(){
    @Override
    public int compare(People a, People b){
       return a.getName().compareTo(b.getName);
    }
});
```

Zbiory

Zbiory (Set) są kolekcją (interfejs Collection), które mają tą właściwość, że pozwalają na przechowywanie elementów unikalnych tj. w zbiorze nie może być 2-ch takich samych elementów. Co to znaczy takich samych elementów? Nie jest to prosta odpowiedź gdyż jest kilka implementacji Set:

- HashSet - zbiór o implementacji na hash table. Nie zachowuje kolejności dodawanych elementów
- LinkedHashSet - zbiór taki jak HashSet ale zachowujący kolejność dodawanych elementów
- TreeSet - zbiór o implementacji na tzw. drzewie

HashSet - bardzo powszechnie używana implementacja. Jest oparta o tzw. hashe, które mają tą zaletę, że dają błyskawiczne czasy dodawania, sprawdzania, usuwania elementów ze zbioru. Niestety iterując się po elementach wcale nie jest powiedziane, że będziemy widzieć je wedle kolejności w jakiej do zbioru je dodawaliśmy. Tej wady nie ma **LinkedHashSet** choć owa "pamięć" kosztuje nieco na wydajności. Na jakiej podstawie Set określa czy posiada w swoim zbiorze jakiś element? Sprawdza każdy element jeden po drugim w Set i porównuje? Przecież byłoby to bez sensu i trwałoby wieki dla dużych zbiorów. A przecież napisałem, że czas dodawania jest natychmiastowy.

Magia ta jest wspierana przez funkcję skrótu (hash) oraz metodę equals. Zarówno **equals()** oraz **hashCode()** są metodami dziedziczonymi z **Object**. Wypada nadpisać te metody jeśli chcemy aby java prawidłowo rozpoznawała te same elementy i prawidłowo liczyła hashCode. Jest to tak powszechne, że Eclipse potrafi sam zaproponować poprawne nadpisanie tych metod (PPM → Source → Generate hashCode and equals).

Jak można wydrukować zawartość Set? Nie posiada on wszak metody **get(int index)**. Robi się to za pomocą tzw. Iteratorów. Już korzystaliśmy z nich wcześniej - ale nawet nie byliśmy tego świadomi.

```
Set<Person> people = new HashSet<>();
people.add(new Person("Arek", "Nowak", 25));
people.add(new Person("Natalia", "Kowalska", 33));

for(Person person: people){   // tutaj używany jest iterator
    System.out.println(person);
}
```

Każda kolekcja potrafi zwrócić Iterator. Interfejs Iterator posiada metody hasNext(), next(), remove(). Najłatwiej przedstawić na przykładzie:

```
Set<String> newset = new HashSet<>();

newset.add("Learning");
newset.add("Easy");
newset.add("Simply");

// tworzymy iterator
Iterator iterator = newset.iterator();

// iterujmy się
while (iterator.hasNext()){
   String text = iterator.next();
   System.out.println("Value: " + text);
}
iterator.remove(); // usuwam element nad którym jest iterator - w tym
przypadku ostatni
```

Świadomość czym jest hashCode oraz equals jest bardzo często sprawdzana w przypadku rozmów o pracę. Standardowe pytanie podczas rozmowy kwalifikacyjnej: “Jaka jest zależność między hashCode a equals?”. Odpowiedź jest następująca: Jeśli następuje zależność **equals == true** dla dwóch obiektów to **hashCode1 == hashCode2**. W drugą stronę ta zależność nie zachodzi tj. jeśli hashCode dwóch obiektów pokryje się - to wcale nie daje gwarancji, że obiekty są sobie równe wg. metody equals. Wszak hashCode może się pokryć przypadkiem.

TreeSet - jest to implementacja Set, która przy dodawaniu kolejnego elementu - sortuje tam powstały zbiór. Oznacza to, że zbiór ten zawsze jest posortowany wedle tego co mówi **Comparable** lub tak jak chce **Comparator**:

```
Set<String> newset = new TreeSet<>(); // sortowanie według Comparable w
String
Set<String> newSet2 = new TreeSet<>(new Comparator<String>(){ // sortowanie
według Comparator
// implementacja compare(String a, String b){.... }
});
```

Mapy

Mapy umożliwiają tworzenie par Klucz-Wartość. Kluczem i Wartością mogą być dowolne obiekty. Właściwość jest taka, że dla jednego klucza może istnieć tylko jedna wartość. Klucz jest elementem unikalnym. Może od razu od przykładu:

```
Map<String, Double> tm = new TreeMap<>();

// Osoby i ich dług
tm.put("Marysia", new Double(3434.34));
tm.put("Michal", new Double(123.22));
tm.put("Adrian", new Double(1378.00));
tm.put("Maks", new Double(99.22));

// zwróć zbiór wszystkich par klucz-wartość
Set<Map.Entry<String, Double>> set = tm.entrySet();

// iteratoror po zbiorze
Iterator<Map.Entry<String, Double>> i = set.iterator();
```

```java
// Display elements
while(i.hasNext()) {
    Map.Entry<String, Double> pair = i.next();
    System.out.println(pair.getKey() + ": " + pair.getValue());
}
System.out.println();

// Adrian pożycza kolejne 1000
double balance = tm.get("Adrian").doubleValue();
tm.put("Adrian", new Double(balance + 1000));
System.out.println("Adriana nowy dług: " + tm.get("Adrian"));
```

Klasa: Map.Entry<String, Double> wygląda nieco dziwnie ale jest to w zasadzie coś na styl Pair<String, Double> tyle, że jest to klasa statyczna wewnętrzna **Entry** w klasie **Map**.

Chodzi wśród programistów legenda, że w zasadzie nie są potrzebne inne struktury danych. Wszystko można opędzić Map'ami. Wszak można nawet tworzyć Mapę Map itp.

Jakie Map'y wyróżniamy? **HashMap** - która wkłada elementy do “szuflady” wskazane przez hashCode klucza. Jeśli się zdarzy, że pod tą szufladą jest już jakiś element - sprawdza czy owy element jest równy (equals) z tym, którego zamierza wstawić. Jeśli jes ten sam - nie pozwoli na dodanie do mapy, jeśli inny - powtórnie dokona hashCode i spróbuje znaleźć inne miejsce. Jako, że hashCode liczony jest błyskawicznie - taka mapa działa natychmiastowo. **LinkedHashMap** jest odmianą mapy, która zachowuje kolejność wstawianych elementów. **TreeMap** to Mapa która automatycznie sortuje po **Kluczu**.

Pobaw się Map'ami, zobacz jakie mają metody (values(), keySet(), …).