# Kolekcje

## Spis treści

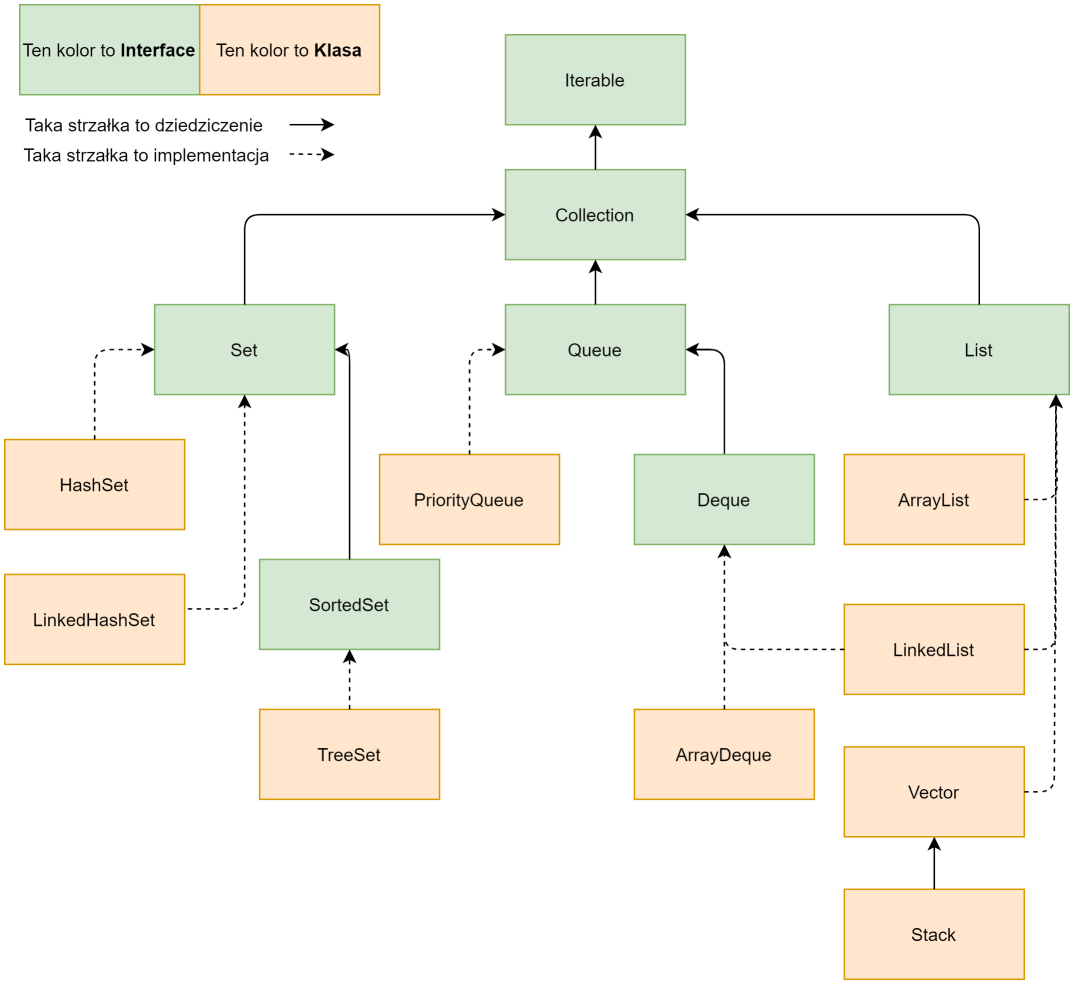


Zapiski prowadzącego Karola Rogowskiego i uczestnika Bootcampu Zajavka Bartek Borowczyk aka Samuraj Programowania.

## Collections (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`. Na ten moment poruszymy pierwsze 3, o kolejkach będziemy rozmawiać w przyszłości.

W taki sposób wygląda diagram hierarchii klas i interfejsów, które dziedziczą z interfejsu `Collection`, o których będziemy rozmawiać. Dodane są tu też klasy, o których będziemy rozmawiać w przyszłości.




*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 3 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

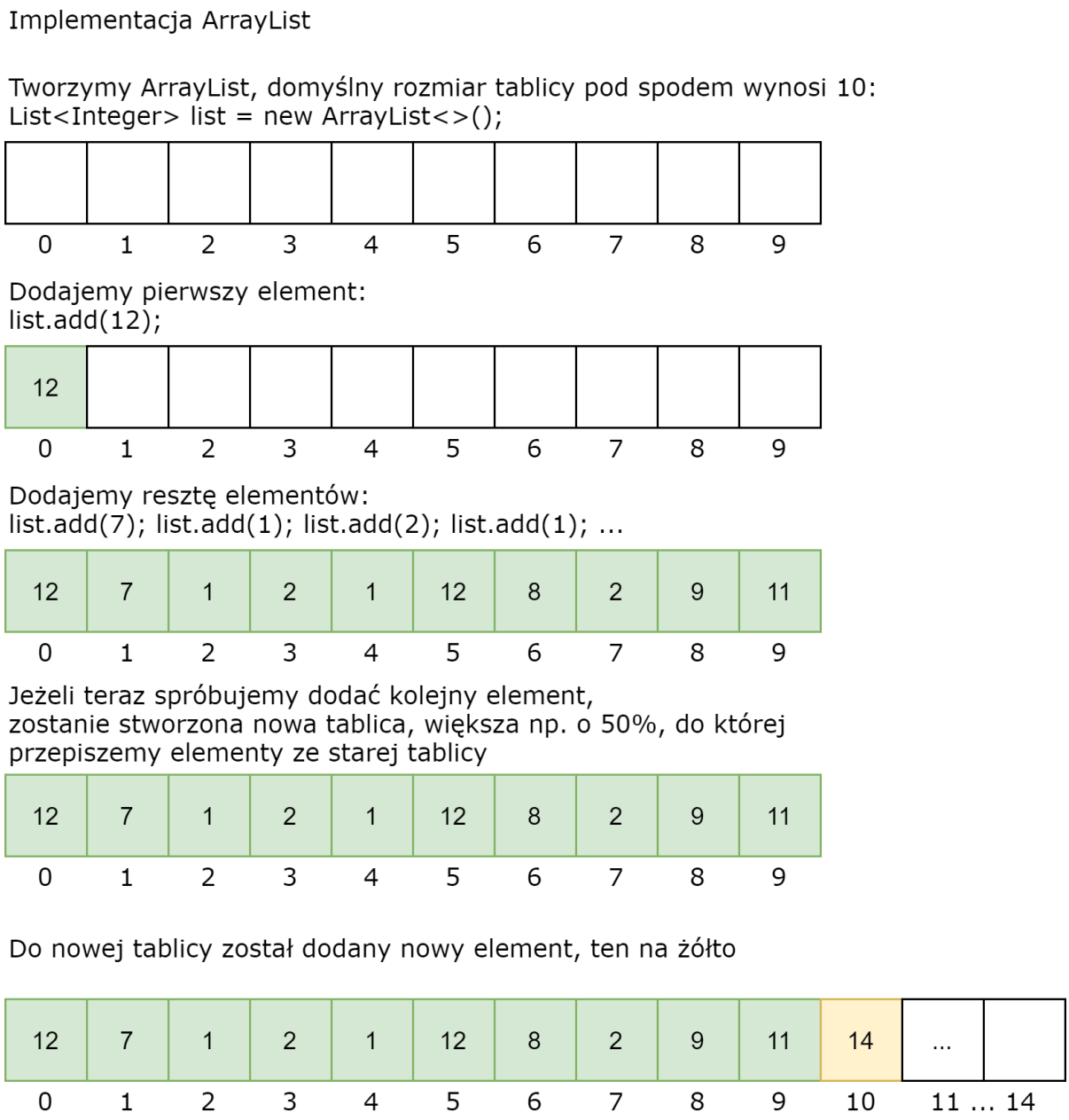
dodaniu elementów do `ArrayList` jest ona w stanie automatycznie zwiększać swój rozmiar.

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`.




*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%.

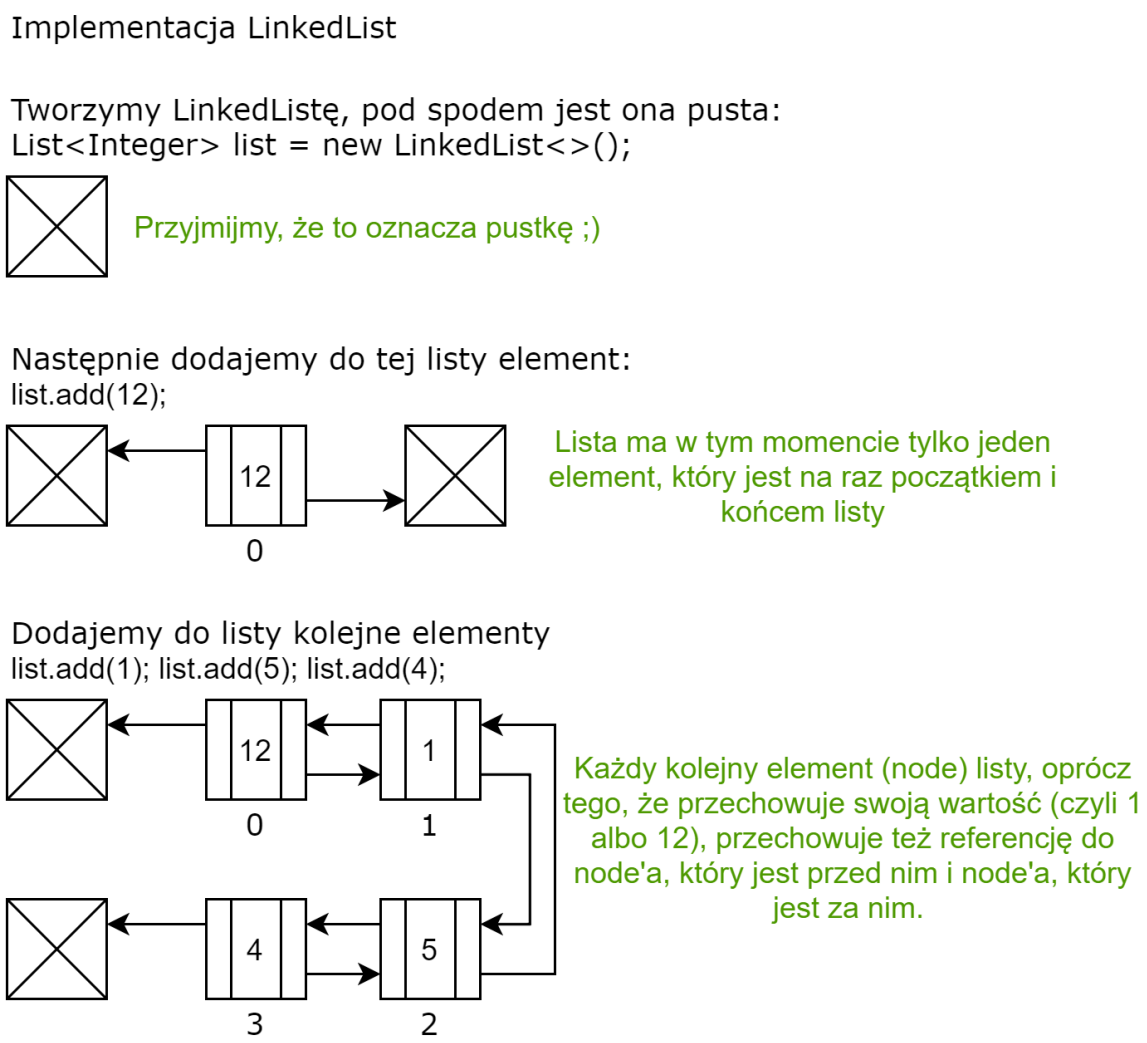
# LinkedList

LinkedList jest o tyle specyficzną implementacją, że implementuje jednocześnie interfejsy `List` i Queue. Skoro implementuje interface `List`, oznacza to, że ma dostępne wszystkie metody z `List`. Posiada również dodatkowe metody ułatwiające dodawanie lub usuwanie z początku i / lub końca listy, gdyż implementuje interface Queue. O kolejkach będziemy rozmawiać w przyszłości, więc na ten moment nie będę się skupiał na tych funkcjonalnościach.

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.




*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 namiar 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 faktura, 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, o której dowiemy się później.

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 Linked`HashSet`, my na ten moment poruszamy tylko `HashSet`.

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.

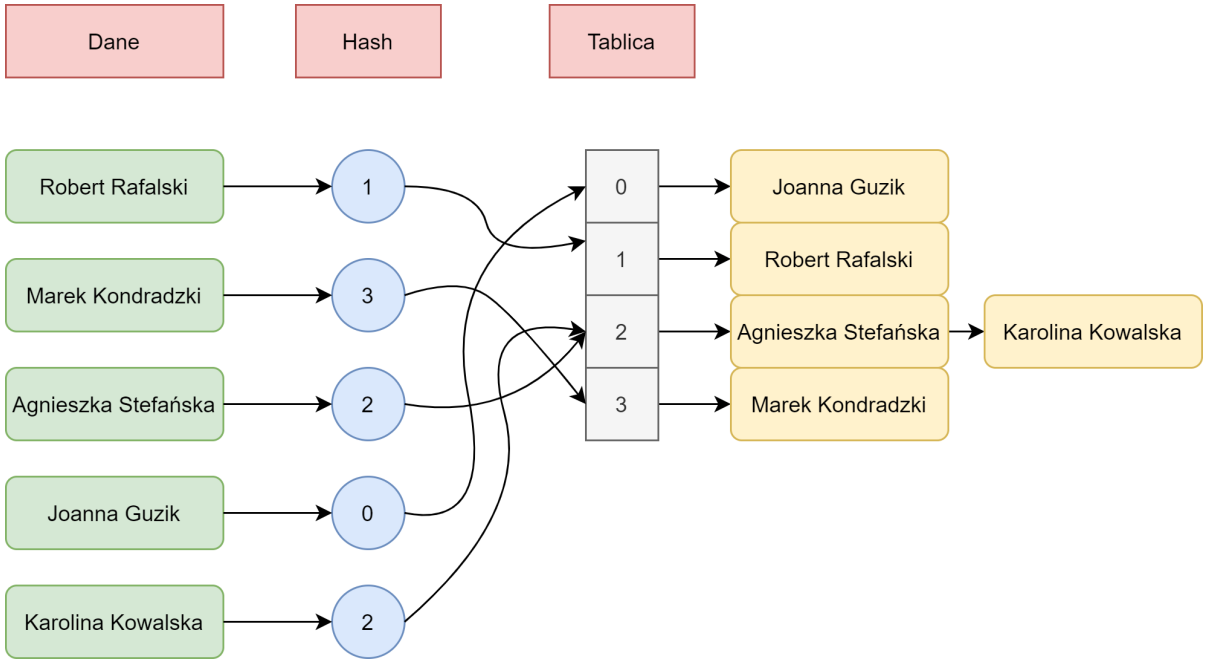
Wyobraź sobie, że masz przed sobą 1000 osób i potrzebujesz wśród nich znaleźć Karolinę Kowalską (patrz obrazek niżej). Ja wiem, że w życiu codziennym można krzyknąć "Hej, szukam Karoliny Kowalskiej" i mogłaby ona podnieść rękę, ale programując, to tak nie działa.

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 mam 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 byliby 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.

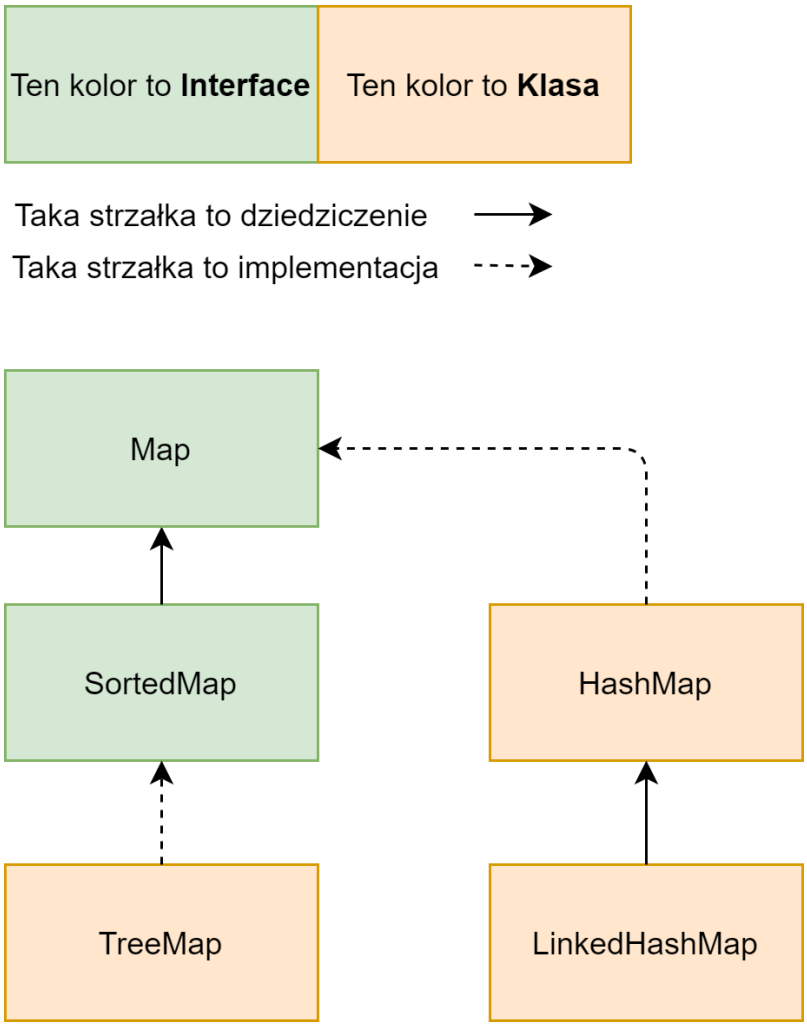
```
Set<String> setWithCities = new HashSet<>();
setWithCities.add("Warszawa");
setWithCities.add("Gdańsk");
setWithCities.add("Łódź");
setWithCities.add("Wrocław");
setWithCities.add("Warszawa");
setWithCities.add("Wrocław");

// 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"));
```

# Map




*Obraz 5. Hierarchia Map*

Map również jest interfejsem, który posiada różne implementacje, które różnią się pewnymi niuansami. Najczęściej używane to `HashMap`, `TreeMap` i `LinkedHashMap`, my na ten moment poruszamy tylko `HashMap`, oraz lekko wspomnimy o `Hashtable` (zwróć uwagę na różnicę w wielkiej literze 't' poprzednio dla struktury `HashTable` była ona wielką literą, 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 wielką literą, 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, które zostanie poruszone w przyszłości.

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", "XC80");
cars.put("Fiat", "Panda");
cars.put("Volkswagen", "Golf");

// Get key: XC80
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"));
```

HashMapa 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.

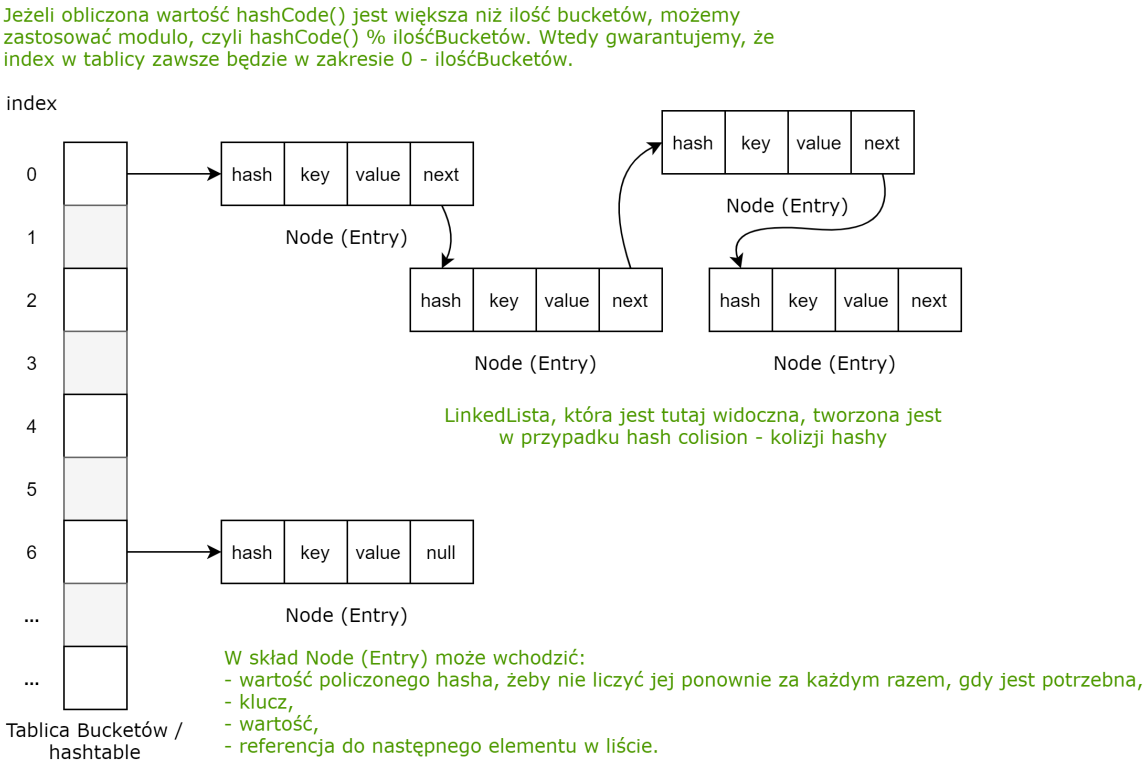
Należy pamiętać o klasie `Entry`, gdyż implementacja `HashMapy` wylicza `hashCode()` z klucza, który jest zdefiniowany w `Entry`, a nie z jej całości. Na tej podstawie następnie następuje przetwarzanie tak jak w pokazanej strukturze `HashTable`.

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

Konkretne 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:




*Obraz 6. 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.

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