**2. Wstęp do kolekcji**

**Wyzwania:**

* nauczysz się korzystać z pętli for,
* poznasz kontrakt equals() i hashCode(),
* poznasz List i Map, czyli zaawansowane tablice.

**2.1. Pętla for**

Bardzo często podczas programowania spotkasz się z koniecznością **wielokrotnego wykonywania** takich samych lub podobnych czynności. Oczywiście pisanie wszystkich instrukcji ręcznie nie byłoby zbyt wygodne, a że programiści są znani ze swojego "lenistwa" i upraszczania sobie wielu rzeczy dzięki automatyzacji, również na ten problem istnieje rozwiązanie — **pętle.**

**Co to jest pętla?**

Zapewne pamiętasz tablice, które omawialiśmy wcześniej w kursie. Jedną z często wykonywanych operacji na tablicach jest **wyświetlenie w oknie konsoli wszystkich elementów tablicy**. Łatwe do zrobienia? No ba!

Problem w tym, że gdybyśmy chcieli napisać taki kod za pomocą poznanych dotychczas metod, musielibyśmy uwzględnić każdy indeks tablicy w kodzie i oddzielnie wywoływać dla niego funkcję wyświetlającą zawartość. Takie rozwiązanie nie jest oczywiście zbyt eleganckie i jego przykład możesz zobaczyć pod [tym linkiem](https://kodilla.com/pl/project-java/38618).

Teraz wyobraź sobie, jak wyglądałby Twój kod, gdyby tablica liczyła 20, 200, 2000 elementów — straszny bałagan! Zauważ, że kolejne linijki kodu różnią się od siebie jedynie indeksem elementu tablicy do wyświetlenia. Właśnie dla tego typu zadań w języku Java (właściwie to praktycznie we wszystkich językach programowania) wymyślono konstrukcję tak zwanej pętli, która ułatwia pisanie powtarzających się sekwencji kodu.

Tego typu zadania będziesz spotykać bardzo często — na przykład kiedy będzie potrzeba wyświetlenia listy wszystkich ustawień użytkownika, listy wszystkich użytkowników, obliczenia średniej ocen w elektronicznym dzienniczku ucznia, itd. Zapewne domyślasz się, jaki nieład w kodzie mogłoby spowodować operowanie na dużych tablicach bez użycia pętli.

**Pętla for**

Pętle wystepują w kilku wariantach, jednak najpopularniejszy z nich to tak zwana pętla for. Taką pętlę zapisuje się w kodzie w następujący sposób:

for(initializing\_expression; loop\_condition; modifying\_expression){

loop\_body

};

Nie wygląda to zbyt przyjaźnie, hm? Rozbijmy więc ten fragment kodu na części pierwsze:

* **initializing\_expression** — przypisanie do zmiennej sterującej pętli wartości początkowej
* **loop\_condition** — warunek pętli. Wyrażenie to zwraca wartość true lub false, a pętla wykonuje się tak długo, jak warunek pętli zwraca wartość true
* **modifying\_expression** — instrukcja zmieniająca wartość zmiennej sterującej pętli po każdym jej przebiegu
* **loop\_body** — instrukcje do wykonania wewnątrz pętli (ciało pętli)

Przykład kodu z pętlą for mógłby więc wyglądać następująco:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class BetterLoop

{

public static void main (String[] args) throws java.lang.Exception

{

String[] names = new String[5];

names[0] = "Cat";

names[1] = "Dog";

names[2] = "Horse";

names[3] = "Elephant";

names[4] = "Tiger";

for(int i=0; i<5; i++){

System.out.println("Array element names[" + i + "] = " + names[i]);

};

System.out.println(names.length);

}

}

<https://kodilla.com/pl/project-java/38619>

Zatrzymajmy się chwilę w tym miejscu, aby zrozumieć, co tak naprawdę dzieje się w pętli z powyższego przykładu:

1. W pierwszym kroku następuje wykonanie wyrażenia inicjującego, czyli int i = 0. Za pomocą tej instrukcji deklarujemy nową zmienną typu int o nazwie i, oraz przypisujemy do niej wartość początkową — zero. Nazywanie tej zmiennej "i" jest ogólnie przyjętą praktyką, to nic innego jak skrót od słowa "iterator".
2. Sprawdzany jest warunek pętli, czyli i<5. Zmienna i wynosi zero, więc warunek jest spełniony (jego wartość wynosi true, czyli prawda), więc pętla może wykonać jeden przebieg.
3. Następuje wykonanie przebiegu pętli, po prostu wykonanie kodu zawartego w ciele pętli. W tym przypadku będzie to następująca linijka:

System.out.println("Array element names[" + i + "] = " + names[i]);

W miejsce zmiennej i wstawiana jest jej wartość — obecnie jest to zero. Na ekranie konsoli wyświetli się więc napis:

Array element names[0] = Cat

1. Następuje wykonanie wyrażenia modyfikującego zmienną i w pętli, czyli i++. W wyniku tego do zmiennej i zostanie przypisana wartość o jeden większa (tzw. inkrementacja) od dotychczasowej — w tym przypadku wynikiem będzie 1.
2. Sprawdzany jest warunek pętli, czyli i<5. Zmienna i wynosi jeden, więc warunek jest spełniony i pętla może wykonać jeden przebieg.

Następnie program wraca do kroku 3 i tak w kółko, aż wartość zmiennej i wyniesie 5. Wówczas warunek pętli przestanie być spełniony (jego wartość zostanie wyliczona jako false, czyli fałsz) i pętla zakończy swoje działanie, a sterowanie w programie przejdzie do następnej instrukcji po pętli for.

Warto zwrócić uwagę na to, że w momencie, kiedy wartość zmiennej i będzie równa 5, instrukcje zawarte w ciele pętli już się nie wykonają. W momencie sprawdzania warunku pętli (krok 5), wartość zmiennej i nie będzie już mniejsza niż 5, więc sprawdzenie warunku nie dopuści do wykonania kodu ponownie.

**Przypomnienie**

Operację zwiększania wartości dowolnej zmiennej numerycznej o jeden nazywamy inkrementacją zmiennej i możemy ją zapisać na dwa równoważne sposoby w kodzie:

* n = n + 1
* n++

Obydwa powyższe sposoby robią dokładnie to samo i po prostu zwiększają wartość zmiennej n o jeden.

Operację zmniejszania wartości dowolnej zmiennej numerycznej o jeden nazywamy dekrementacją zmiennej i również możemy ją zapisać na dwa równoważne sposoby w kodzie:

* n = n - 1
* n--

Pojedynczy przebieg pętli nazywamy jej **iteracją**, zaś działanie pętli polegające na przechodzeniu kolejnych jej iteracji nazywamy **iterowaniem** pętli.

**Wykorzystanie pętli for**

Pętle można wykorzystywać do wielu różnych powtarzalnych czynności, na przykład do sumowania elementów tablicy czy obliczania średniej wszystkich jej elementów, do wyświetlania tych elementów czy też przykładowo do rysowania kolejnych słupków wykresu w raporcie.

Najczęściej pętle służą do wykonywania operacji na tablicach czy **kolekcjach** (*ang. collection*) danych, o których więcej dowiesz się za chwilę w niniejszym module.

**Liczba wykonań pętli a długość tablicy**

Pętle często znajdą również zastosowanie w sytuacjach, gdy liczba elementów kolekcji (np. tablicy) nie jest znana na etapie pisania programu (czyli jest dynamiczna i może się zmieniać w trakcie działania programu). Wówczas w warunku pętli nie umieścimy konkretnej stałej wartości do porównania, jak na przykład i<10 — w to miejsce posłużymy się wyrażeniem sprawdzającym liczbę elementów w **kolekcji**.

Przykładowo, dla tablicy o nazwie array, będzie to wyrażenie: i<array.length, gdzie właściwość length tablicy zwraca liczbę odpowiadającą ilości elementów w tablicy. Przykładowy kod znajdziesz poniżej:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class BetterDynamicLoop

{

public static void main (String[] args) throws java.lang.Exception

{

String[] names = new String[5];

names[0] = "Cat";

names[1] = "Dog";

names[2] = "Horse";

names[3] = "Elephant";

names[4] = "Tiger";

for(int i=0; i < names.length; i++){

System.out.println("Array element names[" + i + "] = " + names[i]);

};

System.out.println(names.length);

}

}

<https://kodilla.com/pl/project-java/38620>

**Nieskończona pętla**

Trzeba w tym miejscu również wspomnieć o zagrożeniach, jakie mogą czyhać na programistę — jednym z nich jest zjawisko "nieskończonej pętli". Wyobraź sobie, że wewnątrz pętli ingerujemy w wartość zmiennej sterującej pętli (czyli w naszym przykładzie w wartość zmiennej i). Mogłoby to być na przykład zmniejszanie wartości zmiennej i o jeden w każdym przebiegu, jak w poniższym przykładzie:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class InifinityLoop

{

public static void main (String[] args) throws java.lang.Exception

{

String[] names = new String[5];

names[0] = "Cat";

names[1] = "Dog";

names[2] = "Horse";

names[3] = "Elephant";

names[4] = "Tiger";

for(int i=0; i<5; i++){

System.out.println("Array element names[" + i + "] = " + names[i]);

i--;

};

}

}

<https://kodilla.com/pl/project-java/38621>

W tej sytuacji wartość zmiennej i wynosiłaby ciągle zero, ponieważ w każdym przebiegu pętli byłaby zwiększana o jeden w wyrażeniu modyfikującym pętli, a za chwilę zmniejszana o jeden wewnątrz ciała pętli. Taka pętla nigdy nie zakończyłaby swojego działania i wykonywałaby się non stop (sprawdź, jaki efekt wywoła uruchomienie przykładowego programu podanego powyżej). Użytkownik odebrałby to jako "*zawieszenie się*" programu.

W przypadku pętli for rzadko modyfikujemy zmienną sterującą wewnątrz pętli, więc ryzyko wystąpienia pętli nieskończonej jest niewielkie. Istnieją jednak inne rodzaje pętli, jak na przykład pętla while, czy też pętla do-while, które poznasz w dalszej części kursu. Te rodzaje pętli są bardziej podatne na wystąpienie zjawiska pętli nieskończonej, ponieważ pętle te, w odróżnieniu od pętli for, nie kontrolują same procesu modyfikowania zmiennej sterującej pętli, robi to natomiast samodzielnie programista wewnątrz pętli. Należy więc o tym pamiętać.

**Pętla w pętli**

Ważnym aspektem dotyczącym pętli for (jak i zresztą innych typów pętli) jest to, że mogą one być zagnieżdżane. Innymi słowy, możliwe jest napisanie programu zawierającego "pętlę w pętli". Wygląda to mniej więcej tak:

for(int i=0; i<10; i++){

for(int n=0; n<5; n++){

loop\_body

};

};

Działanie pętli zagnieżdżonej polega na tym, że dla każdego przebiegu pętli zewnętrznej (czyli tej ze zmienną sterującą i), wykonywana jest pełna pętla wewnętrzna (czyli ta ze zmienną n). Innymi słowy, pętla zewnętrzna uruchamia w swoim ciele inną, wewnętrzną pętlę, czeka na jej wykonanie i uruchamia się od nowa, dopóki warunek pętli jest prawdziwy. Takie konstrukcje wykorzystuje się najczęściej w przypadku działania na tzw. tablicach dwuwymiarowych, czyli po prostu tablic zawierających inne tablice.

**Zadanie: Obliczamy średnią**

Napisz program obliczający wartość średnią 20 elementów tablicy. Pamiętaj, że:

* elementy tablicy powinny być typu int,
* wynik prawdopodobnie nie będzie wartością całkowitą,
* pętla powinna wyświetlić wszystkie elementy tablicy,
* na koniec powinna zostać wyświetlona wartość średniej.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/479784)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162685)

**2.2. Notacja Big-O**

**Co to jest notacja Big-O?**

W trakcie pracy z **kolekcjami** (i nie tylko), często staniesz przed wyborem: którego algorytmu użyć? Dobrze jest w takiej sytuacji wiedzieć, w jaki sposób można porównywać algorytmy ze sobą. Jednym z ciekawych kryteriów porównania algorytmów jest ich złożoność, która wprost przekłada się na wydajność algorytmu.

Złożoność algorytmów możemy rozpatrywać w kontekście liczby obliczeń potrzebnych do wykonania (złożoność czasowa lub obliczeniowa), lub w kontekście potrzebnych do działania algorytmu zasobów (złożoność pamięciowa). Tym razem zajmiemy się złożonością obliczeniową.

Aby w jakiś znormalizowany sposób opisać złożoność obliczeniową algorytmów, wymyślono specjalny zapis — tak zwaną notację Big-O ( wjęzyku polskim można się też spotkać z pojęciem O-notacja).

Złożoność algorytmu wyrażona w O-notacji mówi nam o tym, jaka jest zależność pomiędzy wielkością zbioru danych wejściowych (np. elementów tablicy czy rekordów w bazie danych), a liczbą obliczeń niezbędnych do wykonania.

**Do czego służy?**

Wiedząc już, o co chodzi z **O-notacją**, łatwiej będzie Ci porównywać i dokonywać wyboru pomiędzy różnymi algorytmami służącymi rozwiązaniu tego samego problemu — dzięki temu łatwo wybierzesz najlepsze podejście do zagadnienia.

**Jak się ją wylicza?**

Aby zrozumieć, jak wylicza się złożoność algorytmu, skorzystajmy z przykładu. Jeżeli posiadamy zbiór n elementów w tablicy i dla każdego z nich musimy wykonać jedno obliczenie, to łączna liczba obliczeń w algorytmie będzie liniowo zależna od liczby elementów w tablicy. Mówimy wówczas, że algorytm ma złożoność obliczeniową wynoszącą:

O(n)

gdzie n oznacza ilość danych wejściowych.

W sytuacji, gdy algorytm wykonuje bardziej złożone obliczenia, na przykład gdy posiadamy dwie tablice jednowymiarowe — jedną zawierającą i elementów, i drugą zawierającą k elementów — działanie algorytmu polega na wykonaniu pętli po i elementach pierwszej tablicy oraz w ramach każdego przebiegu pętli wykonuje dodatkową pętlę po k elementach drugiej tablicy (pętla zagnieżdżona).

Przykład takiego programu możesz obejrzeć w linku poniżej:

<https://kodilla.com/pl/project-java/38623>

Złożoność zaprezentowanego algorytmu wynosi:

O(i \* k)

Aby ten zapis nieco uprościć, przyjmuje się, że wartości i oraz k są tak samo duże i oznacza się je jedną zmienną n. Pamiętasz asymptoty z matematyki? To były te linie na wykresach, do których funkcja zmierzała, ale których nigdy nie osiągała. **O-notacja** jest właśnie tak zwaną **notacją asymptotyczną** i określa górną (najgorszą możliwą) granicę złożoności algorytmów, dlatego też w skrócie przyjmujemy, że zarówno i jak i k są równe większej z wartości i oraz k i wartość tę oznaczamy przez n.

W praktyce może się zdarzyć, że zmienne i oraz k będą miały jakieś korzystniejsze dla wydajności wartości, np. 1 i 500 (pętla wewnętrzna będzie miała wówczas 1 x 500 = 500 przebiegów). Równie dobrze mogą jednak wynosić 500 i 500 (pętla wewnętrzna będzie miała wówczas 500 x 500 = 250000 przebiegów).

Złożoność określamy zawsze dla przypadku pesymistycznego, najgorszego — na tym polega asymptotyczność O-notacji. Bierzemy pod uwagę wartość asymptoty, która prawdopodobnie się nie zdarzy, ale w najgorszym możliwym scenariuszu taka sytuacja będzie właśnie miała miejsce. W tej sytuacji, po zastąpieniu i oraz k niewiadomą n, widzimy, że złożoność obliczeniowa algorytmu będzie wynosiła:

O(n \* n), czyli O(n^2).

Spotykane w praktyce złożoności obliczeniowe algorytmów, ułożone w kolejności od najlepszej (najszybszy algorytm) do najsłabszej (najwolniejszy algorytm) przedstawiono poniżej:

* O(c) — złożoność stała, niezależna od ilości danych wejściowych
* O(log n) — złożoność logarytmiczna, spotykana na przykład przy przeszukiwaniu uporządkowanych *drzew binarnych*
* O(n) — złożoność liniowa
* O(n^2) — złożoność kwadratowa
* O(c^n) — złożoność wykładnicza

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

Zapoznałe(a)m się!

**2.3. Kolekcje część pierwsza**

**Skąd biorą się biblioteki i do czego służą?**

Na przestrzeni lat, twórcy języka Java zbudowali całkiem sporo fragmentów kodu, który może być ponownie wykorzystywany przez innych programistów. Fragmenty te rozwiązują typowe problemy, na przykład pozwalają wygodnie operować na zmiennych przechowujących daty, służą do generowania liczb pseudolosowych czy też stanowią narzędzia do pisania oprogramowania wielojęzycznego (przeznaczonego na rynki w różnych krajach).

Wspomniane gotowe kawałki oprogramowania zostały przez twórców języka Java pogrupowane w tak zwane biblioteki (ang. *library*) i są dostarczane programistom wraz z samym językiem Java, aby programista nie musiał wymyślać wszystkiego sam od zera. Wprowadza to także swoistą standaryzację, dzięki czemu łatwiej jest się "przesiadać" programistom pomiędzy różnymi projektami — wiele z nich wykorzystuje te same biblioteki, więc duża część kodu będzie od razu znajoma programiście.

**java.util**

Jedną z najbardziej podstawowych i powszechnie stosowanych bibliotek Javy jest biblioteka java.util.

W bibliotece java.util można znaleźć wiele użytecznych funkcji i gotowych rozwiązań, na przykład:

* klasy ułatwiające operowanie na zmiennych przechowujących daty,
* klasy pozwalające na generowanie liczb losowych,
* narzędzia do uruchamiania zadań w tle w ustalonych odstępach czasu (tzw. timery),
* zestaw narzędzi do operowania na zbiorach obiektów (o tym więcej znajdziesz poniżej) — tak zwany *Java Collection Framework*.

Oprócz bibliotek standardowych dołączanych do Javy, na rynku funkcjonuje bardzo wiele różnych bibliotek — czy to dostępnych komercyjnie, czy też rozwijanych przez społeczność w modelu open source.

**Przypomnienie**

Być może nie znasz terminu open source, więc należą Ci się dwa słowa wyjaśnienia:

*Source*, czyli Źródło po angielsku, oznacza po prostu kod aplikacji (czyli tzw. kod źródłowy).  
*Open* oznacza po angielsku *otwarty*.

Jak należy to rozumieć? Jak zapewne Ci wiadomo, spora ilość oprogramowania powstaje w firmach komercyjnych po to, aby później takie oprogramowanie sprzedawać użytkownikom— w takim przypadku mówimy o oprogramowaniu komercyjnym.

Czasem jakaś grupa zapaleńców postanawia stworzyć oprogramowanie i udostępnić je wszystkim bez pobierania żadnych opłat, a do jego współtworzenia zaproszony jest każdy chętny. Oprogramowanie takie powstaje w formie *"wolontariatu dla społeczności programistów"* — każdy może bez opłat używać tego oprogramowania, i każdy może je modyfikować. To jest właśnie oprogramowanie open source.

**Podpinanie biblioteki**

Aby skorzystać z klas i interfejsów udostępnianych przez daną bibliotekę, należy ją "zaimportować". Robi się to przy pomocy słowa kluczowego import umieszczonego na początku klasy, na przykład:

import java.util;

Dla przykładu, zerknij na standardowy nagłówek programu — zwróć uwagę na górną część kodu, w której widać wspomniane importy:

<https://kodilla.com/pl/project-java/38624>

Polecenia import będziesz również używać do uzyskiwania dostępu pomiędzy klasami, które napiszesz samodzielnie. Standardowo w języku Java przyjmuje się, że jedna klasa zapisywana jest w jednym pliku z rozszerzeniem .java, więc ile klas w programie, tyle plików.

Aby klasy mogły się nawzajem "widzieć" (czyli mieć do siebie nawzajem dostęp), będziesz używać właśnie polecenia import. Tymi szczegółami zajmiemy się jednak w dalszych modułach kursu, gdy przesiądziesz się z uproszczonego środowiska, którego używamy w edytorze wewnętrznym Kodilla, na docelowe środowisko do programowania, czyli IntelliJ.

**Popularne biblioteki**

Do standardowych bibliotek Javy zaliczamy między innymi (najczęściej używane):

* java.io — biblioteka służąca do realizowania zadań związanych z wymianą danych między innymi poprzez odczyt i zapis do pliku (io — skrót oznaczający *Input/Output, wejście/wyjście*),
* java.net — biblioteka służąca do komunikowania się przez internet,
* java.lang — podstawowa biblioteka języka Java. Biblioteka ta nie musi być importowana do pisanych klas — jest ona domyślnie dołączana przez kompilator Javy do każdej klasy,
* java.util — biblioteka zawierająca zestaw narzędzi do operowania na kolekcjach, datach czy kalendarzach.

Pełna lista bibliotek wchodzących w skład Java 8 SE dostępna jest w dokumentacji języka Java pod linkiem:

<https://docs.oracle.com/javase/8/docs/api/>

Lista interfejsów i klas udostępnianych przez poszczególne biblioteki również jest dostępna w dokumentacji Javy w lokalizacji wskazanej powyżej. Przykładowo pełny opis zawartości biblioteki java.util znajdziesz pod linkiem:

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

Oczywiście nie musisz się tego uczyć na pamięć — programista najczęściej pracuje posiadając dokumentację pod ręką i zagląda do niej często, gdy czegoś potrzebuje. Podstawowe elementy bibliotek warto jednak poznać, choćby po to, aby wiedzieć, że one w tych bibliotekach są oraz aby było wiadomo, w której części dokumentacji szukać szczegółów.

Bieżący moduł kursu poświęcony jest jednej z podstawowych struktur, których będziesz bardzo często używał podczas programowania. Tymi strukturami są kolekcje. Narzędzia do operowania na kolekcjach zawarte są właśnie w bibliotece java.util.

**Czym jest kolekcja?**

Po długim, ale koniecznym wstępie, możemy przejść w końcu do **kolekcji** :)

**Przykłady kolekcji**

Zastanówmy się, jak to jest w realnym świecie — kolekcją może być np. kolekcja obrazów, kolekcja znaczków czy też kolekcja biletów. Ogólnie możemy powiedzieć, że kolekcja to zbiór kolekcjonowanych elementów (obiektów), w którym mamy możliwość przeglądania poszczególnych obiektów, mamy również możliwość dodawania nowych obiektów do kolekcji oraz usuwania obiektów z kolekcji. Istotne jest również to, że w ramach kolekcji przechowujemy obiekty tego samego typu — przykładowo klaser ze znaczkami pocztowymi przechowuje tylko znaczki, opakowanie z lekami tylko leki, i tak dalej.

W programowaniu jest podobnie — kolekcja to zbiór obiektów, na którym możemy wykonywać operacje: dodawania obiektu do kolekcji, usuwania obiektów z kolekcji oraz przeglądania obiektów wchodzących w skład kolekcji. Kolekcje mogą być uporządkowane (np. alfabetycznie) lub nieuporządkowane.

**Rodzaje kolekcji**

Kolekcje mogą być zorganizowane w różny sposób. Najprostszym typem kolekcji jest **lista** (*ang. list*), co w przybliżeniu odpowiada ułożonym kolejno według ceny znaczkom pocztowym w klaserze (kolekcja uporządkowana) lub też znaczkom pocztowym wrzuconym luźno do pudełka (kolekcja nieuporządkowana).

Innym rodzajem kolekcji jest **mapa** (*ang. map*). Mapa jest zbiorem elementów typu **klucz-wartość** albo **klucz-obiekt** (*ang. key-value*). Jak to rozumieć? Wyobraźmy sobie bibliotekę z książkami. Każda książka ma nadany numer sygnatury bibliotecznej, czyli unikalny identyfikator książki. W programowaniu taki identyfikator nazywamy **kluczem**.

Jeżeli spojrzymy na bibliotekę z książkami, to jest ona właśnie kolekcją typu mapa, ponieważ jednoznacznie *mapuje* identyfikatory książek (klucze) na same książki (czyli obiekty albo inaczej — wartości). W takiej kolekcji możemy wyszukiwać zarówno przy pomocy klucza, jak i przy pomocy obiektu (wartości). Więcej o mapach dowiesz się w dalszej części modułu.

**java.util a kolekcje**

W bibliotece java.util znajdują się interfejsy służące do obsługi kolekcji. Najważniejsze z nich to:

* interface Collection — podstawowy interfejs dla kolekcji
* interface List — interfejs dla kolekcji realizowanych w postaci list (dziedziczy z interfejsu Collection)
* interface Map — interfejs dla kolekcji realizowanych w postaci map

Jak już wiesz, interfejs jest tylko definicją "kontraktu", czyli określa, jakie metody mogą być wywoływane w klasach implementujących dany interfejs.

W pakiecie java.util znajduje się również zestaw klas służących do obsługi kolekcji, które implementują wspomniane powyżej interfejsy List oraz Map. Jakie to są konkretnie klasy, dowiesz się za chwilę.

Wymienione wyżej interfejsy to oczywiście nie wszystkie mechanizmy służące do obsługi kolekcji, które możemy znaleźć w bibliotece java.util. Znajomość powyższych jest jednak kluczowa, a pozostałe są ich różnego rodzaju modyfikacjami, mutacjami i rozszerzeniami.

**Interface Collection**

Omówienie kolekcji rozpoczniemy od omówienia bazowego interfejsu dla wszystkich rodzajów kolekcji — od interfejsu Collection. Interfejs ten dziedziczy z interfejsu Iterable i udostępnia wszystkie jego metody. Do czego służy interfejs Iterable? Aby na to odpowiedzieć, poznaj nieco inną konstrukcję pętli for.

**Przypomnienie**

Interfejsy, tak samo jak klasy, mogą między sobą dziedziczyć. Możemy posiadać np. ogólny interfejs FiguraGeometryczna, z którego dziedziczyć może bardziej szczegółowy interfejs FiguraPlaska, z którego z kolei może dziedziczyć interfejs Rownoleglobok. Taki interfejs może być następnie implementowany przykładowo przez klasy Kwadrat, Prostokat, Romb.

W praktyce, w bibliotece java.util często mamy do czynienia z dziedziczeniem interfejsów. Na przykład Iterable - Collection - List, z których ostatni implementowany jest między innymi przez klasę ArrayList, którą omówimy niebawem.

Pamiętasz, jak omawialiśmy pętlę for? Wówczas wspominaliśmy, że składnia dla pętli for jest następująca:

for(initializing\_expression; loop\_condition; modifying\_expression){

…

};

Natomiast przykładowa pętla for wyglądała w następujący sposób:

for(int i=0; i<5; i++){

System.out.println("Element array[" + i + "] equals: " + array[i]);

};

Istnieje również inna składnia pętli for, nazywana for-each, czyli *dla każdego*. Poniżej możesz się z nią zapoznać bliżej.

**Pętla for-each**

Służy ona do wykonania pętli dla każdego elementu kolekcji bez konieczności podawania warunku pętli, a także wyrażenia modyfikującego zmienną sterującą pętli. Jej składnia jest następująca:

for(type variable : iterable) {

…

}

Gdzie:

* type — typ elementu w kolekcji
* variable — nazwa zmiennej, do której przypisywany jest element kolekcji podczas działania pętli
* iterable — kolekcja, na której ma zostać wykonana pętla

Przykładowa pętla for tego typu może wyglądać następująco:

for(String name: names){

System.out.println(name);

}

Gdzie names to kolekcja zawierająca np. nazwy użytkowników.

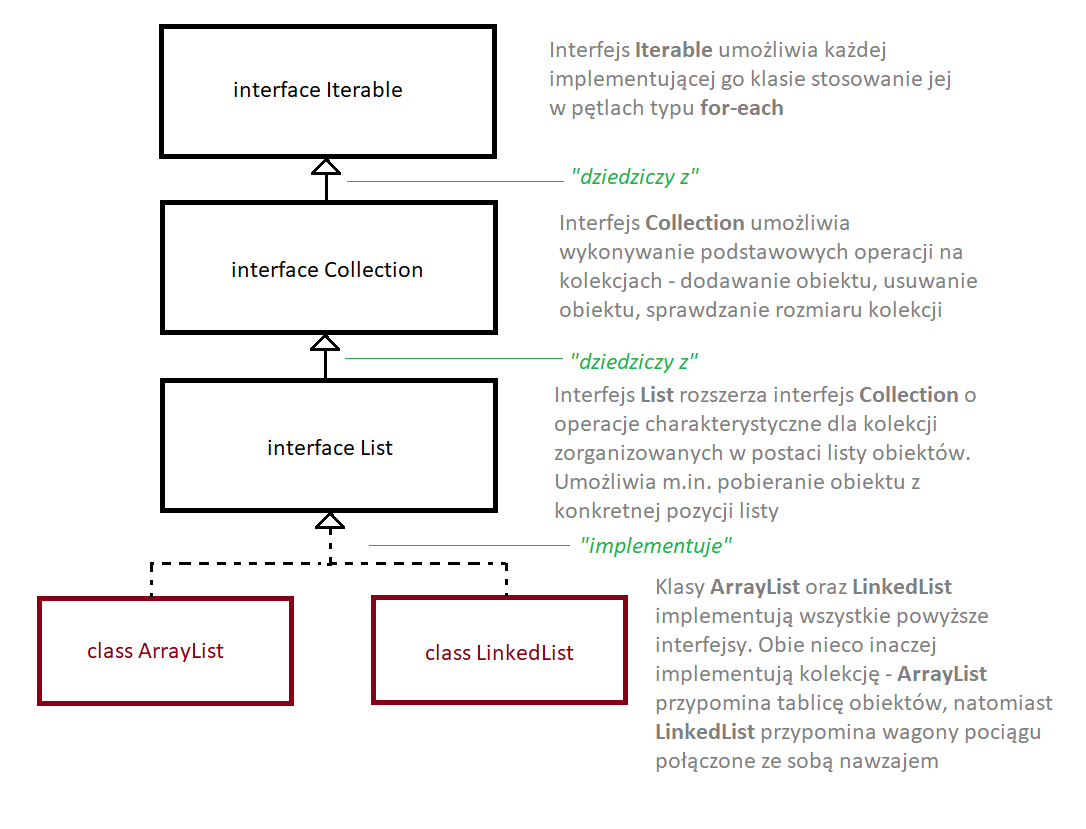
Z definicji pętli for widać, że elementy tej kolekcji są typu String. Działanie tej pętli polega na tym, że do zmiennej name (typu String) przypisywana jest wartość kolejnego elementu kolekcji names, a następnie — już wewnątrz pętli — nazwa wyświetlana jest na ekranie konsoli.

Zauważ, że nie podajemy tutaj żadnych indeksów górnych ani dolnych — wskazujemy jedynie kolekcję, na której ma zostać wykonana pętla. Kompilator sam zatroszczy się o to, aby instrukcje wewnątrz pętli zostały wykonane kolejno dla każdego elementu kolekcji — innymi słowy, aby działanie programu polegało na przejściu po kolejnych elementach kolekcji.

**Interfejs Iterable**

Wróćmy teraz do naszego interfejsu Iterable. Otóż każda klasa implementująca ten interfejs może być użyta jako typ zmiennej sterującej pętli typu for-each.

Spójrz na poniższy rysunek przedstawiający wzajemne zależności pomiędzy niektórymi interfejsami i klasami w ramach Java Collections Framework. Poszczególne elementy tego rysunku (klasy i interfejsy) omówimy kolejno w dalszej części modułu.

[](https://kodilla.com/static/bootcamp-java/java-2_01.png)

Skoro więc nasz bazowy interfejs dla kolekcji — Collection — dziedziczy z interfejsu Iterable, oznacza to, że każda klasa implementująca interfejs Collection może być użyta jako typ zmiennej sterującej dla pętli for-each. Przykład poniżej.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class IterableTest

{

public static void main (String[] args) throws java.lang.Exception

{

//Creating ArrayList, which implements the Iterable interface and fill it

ArrayList<String> theList = new ArrayList<String>();

theList.add("object 1");

theList.add("object 2");

theList.add("object 3");

theList.add("object 4");

theList.add("object 5");

//Displaying the list using Iterable interface and for-each loop

for(String theObject: theList){

System.out.println(theObject);

}

}

}

<https://kodilla.com/pl/project-java/38625>

**Dla dociekliwych**

Interfejs Map nie dziedziczy tak naprawdę z bazowego interfejsu Collection. Posiada jednak mechanizmy, które umożliwiają traktowanie go tak, jak gdyby z interfejsu Collection dziedziczył. Wobec tego, mimo, że Map nie jest typową kolekcją, to jednak może być tak traktowany i stanowi część *Java Collection Framework*.

**Interfejs Collection**

Sam interfejs Collection wprowadza również własne metody charakterystyczne dla kolekcji (poza odziedziczonymi z interfejsu Iterator związanymi z pętlą for-each). Najistotniejsze z nich to:

* add(E e) — dodaje nowy element e typu E do kolekcji.
* remove(Object o) — usuwa element o z kolekcji.
* contains(Object o) — sprawdza, czy obiekt o znajduje się w kolekcji i zwraca true lub false.
* size() — zwraca informację o liczbie elementów w kolekcji (uwaga — tu jest inaczej, niż przy tablicach. W tablicach używamy właściwości length).
* clear() — usuwa wszystkie elementy z kolekcji.
* isEmpty() — sprawdza, czy kolekcja jest pusta i nie zawiera żadnych elementów. Zwraca true lub false.
* toArray() — zwraca całą kolekcję w postaci tablicy.

Wymienione powyżej metody nie są wszystkimi metodami interfejsu Collection, jednak na razie w zupełności nam wystarczą.

Bez względu na to, z jakiego rodzaju kolekcją będziemy mieli do czynienia (list, map), możemy być pewni, że każda kolekcja pozwala na wykonywanie operacji udostępnianych przez interfejs Collection.. Dodatkowo, każda z kolekcji może być użyta jako ostatni argument pętli for-each.

**Przypomnienie**

Pętla typu for-each posiada następującą składnię:

for(Type object: collection) { ...<loop body>... };

Gdzie:

* Type — typ (klasa) obiektów, z których składa się kolekcja,
* object — zmienna, do której będą podstawiane kolejne obiekty z kolekcji przy poszczególnych przebiegach pętli,
* collection — kolekcja, po której iteruje pętla for-each.

W kolejnych punktach modułu przyjrzymy się bliżej poszczególnym, wybranym rodzajom kolekcji.

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

Zapoznałe(a)m się!

**2.4. Wrappery**

Jak pamiętasz z poprzednich modułów (dotyczących typów danych), w Javie mamy do czynienia z typami prymitywnymi (np. int, boolean, ...) lub z typami obiektowymi (np. obiekt utworzony na podstawie klasy).

Kolekcje w języku Java operują co do zasady na obiektach. Na szczęście w języku Java istnieją specjalne klasy odpowiadające typom prymitywnym — są to tzw. **wrappery** (po polsku możemy określić je mianem "opakowania").

Przykładowo, typowi prymitywnemu int odpowiada typ obiektowy Integer. Typowi prymitywnemu boolean odpowiada typ obiektowy Boolean. Nie wnikając na razie w to, jak jest to zrealizowane w języku Java, wystarczy zapamiętać, że możliwe są wprost przypisania pomiędzy typami prymitywnymi i odpowiadającymi im typami obiektowymi. Tak więc, zmiennej o typie obiektowym można przypisać zmienną o typie prymitywnym:

int n = 10;

Integer k = n;

int q = k;

Na powyższym przykładzie widzimy, że zmienna n, będąca typu prymitywnego int, jest przypisywana do zmiennej k, mającej typ obiektowy Integer.

Pełna lista dostępnych w języku Java wrapperów znajduje się poniżej:

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

Pewnie zastanawiasz się, dlaczego akurat teraz wspomnieliśmy o wrapperach. Wynika to z faktu, że kolekcje zawsze przechowują typy obiektowe, więc aby wykorzystać jakikolwiek typ prymitywny, np. int w kolekcji, musimy użyć właśnie jego wrappera — więcej o tym za chwilę.

**Zadanie: Wrapper prymitywowi (nie)równy?**

Napisz program, przypisujący jakieś wartości do zmiennych o typach prymitywnych. Następnie takie same wartości przypisz do typów będących wrapperami typów prymitywnych. Niech program porówna, czy wartości w typach prymitywnych oraz odpowiadających im wrapperach są sobie równe. Do realizacji zadania wybierz trzy dowolne typy prymitywne.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/479787)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162686)

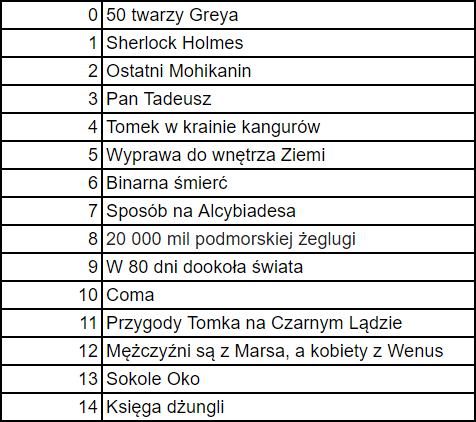
**2.5. ArrayList i LinkedList**

**ArrayList**

W jaki sposób zorganizowane są kolekcje typu lista? W bibliotece java.util znajdziemy kilka różnych implementacji tego typu kolekcji. Najistotniejszym (najczęściej stosowanym) i zarazem najprostszym rodzajem kolekcji jest lista typu tablicowego.

Klasa implementująca interfejs List w sposób przypominający tablicę nazywa się ArrayList. Tablice omówiliśmy już wcześniej, więc mniej więcej przypominasz sobie, jak są one zorganizowane. Dla przypomnienia — tablice zawierają elementy w kolejnych "komórkach", a dostęp do zawartości tych komórek uzyskuje się poprzez indeks (numer) danej komórki.

Dokładnie tak zrealizowana jest kolekcja typu ArrayList. Wyobraź sobie listę książek w bibliotece zapamiętaną w kolekcji typu tablicowego. Aby sobie to zobrazować, spójrz na poniższy rysunek.

[](https://uploads.kodilla.com/bootcamp/kodilla-bootcamp-27-06-2018-1530109485-7456.png)

Poszczególne tytuły książek znajdują się w kolejnych komórkach tablicy i opatrzone są numerem komórki w tablicy (pamiętaj, że elementy tablic numerujemy zaczynając od 0, nie od 1).

**Tworzenie ArrayList**

Wyobraź sobie teraz, że tego typu kolekcję możesz utworzyć w języku Java. Jak to zrobić? Należy utworzyć nowy obiekt klasy ArrayList (klasa ta jest dostępna w bibliotece java.util, więc nie musisz jej pisać samodzielnie), a następnie przy pomocy metody add(Object o) należy dodać do kolekcji kolejne pozycje. Kod, który tworzy taką kolekcję, wygląda następująco:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class ArrayListTest

{

public static void main (String[] args) throws java.lang.Exception

{

ArrayList<String> library = new ArrayList<String>();

library.add("50 Shades of Grey");

library.add("Sherlock Holmes");

library.add("The Last of the Mohicans");

library.add("The Infinity");

library.add("Better Call Saul");

library.add("Journey to the Center of the Earth");

library.add("Binary Death");

library.add("Breaking Bad");

library.add("Twenty Thousand Leagues Under the Sea");

library.add("The Titanfall");

library.add("Coma");

library.add("Of Ants and Men");

library.add("To Understand a Woman");

library.add("Neverending Story");

library.add("The Jungle Book");

System.out.println(library.size());

}

}

<https://kodilla.com/pl/project-java/38626>

Na pewno rzuciła Ci się w oczy linia 10, w której tworzona jest kolekcja typu ArrayList:

ArrayList<String> library = new ArrayList<String>();

Bezpośrednio za nazwą klasy (ArrayList), pojawił się w tej linii ciąg znaków <String>. W ten sposób mówimy kompilatorowi Javy, jakiego typu będą obiekty składające się na listę. W tym przypadku będą to ciągi znaków, czyli String. Jak widzisz, kolekcje tworzymy prawie tak samo, jak obiekty :)

Oczywiście mogą to być dowolne typy obiektowe. W przypadku, gdy chcemy przechowywać w kolekcji bardziej złożone struktury danych (np. Tytuł + Autor + Rok wydania), najlepiej jest zdefiniować osobną klasę Book z atrybutami title, author oraz year i w ramach kolekcji przechowywać obiekty typu Book.

**Ważne**

Nie możemy mieszać różnych typów w kolekcji — wszystkie elementy w kolekcji muszą być tego samego typu.

Ten dziwny zapis (czyli <String> po nazwie klasy) jest konieczny, ponieważ klasa ArrayList jest tak zwanym prototypem klasy. O prototypach dowiesz się więcej w późniejszych modułach — na razie wystarczy zapamiętać, że w przypadku kolekcji po nazwie klasy (albo po nazwie interfejsu) należy w nawiasach ostrokątnych podać typy obiektów, które będą przechowywane w kolekcji.

**Ważne**

Obiekty przechowywane w kolekcji muszą być typami obiektowymi — nie mogą to być typy prymitywne, takie jak int, boolean i podobne. Zamiast nich, należy użyć odpowiadających im typów obiektowych (czyli wrapperów, które poznaliśmy przed chwilą): Integer, Boolean, itd. Wyjątek stanowi tu typ String, który zawsze jest typem obiektowym i nie istnieje w postaci typu prymitywnego.

Program, który przed chwilą omawialiśmy, podzielony jest na trzy funkcjonalne części:

1. utworzenie kolekcji typu ArrayList (linia 10),
2. wypełnienie kolekcji danymi przy pomocy metody add(Object o) (linie 12-26),
3. sprawdzenie, czy kolekcja zawiera wprowadzone do niej dane poprzez wyświetlenie jej aktualnego rozmiaru przy pomocy metody size() (linia 28).

**Przeglądanie danych zapisanych w kolekcji typu ArrayList**

W jaki sposób możemy operować na danych zapisanych w kolekcji? Na początek zastanówmy się nad odczytaniem danych. Jak wspomnieliśmy wcześniej, do obiektów kolekcji możemy odwoływać się przy pomocy metody get(int index), jako argument podając numer (indeks) elementu kolekcji.

Zmodyfikujmy nasz program tak, aby po wypełnieniu kolekcji cała jej zawartość została wyświetlona w konsoli.

Do tego celu użyjemy pętli for. Przy pomocy tej pętli przeiterujemy po wszystkich elementach kolekcji i każdy z nich wyświetlimy. Skorzystamy tu z tego, że wiemy, jaki rozmiar ma kolekcja — mówi nam o tym metoda size(). Użyjemy jej w warunku pętli do sprawdzenia, czy podczas wyświetlania dotarliśmy do końca kolekcji. Spójrz na poniższy kod:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class ArrayListAndForLoop

{

public static void main (String[] args) throws java.lang.Exception

{

ArrayList<String> library = new ArrayList<String>();

library.add("50 Shades of Grey");

library.add("Sherlock Holmes");

library.add("The Last of the Mohicans");

library.add("The Infinity");

library.add("Better Call Saul");

library.add("Journey to the Center of the Earth");

library.add("Binary Death");

library.add("Breaking Bad");

library.add("Twenty Thousand Leagues Under the Sea");

library.add("The Titanfall");

library.add("Coma");

library.add("Of Ants and Men");

library.add("To Understand a Woman");

library.add("Neverending Story");

library.add("The Jungle Book");

for(int n=0; n < library.size(); n++) {

System.out.println("Element " + n + " of the library is: " + library.get(n));

}

}

}

W związku z tym, że wszystkie kolekcje implementują interfejs Iterable, o czym wspomnieliśmy wcześniej, możemy również użyć pętli typu for-each, która nie wymaga od nas odwoływania się do zmiennej kontrolnej pętli. Jest ona po prostu wywoływana dla każdego obiektu w kolekcji. Takie rozwiązanie możesz zaobserwować w poniższym programie:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

ArrayList<String> library = new ArrayList<String>();

library.add("50 Shades of Grey");

library.add("Sherlock Holmes");

library.add("The Last of the Mohicans");

library.add("The Infinity");

library.add("Better Call Saul");

library.add("Journey to the Center of the Earth");

library.add("Binary Death");

library.add("Breaking Bad");

library.add("Twenty Thousand Leagues Under the Sea");

library.add("The Titanfall");

library.add("Coma");

library.add("Of Ants and Men");

library.add("To Understand a Woman");

library.add("Neverending Story");

library.add("The Jungle Book");

for(String book: library) {

System.out.println("Element of the library is: " + book);

}

}

}

<https://kodilla.com/pl/project-java/38627>

Przypomnijmy sobie na powyższym przykładzie, jak działa pętla for-each. W każdym przebiegu pętli, do zmiennej book typu String wstawiany jest kolejny element z kolekcji library. Następnie używamy zmiennej book do wyświetlenia na ekranie konsoli napisu zawierającego tytuł pozycji książkowej znajdującej się w bibliotece. Zauważ, że nie dysponujemy tutaj wprost numerem indeksu pozycji.

**Usuwanie danych z kolekcji ArrayList**

Aby usunąć z kolekcji jakiś element, należy skorzystać z metody remove(int index). Argumentem tej metody jest numer indeksu obiektu, który ma być usunięty.

**Ważne**

Elementy tablic oraz kolekcji zawsze indeksujemy od 0, a nie od 1.

Przykładowo, aby usunąć ostatni element kolekcji, musimy znać jego indeks. Indeks ten ma wartość wynoszącą library.size()-1 (-1 ponieważ indeksujemy od 0), więc aby usunąć ostatni element kolekcji library, należy użyć polecenia:

library.remove(library.size()-1);

Działanie takiej operacji możesz zobaczyć w poniższym programie. Po usunięciu ostatniego elementu ponownie wyświetlana jest cała zawartość kolekcji, aby upewnić się, że ostatni element rzeczywiście został usunięty.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class RemovingFromArrayList

{

public static void main (String[] args) throws java.lang.Exception

{

ArrayList<String> library = new ArrayList<String>();

library.add("50 Shades of Grey");

library.add("Sherlock Holmes");

library.add("The Last of the Mohicans");

library.add("The Infinity");

library.add("Better Call Saul");

library.add("Journey to the Center of the Earth");

library.add("Binary Death");

library.add("Breaking Bad");

library.add("Twenty Thousand Leagues Under the Sea");

library.add("The Titanfall");

library.add("Coma");

library.add("Of Ants and Men");

library.add("To Understand a Woman");

library.add("Neverending Story");

library.add("The Jungle Book");

System.out.println("Collection contains:");

for(String book: library) {

System.out.println("Element of the library is: " + book);

}

System.out.println("Removing last object from collection.");

library.remove(library.size()-1);

System.out.println("Collection contains:");

for(String book: library) {

System.out.println("Element of the library is: " + book);

}

}

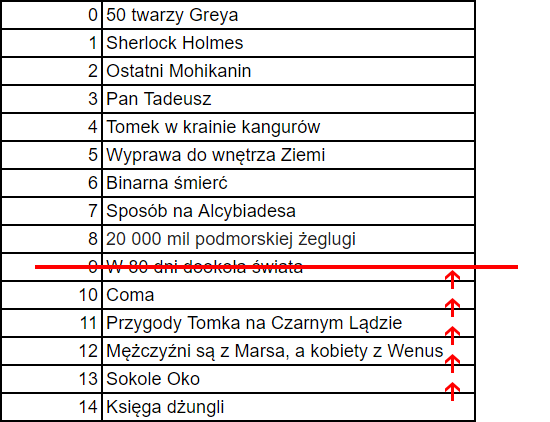
}

<https://kodilla.com/pl/project-java/38628>

**Reorganizacja kolekcji po usunięciu elementu**

W jaki sposób reorganizowana jest tablica, w której kolekcja przechowuje wewnętrznie dane? W przypadku usuwania ostatniego elementu sprawa jest prosta: rozmiar tablicy jest po prostu zmniejszany o jeden.

Bardziej skomplikowana jest sytuacja, gdy chcemy usunąć element znajdujący się w środku kolekcji albo na jej początku. Jeżeli usuwamy obiekt o indeksie *n*, to wszystkie obiekty znajdujące się w kolekcji na dalszych pozycjach (*n*+1, *n*+2, ...) muszą zostać przesunięte o jedno miejsce w górę w wewnętrznej tablicy kolekcji (*n*+1 na miejsce *n*, *n*+2 na miejsce *n*+1, ...). Po zakończeniu przesuwania elementów, rozmiar tablicy jest zmniejszany o jeden. Pomoże Ci to zrozumieć poniższy rysunek:

[](https://uploads.kodilla.com/bootcamp/kodilla-bootcamp-27-06-2018-1530110280-1747.png)

Jak łatwo się domyślić, gdy usuwamy pierwszy element listy, wówczas do skopiowania jest praktycznie cała tablica wewnętrzna listy. Przy dużych listach może to być długotrwały proces.

**Dodawanie danych do kolekcji**

Podobna sytuacja ma miejsce przy dodawaniu elementów do kolekcji. Gdy dodajemy nowy element na końcu kolekcji (tablicy), czas wykonania takiej operacji jest pomijalnie mały.

Jeżeli jednak chcielibyśmy dodać element na pozycji oznaczonej indeksem zero (czyli na początku wewnętrznej tablicy kolekcji) lub gdzieś w środku kolekcji, wówczas wykonanie dodawania obiektu będzie dużo dłuższe, ponieważ operacja ta będzie polegała na stworzeniu nowej tablicy, wypełnieniu w niej elementu z indeksem zero, a następnie skopiowaniu całej dotychczasowej tablicy.

Metoda add(Object o), poza przedstawioną wcześniej postacią, posiada również drugą postać — dwuparametrową add(int index, Object o). W pierwszym parametrze możemy podać numer indeksu, na którym ma zostać umieszczony obiekt w kolekcji. Przykładowo może to wyglądać w następujący sposób:

library.add(0, "Metro 2020");

Wykonanie tej instrukcji spowoduje dodanie do kolekcji obiektu "Metro 2020" na samym początku kolekcji (na pozycji z indeksem 0).

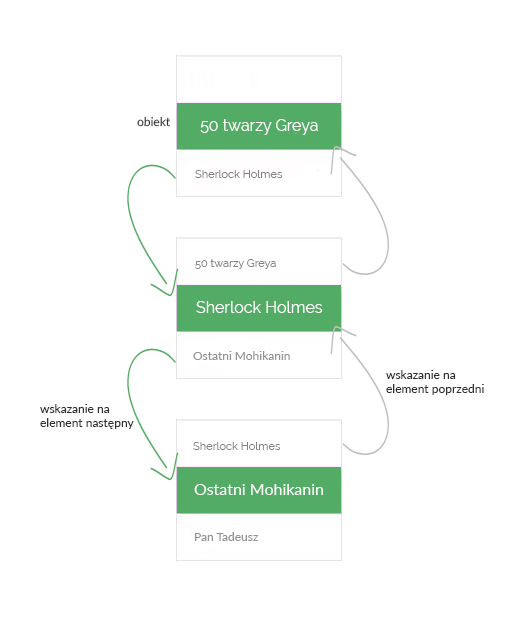
**Podsumowanie ArrayList**

Kiedy warto zastosować kolekcję typu ArrayList? Na pewno jest ona najwygodniejsza przy małych zbiorach danych. Jest ona równocześnie bardzo prosta — działa praktycznie jak tablice, które już znasz.

Nie jest ona jednak rekomendowana, gdy chcesz wstawiać lub usuwać elementy ze środka lub z początku dużej kolekcji, ponieważ operacje te będą się długo wykonywały. W tej sytuacji lepiej sprawdzi się kolekcja listowa typu LinkedList, o której przeczytasz niżej.

**LinkedList**

Innym sposobem przechowywania kolekcji typu List jest lista połączona, czyli kolekcja typu LinkedList. Czym różni się od ArrayList? Najłatwiej jest zobrazować to na rysunku:

[](https://uploads.kodilla.com/bootcamp/kodilla-bootcamp-27-06-2018-1530110607-6476.png)

Kolejne elementy listy LinkedList, poza samym obiektem, pamiętają również referencje (odnośniki) do elementu poprzedzającego oraz następnego. Możesz łatwo to sobie wyobrazić jako pociąg z wagonami — każdy wagon jest podłączony do wagonu poprzedniego oraz do wagonu następnego za nim. Dzięki temu, że z punktu widzenia LinkedList istotne jest tylko to, do jakich "wagonów" jest podłączony "wagon" na którym operujemy, jakiekolwiek operacje dodawania/usuwania/zmiany "wagonów" są wykonywane bardzo szybko — nawet w przypadku dużych kolekcji.

**Odczytywanie danych z listy**

Nową kolekcję typu LinkedList tworzymy podobnie do ArrayList. Spójrz na poniższy kod:

LinkedList<String> theList = new LinkedList<String>();

w powyższej linii tworzony jest nowy obiekt o nazwie theList, będący kolekcją typu LinkedList, którego zawartość tworzy kolekcja obiektów typu String.

Do kolekcji typu LinkedList dodajemy kolejne obiekty tak samo, jak do wcześniej omawianej ArrayList — przy pomocy metody add(Object o). Dodawane w ten sposób obiekty trafiają na koniec listy, rozszerzając ją.

Możemy również dodawać obiekty przy pomocy metody z indeksem, czyli add(int index, Object o) — wówczas w sposób jawny wskazujemy, na której pozycji listy ma być wstawiony nowy obiekt. Spójrz na przykład:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class LinkedListTest

{

public static void main (String[] args) throws java.lang.Exception

{

// Create and fill the list

LinkedList<String> list = new LinkedList<String>();

list.add(0, "Zero");

list.add(1, "One");

list.add(2, "Two");

list.add(3, "Three");

list.add(4, "Four");

list.add(5, "Five");

list.add(6, "Six");

list.add(7, "Seven");

list.add(8, "Eight");

list.add(9, "Nine");

list.add(10, "Ten");

// Retrieve and display an element at index 7

System.out.println("Element list(7) = " + list.get(7));

}

}

<https://kodilla.com/pl/project-java/38630>

Aby odczytać z listy LinkedList element o dowolnym indeksie, korzystamy z takich samych metod, jak w przypadku ArrayList. W tym przypadku będzie to metoda get(int index). Na przykład:

String text = list.get(7);

Wykonanie tej linii programu spowoduje, że do zmiennej o nazwie text (typu String) zostanie przypisana wartość ósmego (pamiętaj, numerujemy od zera) elementu listy o nazwie list, która gdzieś we wcześniejszej części programu została zadeklarowana jako typ LinkedList.

**Indeksy w LinkedList**

Co dzieje się "we wnętrzu" takiej kolekcji? Otóż program musi najpierw ustawić się na pierwszym obiekcie listy, a następnie przechodzić kolejno między obiektami (jak z wagonu na wagon) tak długo, aż osiągnie obiekt o żądanym indeksie. Wyszukanie elementu o dowolnym indeksie będzie więc trwało dłużej, niż w przypadku list tablicowych ArrayList.

**Przypomnienie**

W przypadku list typu tablicowego, aby określić miejsce w pamięci komputera, w którym znajduje się szukany element, i od razu odczytać dane z tego właśnie miejsca pamięci, wystarczy obliczyć, ile miejsca w pamięci zajmuje jedna pozycja, a następnie pomnożyć tę ilość przez numer indeksu.

W przypadku list typu LinkedList należy przejść przez wszystkie obiekty poprzedzające — czyli na przykład dla elementu o indeksie równym 1000 należy przejść przez 1000 elementów listy, aby móc go odczytać.

Wyraźnie widać, że szybkość odczytywania danych z kolekcji LinkedList będzie dużo mniejsza od szybkości odczytywania danych z kolekcji ArrayList. Im większa lista, tym spadek szybkości będzie wyraźniejszy. Przykład odczytu danych z kolekcji LinkedList przedstawiono na poniższym przykładzie:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class LinkedListExample

{

public static void main (String[] args) throws java.lang.Exception

{

//Creat the list and fill the list with 1000 sequent numbers

//multiplied by 2

LinkedList<Integer> theList = new LinkedList<Integer>();

for(int n = 0; n < 1000; n++){

theList.add(n\*2);

}

//Retrieve and display the element of list from position 760

System.out.println("List element at index 760 equals: " + theList.get(760));

}

}

<https://kodilla.com/pl/project-java/38631>

Używając poznanej wcześniej O-notacji, możemy powiedzieć, że złożoność obliczeniowa operacji odczytu danych z kolekcji ArrayList jest rzędu O(c) (czyli jest stała bez względu na to, ile elementów liczy kolekcja, gdyż zawsze trzeba wykonać tylko jedno obliczenie), natomiast złożoność obliczeniowa operacji odczytu danych z kolekcji LinkedList jest rzędu O(n) (czyli jest liniowo zależna od rozmiaru kolekcji, gdyż zawsze trzeba przejść przez wszystkie wcześniejsze obiekty kolekcji).

**LinkedList — dodawanie elementów**

Aby dodać nowy element w środku lub na początku kolekcji LinkedList, należy "rozpiąć" łańcuch elementów składających się na listę, a następnie zmienić wskazania elementów znajdujących się przed i po indeksie, w który wstawiamy nowy element.

Zakładając, że chcemy wstawić nowy element pod indeksem n, musimy:

1. W elemencie poprzedzającym (n-1) zmienić informację o tym, jaki jest kolejny element,
2. a następnie w elemencie (n+1) zmieniamy informację o jego poprzedniku.

Oczywiście te działania są wbudowane w metodę add().

Operacja ta będzie zawsze trwała tak samo długo, bez względu na to, w którym miejscu listy wstawiamy nowy element.

Przykład wstawiania nowego elementu do kolekcji LinkedList możesz obejrzeć na przykładzie poniżej:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class LinkedListExample

{

public static void main (String[] args) throws java.lang.Exception

{

//Create the list and fill it with 1000 sequent numbers

//multiplied by 2

LinkedList<Integer> theList = new LinkedList<Integer>();

for(int n = 0; n < 1000; n++){

theList.add(n\*2);

}

//Retrieve and display element at index 760

System.out.println("List element at index 760 equals: " + theList.get(760));

//insert new element with value 500 at index 760

theList.add(760, 500);

//Checking out the value of element at index 760:

System.out.println("List element at index 760 equals: " + theList.get(760));

//Checking size of the list after addition of one element

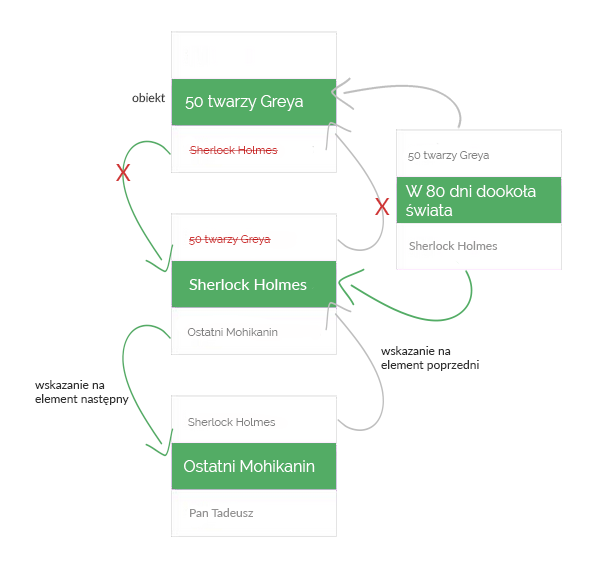
System.out.println("List size (after addition of 1 element) equals: " + theList.size());

}

}

<https://kodilla.com/pl/project-java/38632>

Proces ten można zilustrować następującym rysunkiem:

[](https://uploads.kodilla.com/bootcamp/kodilla-bootcamp-27-06-2018-1530111303-2223.png)

Na rysunkach widzisz usunięcie połączenia między elementami w celu wstawienia w środek nowego elementu listy, a także utworzone nowe połączenia do nowego (wstawionego) elementu listy.

Zwróć uwagę, że po wstawieniu nowego elementu w środku listy otrzyma on nowy indeks, zgodny z tym, jakiego numeru indeksu użyjesz w metodzie add(int index, Object o). Zatem, jak wynika z rysunku powyżej, wstawiany element otrzyma indeks o numerze 2.

**Usuwanie elementów LinkedList**

Podobnie wygląda sytuacja w przypadku, gdy chcemy usunąć jakiś element kolekcji typu LinkedList. Wystarczy zmienić wskazania następnika na element poprzedni, oraz poprzednika na element następny. Na końcu należy usunąć sam, niepowiązany już z niczym, obiekt o wskazanym indeksie.

Operacja ta zawsze trwa tyle samo czasu, bez względu na to, który element listy jest usuwany: pierwszy, ostatni, czy też dowolny w środku listy.

Przykład usuwania dowolnego elementu z kolekcji LinkedList możesz zobaczyć poniżej:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class LinkedListExample

{

public static void main (String[] args) throws java.lang.Exception

{

//Create the list filled with 1000 sequent numbers multiplied by 2

LinkedList<Integer> theList = new LinkedList<Integer>();

for(int n = 0; n < 1000; n++){

theList.add(n\*2);

}

//Retrieve and display element at index 760

System.out.println("List element at index 760 equals: " + theList.get(760));

//Deleting element at index 760

theList.remove(760);

//Checking the value of element at index 760:

System.out.println("List element at index 760 equals: " + theList.get(760));

//Checking length ot the list after deletion

System.out.println("List size (after deletion of 1 element) equals: " + theList.size());

}

}

**Podsumowanie LinkedList**

Kolekcje typu LinkedList najlepiej jest stosować wówczas, gdy odczyty z kolekcji są rzadkie lub dotyczą z reguły pierwszych elementów listy. Warto także zastosować LinkedList w sytuacji, gdy często wstawiamy lub usuwamy obiekty z listy w dowolnych jej miejscach.

**Pętla for i poznane listy**

Listy ArrayList oraz LinkedList są bardzo powszechnie wykorzystywane w programowaniu. Wykonywanie operacji na całych listach mocno ułatwiają pętle, takie jak na przykład omówiona pętla for lub for-each.

Spójrz na program, w którym znajduje się lista zawierająca imiona użytkowników forum. Program ten wyświetla tych użytkowników, których imiona są krótkie, to znaczy mają mniej niż 5 liter:

/\* package whatever; // don't place package name! \*/

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class ListWithConditionExample

{

public static void main (String[] args) throws java.lang.Exception

{

//Create and fill the list with first names

ArrayList<String> userNames = new ArrayList<String>();

userNames.add("Jennifer");

userNames.add("Kimberly");

userNames.add("Nicole");

userNames.add("Amy");

userNames.add("Sarah");

userNames.add("Brandon");

userNames.add("Robert");

userNames.add("Ken");

userNames.add("Jack");

//Displaying that first names, which are no longer than 5 characters

for(String name : userNames){

if(name.length() < 5){

System.out.println(name);

}

}

}

}

<https://kodilla.com/pl/project-java/38633>

**Zadanie: średnia bez skrajnych ocen oraz lista obiektów Book**

Zadanie składa się z dwóch części — jedna dotyczy ArrayList, a druga LinkedList. Obie części należy zrealizować wewnątrz edytora Kodilla. Oba rozwiązania powinny być w ramach jednego projektu.

Część 1

1. Napisz program przy użyciu listy ArrayList, zawierającej kilkanaście ocen z przedmiotu *Informatyka* w liceum.
2. Wypełnij listę przykładowymi ocenami.
3. Następnie przy pomocy pętli for oblicz średnią ocenę ucznia (średnia arytmetyczna), ale w taki sposób, że pominięte zostaną skrajne oceny — JEDNA najmniejsza i JEDNA największa.  
   Oceny największą i najmniejszą należy odnaleźć. Przykładowo, dla ocen:
   1. 3,3,4,4,4,5,5,5,6 — do obliczenia średniej wzięte zostaną oceny: 3,4,4,4,5,5,5
   2. 4,4,4,4,4,4,4 — do obliczenia średniej wzięte zostaną oceny: 4,4,4,4,4

Część 2

1. Napisz program przy użyciu listy LinkedList, zawierającej obiekty opisujące książki. Każda książka powinna posiadać tytuł i rok wydania.
2. Wypełnij listę kilkoma przykładowymi pozycjami.
3. Następnie przy pomocy pętli for-each wyświetl tylko te książki z listy, których rok wydania jest nie starszy niż 2000.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/479788)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162687)

**2.6. Map i HashMap**

**Interfejs Map**

**Co to jest mapa?**

Wspomnieliśmy wcześniej, że kolekcje mogą być również przechowywane w postaci map — do tego celu służy interfejs Map z biblioteki java.util .

Czym jest mapa? Mapa jest strukturą danych przechowującą kolekcję par typu **klucz-wartość** (ang. *key-value*). Pamiętasz, jak we wstępie do modułu z kolekcjami podaliśmy przykład indeksu w bibliotece? Kluczem był numer sygnatury książki (lub inaczej jej numer katalogowy), a wartością (lub inaczej obiektem) był tytuł książki.

**Ważne**

Zarówno klucz (*key*), jak i wartość (*value*) w mapach są obiektami. Nie są to typy prymitywne, jak na przykład indeks w tablicy. Oczywiście jako klucz można zastosować wrapper typu prymitywnego (np. Integer) i wówczas mapa klucz-wartość będzie przypominała tablicę index-wartość, jednak w większości rzeczywistych zastosowań klucz może mieć swoją wewnętrzną strukturę zawierającą właściwości i metody, czyli po prostu będzie obiektem.

Aby lepiej to zrozumieć, wyobraź sobie, że pracujesz w firmie tworzącej oprogramowanie. W aktach osobowych pracowników, oprócz karty z danymi osobowymi (imię, nazwisko, PESEL), znajdują się też różne inne karty (np. wykaz składników wynagrodzenia), które mogą być modyfikowane niezależnie od tej zawierającej dane osobowe.

Ta karta z danymi osobowymi, w kontekście Mapy, reprezentuje obiekt klucz (key). Natomiast przypisana do niej karta będąca wykazem składników wynagrodzenia, odnosząc się do Mapy jest to obiektem reprezentującym wartość (value).

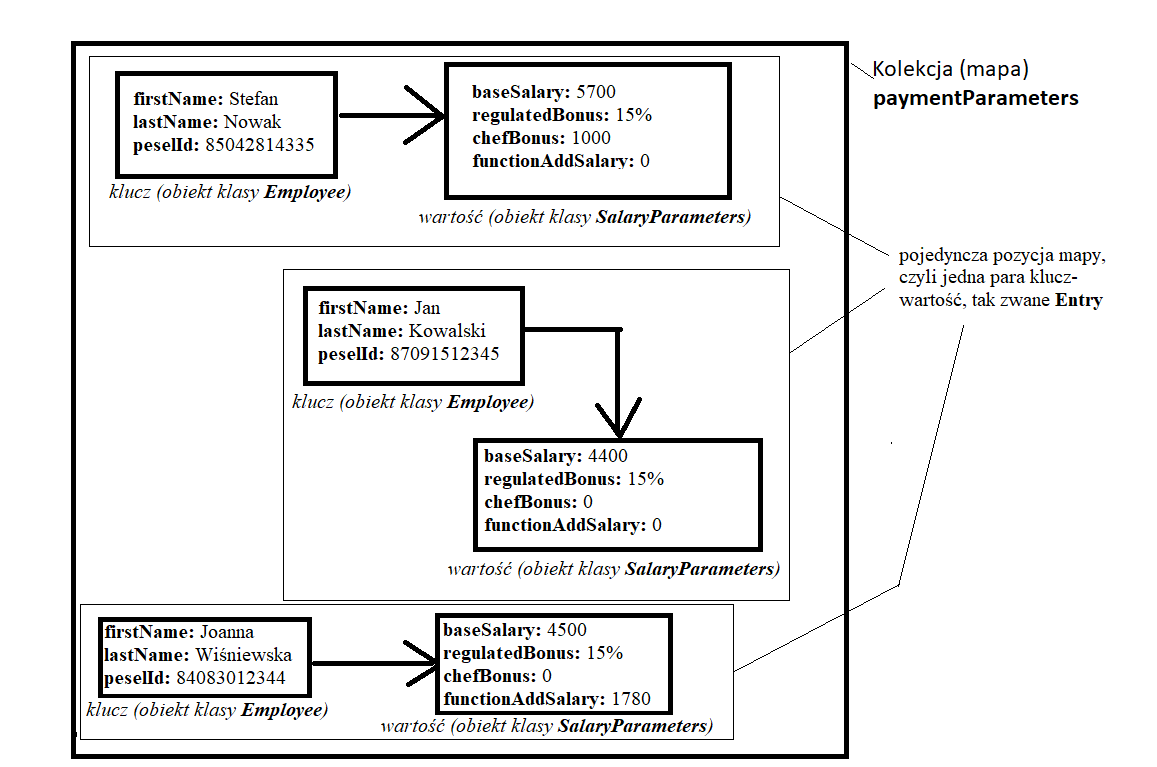
Karta "Składniki wynagrodzenia" zawiera informacje dotyczące wynagrodzenia zasadniczego, premii regulaminowej, dodatku funkcyjnego, premii uznaniowej itp. Gdy otrzymasz podwyżkę, ta karta zostanie zastąpiona nową, zawierającą aktualne wartości parametrów wynagrodzenia, natomiast kartka z Twoimi danymi identyfikacyjnymi nie zmieni się.

Akta osobowe pracowników są więc przykładem mapy typu klucz-wartość. Kluczem w tym przypadku są karty z danymi osobowymi pracowników, wartością natomiast są karty z wynegocjowanymi parametrami wynagrodzenia poszczególnych pracowników.

**Mapa — przykład**

Taką mapę w Javie moglibyśmy nazwać przykładowo paymentParameters (parametry wypłat). Kluczami byłyby obiekty klasy Employee (pracownik), a wartością dla każdego pracownika byłyby obiekty klasy SalaryParameters (parametry wynagrodzenia). Każda para klucz-wartość w tej mapie to tak zwane Entry (obiekt mapy), czyli po polsku "wpis" :)

Spójrz na rysunek poniżej.

[](https://kodilla.com/static/bootcamp-java/java-2_06.png)

Na powyższym rysunku widzisz mapę paymentParameters zawierającą trzy pary klucz-wartość. Klucze są obiektami typu Employee, wartości natomiast są obiektami typu SalaryParameters.

Zwróć uwagę, że na rysunku nie pokazano, jak skonstruowana jest mapa wewnątrz, oraz jakie są powiązania między obiektami (parami — Entry) zawartymi w mapie (pokazano tylko wewnętrzne powiązania w parach klucz -> wartość). Dlaczego? Ponieważ ciągle mówimy o mapie jako czymś ogólnym, bez ustalonych szczegółów implementacji.

**Interfejs Map**

Dla doprecyzowania, omawiamy teraz interfejs Map, określający jedynie kontrakt. Tu pokazujemy tylko teoretycznie, na jakiego rodzaju bytach interfejs ten operuje. Szczegóły dotyczące wewnętrznej budowy mapy zależą od tego, z jaką implementacją interfejsu Map będziemy mieli do czynienia. Jedną z najczęściej używanych implementacji map jest HashMap. Omówimy ją w kolejnej sekcji.

**Przypomnienie**

Interfejsy omawialiśmy w pierwszym module. To, że jakaś klasa implementuje interfejs oznacza, że klasa "zobowiązana jest" do zaimplementowania wszystkich metod (należy napisać ich "ciało") z interfejsu (interfejs zawiera jedynie ich "deklarację", czyli kontrakt). Klasa może dodatkowo wprowadzić swoje własne metody, nie wynikające z implementowanego interfejsu. W tym przypadku klasa HashMap udostępnia implementację wszystkich metod określonych w interfejsie Map, oraz wprowadza kilka dodatkowych.

Poniżej wymieniono kluczowe metody interfejsu Map. Każda implementacja tego interfejsu gwarantuje, że poniższe metody będą dostępne dla programisty:

* put(K key, V value) — wstawia do mapy nową parę klucz-wartość (key-value)
* get(Object key) — zwraca wartość związaną z podanym kluczem key
* remove(Object key) — usuwa z mapy parę wskazywaną przez klucz key
* entrySet() — zwraca całą zawartość mapy w postaci kolekcji nadającej się do użycia w pętli for-each
* size() — zwraca liczbę elementów zawartych w mapie

Nie są to wszystkie metody udostępniane przez interfejs Map. Wystarczą one jednak do naszych zastosowań.

**HashMap**

**Jak działa HashMapa?**

Najczęściej używaną implementacją interfejsu Map jest HashMap (dostarczana przez bibliotekę java.util). Opiera się ona na założeniu, że cała kolekcja par (klucz-wartość) zawartych w mapie dzielona jest na mniejsze zbiory, nazywane "kubełkami" (*ang. buckets*). "Kubełki" odpowiadają kategoriom (podzbiorom) o podobnych cechach.

Odwołując się do wcześniejszego przykładu, pracowników możemy podzielić na kategorie wiekowe: do 20 lat, 21-30 lat, 31-40 lat, 41-50 lat itd. Możemy też wprowadzić podział według stanowisk, płci czy też dowolny inny podział, który jesteśmy w stanie wymyślić. Mimo, że grupowanie obiektów w kubełkach jest możliwe to nie jest to zalecany sposób użycia HashMapy. Wszystkie operacje będą wykonywać się szybciej, gdy w jednym kubełku będzie tylko jedna wartość.

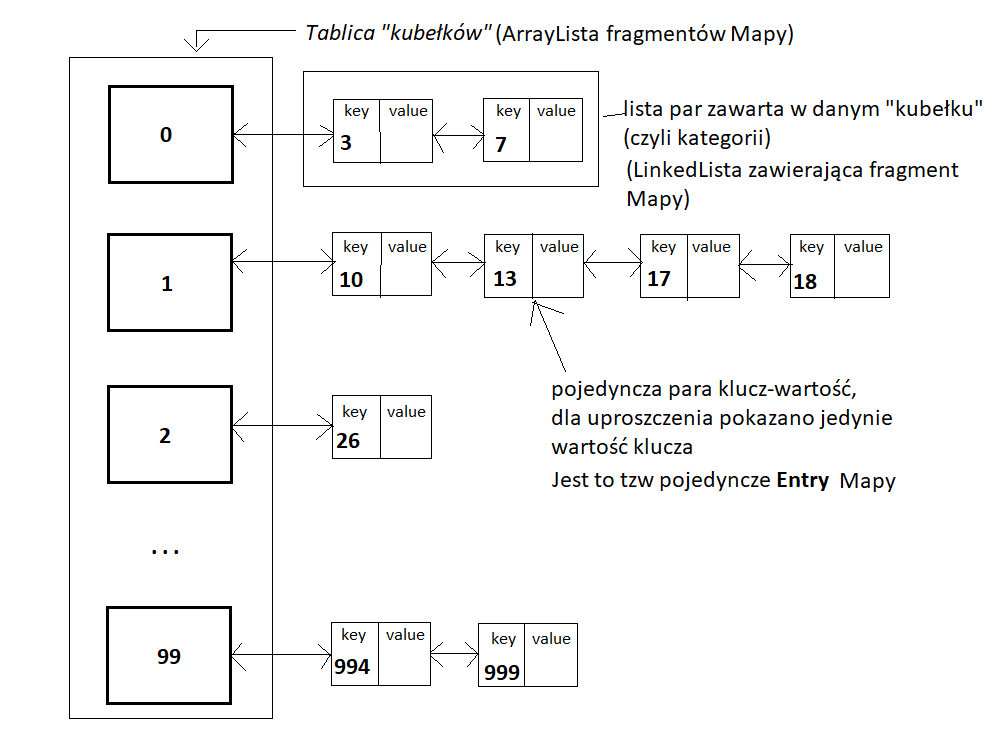
**Ważne**

Wpisy Entry zapisane w mapie dzielone są na kategorie ("kubełki") na podstawie wartości obiektu Key, a nie obiektu Value.

**Czym jest HashMapa?**

HashMapa jest przede wszystkim **Mapą!**

HashMapę najłatwiej jest sobie wyobrazić jako ArrayListę, której elementami są fragmenty Mapy. Z kolei te fragmenty **Map**, w przypadku HashMapy, są [drzewami binarnymi](https://pl.wikipedia.org/wiki/Drzewo_binarne), których elementami są wpisy Entry, czyli nasze pary obiektów klucz-wartość. Poniższa grafika doskonale Ci to zobrazuje :)

[](https://kodilla.com/static/bootcamp-java/java-2_07.png)

Na rysunku przedstawiono, jak w pamięci komputera zorganizowana jest kolekcja typu HashMap, w której klucze par pogrupowano w podzbiory o numerach od 0 do 99. Grupa o numerze 0 przeznaczona jest dla kluczy o wartościach od 0 do 9. W tym konkretnym przypadku, w tej grupie (kubełek 0) mamy obiekty o kluczach: 3 i 7. Grupa o numerze 1 dla kluczy o wartościach od 10 do 19 i tak dalej, aż do grupy o numerze 99 dla kluczy o wartościach od 990 do 999.

Dla uproszczenia przyjęliśmy, że kluczem par jest obiekt typu Integer o zakresie wartości od 0 do 999.

Wracając do grafiki, w pierwszej kolumnie widać tablicę "kubełków" (jest to ArrayLista fragmentów Mapy), czyli kategorii (grup), do jakich klasyfikowane są wpisy Entry (analizowane są klucze, nie wartości). Każdy "kubełek" natomiast posiada referencję (uchwyt, wskaźnik) do głównego węzła drzewa binarnego, które zawiera poszczególne Entry (par obiektów klucz-wartość) należące do kolekcji.

Zapewne zadajesz sobie pytanie: po co ktoś to tak skomplikował i udziwnił? Odpowiedź jest prosta: aby całość działała wydajnie. Implementacja HashMapy zmienia się co jakiś czas, w Javie 8 zostało wprowadzonych [kilka usprawnień](https://www.nagarro.com/en/blog/post/24/performance-improvement-for-hashmap-in-java-8).

Kolekcja typu HashMap jest próbą odpowiedzi na problemy występujące w ArrayList oraz LinkedList. Podzielenie obiektów według kluczy, przykładowo na 100 różnych grup spowoduje, że:

* czas wstawiania obiektu klucza na początku tablicy będzie 100 razy krótszy, niż w standardowej ArrayList,
* wyszukanie wartości znajdującej się na końcu (w najgorszym przypadku) będzie w takiej kolekcji również 100 razy szybsze, niż w standardowej LinkedList, gdyż numer kategorii ("kubełka") zostanie wyznaczony bardzo szybko (tak szybko, jak wyszukiwanie obiektu w uporządkowanej tablicy).

**Porządkowanie elementów HashMapy, czyli kontrakt equals() i hashCode()**

**Metoda hashCode() — podział na "kubełki"**

Zastanawiasz się, jak HashMap dzieli pary zawarte w kolekcji na poszczególne kategorie ("kubełki")? Wykorzystuje do tego metodę hashCode(), obiektu reprezentującego klucz pary. Metoda ta zwraca liczbę (typu int), która w HashMap wykorzystywana jest do wyznaczania numeru "kubełka" dla danego klucza rozpatrywanej pary. Skomplikowane? Już wyjaśniamy :)

Jak zapewne pamiętasz z pierwszego modułu, każda klasa w Javie dziedziczy z klasy Object. Klasa Object posiada już jakąś implementację metody hashCode(), jednak jest ona zbyt prosta, aby można ją było wykorzystać do HashMap (metoda ta zwraca adres obiektu w pamięci skonwertowany do typu int).

Musisz więc w swojej klasie, tworzącej obiekty reprezentujące klucz (key), nadpisać metodę hashCode() tak, aby zwracała numer "kubełka" (kategorii), do której ma być zaklasyfikowany dany obiekt klucz. Reasumując, żeby móc sensownie dzielić na "kubełki", nadpisujesz metodę hashCode() tak, by spełniała Twoje oczekiwania.

**Ważne**

Jeżeli nie nadpiszesz domyślnej (dla klasy Object, z której dziedziczą wszystkie Twoje klasy) implementacji metody hashCode() w swoich obiektach reprezentujących klucz w parach zawartych w HashMap, wówczas mapa taka będzie nieefektywna. Obiekt będziesz mógł dodać, ale nie będziesz mógł go pobrać - przy próbie pobierania zostanie obliczona inna wartość liczbowa. Jak już wspomnieliśmy wszceśniej, w jednym kubełku powinna być jedna wartość. Oznacza to, że metoda hashCode() powinna generować unikalną wartość liczbową.

**Nadpisywanie metody hashCode() w klasie reprezentującej obiekt klucz**

Wyobraźmy sobie, że mamy wiele kluczy typu Integer i chcemy je jakoś pogrupować. Na przykład, do jednego "kubełka" chcemy włożyć te pary, których klucze zawierają się w przedziale wartości od 0 do 9, do drugiego "kubełka" — pary, których klucze zawierają wartość od 10 do 19, a do trzeciego — pary, których klucze mają wartości od 20 do 29 itd. Gdybyśmy mieli klasę MyKey służącą do tworzenia obiektów kluczy w mapie, moglibyśmy to zrobić na przykład tak:

class MyKey {

int value;

//constructor

public MyKey(int value){

this.value = value;

}

//overriden method hashCode():

@override

public int hashCode() {

if(value >= 0 && value <= 9) {

return 0; // bucket 0

}

if(value >= 10 && value <= 19) {

return 1; // bucket 1

}

if(value >= 20 && value <= 29) {

return 2; // bucket 2

}

// etc...

}

W ten sposób otrzymujemy wartość zwracaną przez metodę hashCode() do obiektu klasy HashMap, która wskazuje "kubełek", do którego trafiają obiekty utworzone na podstawie tej klasy.

**Informacja dodatkowa**

Funkcja hashCode() po polsku nazywana jest często *funkcją mieszającą*. Praktycznie wszystkie kolekcje dostępne w bibliotece java.util zawierające słowo hash w nazwie kolekcji korzystają z funkcji mieszających. Oznacza to, że w swoich obiektach zapisywanych w takich kolekcjach musisz nadpisywać domyślną, odziedziczoną z Object metodę hashCode(), aby te kolekcje poprawnie działały z Twoimi obiektami — pamiętając o "kubełkowaniu".

**Unikalność elementów w "kubełkach" — equals()**

W poszczególnych "kubełkach" mapy przechowują unikalne obiekty (sprawdzana jest unikalność obiektów kluczy, nie wartości). Przy próbie wstawienia obiektu, który już istnieje, nie jest on wstawiany (mapa uznaje, że ten obiekt już jest w kolekcji). Gdy mamy już wyznaczony "kubełek", do którego trafi para (Entry), mapa musi sprawdzić, czy dany obiekt nie jest już przypadkiem do mapy dodany.

Spróbuj odpowiedzieć na, proste z pozoru, pytanie: kiedy dwa obiekty są sobie równe?

Omówimy to sobie na przykładzie obiektów reprezentujących pracowników (Employee).

Klasa Employee zdefiniowana jest jak poniżej:

class Employee {

String firstName;

String lastName;

String peselId;

//konstruktor

public void Employee(String firstName, String lastName, String peselid) {

this.firstName = firstName;

this.lastName = lastName;

this.peselId = peselId;

}

}

Na podstawie tej klasy tworzymy obiekty typu Employee. Ich definicję wyobraźmy sobie następująco:

Employee worker1 = new Employee("Zachary", "Taylor", "12345678901");

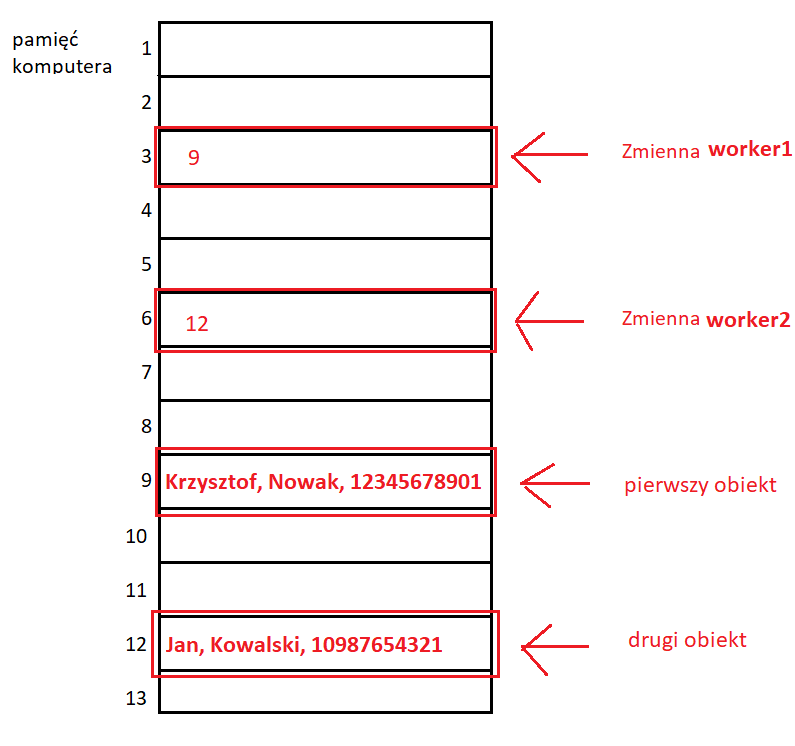
Employee worker2 = new Employee("Elizabeth", "Kowalski", "10987654321");

Aby odpowiedzieć na pytanie, czy zmienne worker1 oraz worker2 są sobie równe, musisz zrozumieć, co tak naprawdę zawierają te zmienne.

**Co tak naprawdę zawierają te zmienne?**

Narzuca się odpowiedź, że zawierają obiekty reprezentujące pracowników. Niestety nie jest to prawdą. Zmienne worker1 oraz worker2 zawierają jedynie **informację o lokalizacji** (adresie w pamięci komputera) obiektów reprezentujących pracowników!

Spójrz na poniższy rysunek.

[](https://kodilla.com/static/bootcamp-java/java-2_08.png)

Na powyższym rysunku przedstawiliśmy Ci (w dużym uproszczeniu), jak wygląda zawartość pamięci komputera, w której zdefiniowano dwie zmienne: worker1 oraz worker2. Zawartość tych zmiennych przechowywana jest w pamięci komputera w komórkach o numerach (adresach) wynoszących odpowiednio: 3 i 6 (jest to duże uproszczenie, ale wystarczy, aby zrozumieć, na czym to polega).

Zawartością tych komórek nie są jednak obiekty typu Employee, a jedynie informacja o tym, w którym miejscu pamięci owe obiekty są przechowywane. Czyli numery komórek pamięci, w których znajdują się rzeczywiście obiekty — odpowiednio: 9 i 12. Dopiero w komórkach o adresach 9 i 12 przechowywane są rzeczywiste obiekty.

**Ważne**

Podczas programowania w Javie nie mówimy, że zmienna ZAWIERA obiekt, mówimy natomiast, że zmienna WSKAZUJE na obiekt w pamięci komputera.

Możesz też spotkać się z pojęciami "zmienna zawiera REFERENCJĘ do obiektu" lub "w zmiennej znajduje się UCHWYT do obiektu".

W powyższym przykładzie, zmienne worker1 oraz worker2 nie są sobie równe. Zmienna worker1 zawiera adres obiektu 9, natomiast zmienna worker2 zawiera adres obiektu 12.

**Stała null**

Przy okazji omawiania zmiennych warto wspomnieć, że zmienne wskazujące na obiekty mogą w danym momencie nie wskazywać żadnego obiektu. Posiadają one wówczas przypisaną wartość null.

Podczas wykonania programu można sprawdzać, czy zmienna ma przypisany obiekt, poprzez porównanie:

if(worker1 != null){

worker1.doBankTransfer();

};

Wartość null oznacza, że do zmiennej nie jest przypisany żaden obiekt.

A co by było w sytuacji, gdyby zarówno obiekt worker1, jak i obiekt worker2 wskazywały na obiekty reprezentujące pracowników o takim samym imieniu, nazwisku oraz numerze PESEL? Mimo że wskazywane przez nie obiekty byłyby takie same, to nie byłyby to **TE SAME** obiekty. Zilustrowano to na poniższym przykładzie:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Employee{

String firstName;

String lastName;

String peselId;

public Employee(String firstName, String lastName, String peselId){

this.firstName = firstName;

this.lastName = lastName;

this.peselId = peselId;

}

}

/\* Name of the class has to be "Main" only if the class is public. \*/

class EqualityTest

{

public static void main (String[] args) throws java.lang.Exception

{

Employee worker1 = new Employee("Zachary", "Taylor", "12345678901");

Employee worker2 = new Employee("Elizabeth", "Kowalski", "10987654321");

System.out.println("worker1 == worker2: " + (worker1 == worker2));

Employee worker3 = new Employee("Monica", "Smith", "99999888887");

Employee worker4 = new Employee("Monica", "Smith", "99999888887");

System.out.println("worker3 == worker4: " + (worker3 == worker4));

}

}

<https://kodilla.com/pl/project-java/38634>

Operator == w przypadku zmiennych reprezentujących obiekty nie porównuje zawartości samych obiektów, a jedynie zawartość zmiennych wskazujących obiekty. Użycie operatora == da więc zawsze wynik równy false przy różnych obiektach — nawet wówczas, gdy będą one takie same (będą miały takie same wartości).

**Ważne**

Jeżeli mamy dwa obiekty o takich samych wartościach atrybutów, to i tak będą to dwa różne obiekty, zlokalizowane pod różnymi adresami (lokalizacjami) w pamięci komputera!

Jak więc powinno się porównywać obiekty? Służy do tego metoda equals(Object o). Metoda ta dostarczana jest do każdej klasy poprzez dziedziczenie z klasy Object. Na potrzebę map, metodę tę trzeba jednak samodzielnie napisać (nadpisując odziedziczoną metodę z klasy Object) dla tworzonych przez siebie klas. A to dlatego, że metoda ta, w swojej dziedziczonej wersji, działa dokładnie tak samo, jako operator ==.

Reasumując, aby sprawdzać czy obiekty są takie same (ich atrybuty mają te same wartości), musisz samodzielnie napisać metodę equals(Object o) w swojej klasie, nadpisując odziedziczoną metodę z klasy Object.

**Przykład własnej wersji metody equals()**

Wracając do naszych obiektów reprezentujących pracowników, metoda equals() dla klasy Employee może wyglądać następująco:

public boolean equals(Object o){

Employee e = (Employee) o;

return this.firstName.equals(e.firstName) && this.lastName.equals(e.lastName) && this.peselId.equals(e.peselId);

}

Omówmy ją dokładnie. W linii pierwszej znajduje się modyfikator dostępu, typ zwracany, nazwa metody oraz przyjmowany typ obiektu wraz z nazwą, której będziemy używać wewnątrz metody. Jedyne słowo, które może się zmienić to o. Reszta musi pozostać bez zmian. W drugiej linii Employee e = (Employee) o; rzutujemy przyjęty argument na typ Employee. Robimy to, aby móc pobrać wartości zmiennych. W klasie Object nie mamy zmiennych firstName i lastName. Rzutowanie pozwala nam traktować obiekt w inny sposób. Co ważne, nie zmieniamy samego obiektu, a jedynie sposób w jaki będzie on traktowany. Przejdźmy do linii trzeciej. Porównujemy w niej wartości. this.firstName odnosi się do obiektu, na którym metoda equals została wywołana. e.firstName oznacza pobranie wartości firstName z obiektu, który został przyjęty w argumencie metody, a następnie zrzutowany na typ Employee.

Sprawdź, jak zmieni się zachowanie prezentowanego chwilę wcześniej programu, gdy użyjesz metody equals(Object o) zamiast operatora ==.

<https://kodilla.com/pl/project-java/38635>

Dopiero teraz jesteśmy w stanie sprawdzić, czy nasze obiekty faktycznie są takie same. A dzięki temu, możemy docelowo sprawdzać unikalność elementów w "kubełkach" :)

**Ważne**

Dlaczego jest to tak ważne? Większość mechanizmów wbudowanych w Javie, w tym kolekcje takie jak HashMap, używają właśnie metody equals(Object o) do sprawdzania, czy obiekty są takie same. Jeżeli nie zaimplementujesz swojej własnej metody equals(Object o) w swoich klasach, wówczas program używający kolekcji Twoich obiektów może zachowywać się w sposób nieprzewidywalny (na przykład może wstawić do mapy dwa razy taką samą parę klucz-wartość, gdyż nie rozpozna, że dany klucz już w mapie istnieje).

**HashMap — podsumowanie**

Działanie kolekcji typu HashMap polega na tym, że zawiera listę (typu ArrayList) grup obiektów, tak zwanych "kubełków" (*ang. buckets*), a w każdym "kubełku" zawiera drzewo binarne kolejnych (różnych) obiektów typu Entry, pasujących do danej kategorii, czyli "kubełka". Ważne — analizowana jest unikalność i kategoria kluczy, a nie wartości!

Kolekcja HashMap, podczas wstawiania obiektu Entry (elementu mapy), używa metody hashCode() (obiektu klucza), aby określić, do którego "kubełka" ma trafić obiekt. Używa także metody equals() (również na obiekcie klucz), do sprawdzenia, czy dany obiekt już jest w danym "kubełku" (wówczas nie jest dodawany po raz drugi — kolekcja traktuje go jako już dodany).

**Tworzenie HashMap**

Spójrz niżej na przykład. Kluczem jest obiekt typu Employee, a funkcja mieszająca grupuje pracowników według roku i miesiąca urodzenia.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Employee{

String firstName;

String lastName;

String peselId;

public Employee(String firstName, String lastName, String peselId){

this.firstName = firstName;

this.lastName = lastName;

this.peselId = peselId;

}

@Override

public boolean equals(Object o){

final Employee e = (Employee) o;

return this.firstName.equals(e.firstName) && this.lastName.equals(e.lastName) && this.peselId.equals(e.peselId);

}

@Override

public int hashCode(){

//This method cuts first 6 (0-5) characters from peselId

//and then converts it into Integer

//peselId includes year and month and day of birth in the first 6 characters

//for example: beginning of peselId 800630 means that persone having this peselId

//was born on 30 july 1980.

//returning value will be 800630 (Integer type)

return Integer.parseInt(peselId.substring(0, 5));

}

}

class SalaryParameters {

Double baseSalary;

Double regulatedBonus;

Double chefBonus;

Double functionAddSalary;

public SalaryParameters(Double baseSalary, Double regulatedBonus, Double chefBonus, Double functionAddSalary){

this.baseSalary = baseSalary;

this.regulatedBonus = regulatedBonus;

this.chefBonus = chefBonus;

this.functionAddSalary = functionAddSalary;

}

@Override

public String toString(){

return baseSalary + " + " + regulatedBonus + " + " +

chefBonus + " + " + functionAddSalary;

}

}

/\* Name of the class has to be "Main" only if the class is public. \*/

class HashMapTest

{

public static void main (String[] args) throws java.lang.Exception

{

Employee worker1 = new Employee("Jennifer", "Nowak", "12345678901");

Employee worker2 = new Employee("Sarah", "Taylor", "12345612345");

Employee worker3 = new Employee("Laura", "Pinelli", "12345698765");

Employee worker4 = new Employee("Laura", "Pinelli", "12345045678");

SalaryParameters params1 = new SalaryParameters(5000.0, 3000.0, 0.0, 0.0);

SalaryParameters params2 = new SalaryParameters(4000.0, 2000.0, 1000.0, 0.0);

SalaryParameters params3 = new SalaryParameters(4500.0, 3500.0, 0.0, 0.0);

SalaryParameters params4 = new SalaryParameters(11000.0, 0.0, 2000.0, 1000.0);

HashMap<Employee, SalaryParameters> paymentParameters = new HashMap<Employee, SalaryParameters>();

paymentParameters.put(worker1, params1);

paymentParameters.put(worker2, params2);

paymentParameters.put(worker3, params3);

paymentParameters.put(worker4, params4);

System.out.println("Salary of Jennifer Nowak equals: " + paymentParameters.get(worker1));

}

}

<https://kodilla.com/pl/project-java/84372>

Przeanalizujmy wywołanie konstruktora mapy:

HashMap<Employee, SalaryParameters> paymentParameters = new HashMap<Employee, SalaryParameters>();

W linijce tej tworzymy obiekt o nazwie paymentParameters, który jest typu HashMap<Employee, SalaryParameters>

Co to oznacza? Oznacza to, że utworzyliśmy właśnie obiekt klasy HashMap, który przechowuje wpisy Entry, w których kluczem jest obiekt klasy Employee, a wartością obiekt klasy SalaryParameters.

**Różnica między mapą, a listą**

Czym różni się mapa od listy?

O części różnic już mówiliśmy, o niektórych jeszcze nie. Wypunktujmy je:

* Listy przechowują obiekty, mapy przechowują **pary** obiektów — jeden z nich nazywamy kluczem, a drugi wartością. Całą parę nazywamy Entry, czyli wpisem.
* Kolejne wpisy do listy dodajemy przy pomocy metody add(Object o), natomiast do mapy przy pomocy metody put(K key, V value).
* Pętli for-each możemy użyć wprost na liście. Aby użyć jej na mapie, musimy skorzystać z właściwości entrySet() mapy.

A teraz dobra wiadomość — to już koniec różnic! No, może poza kilkoma detalami.

**Jakie jest najlepsze zastosowanie HashMap?**

Kolekcje typu HashMap możemy wykorzystywać wszędzie tam, gdzie do jednego obiektu o charakterze bardziej "trwałym" (klucz) chcemy przypisać inny obiekt (wartość) o charakterze bardziej "ulotnym", czy też "zmiennym". Przykładem mogą być tutaj właśnie parametry wynagrodzenia pracowników, o których wspominaliśmy już wcześniej.

**Pętla for i poznane mapy**

Nim przejdziemy do omówienia zastosowań pętli for w mapach, zastanówmy się przez chwilę, jakiego typu są elementy które zawiera mapa. W przypadku ArrayList czy LinkedList typ ten wskazywaliśmy w nawiasach ostrokątnych, np. ArrayList<String>.

W przypadku map mamy jednak do czynienia z dwoma różnymi typami — osobnym dla klucza i osobnym dla wartości, np. HashMap<Integer, String>. Jakiego więc typu jest element kolekcji HashMap, gdy mamy na myśli całą parę?

**Typ Entry**

Typem reprezentującym całą parę jest typ Entry, którego definicja znajduje się w interfejsie Map. Zmienną typu Entry możemy zadeklarować następująco:

Map.Entry<Integer, String> entry = new Map.Entry<Integer, String>();

**Przypomnienie**

Właściwości klas nie powinny być odczytywane lub zapisywane bezpośrednio poprzez wywołanie object.property. Dużo lepiej jest do tego celu zastosować metody dostępowe, czyli tak zwane akcesory (*ang. accessors*). Wyróżniamy osobne metody dostępowe do odczytu (tzw. gettery) i osobne do zapisu (tzw. settery). Przykładowo, dla obiektu o nazwie object i dla jego właściwości o nazwie property należy zdefiniować w klasie dwie metody dostępowe:

* getter o nazwie getProperty() — służący do odczytywania wartości,
* setter o nazwie setProperty(<type> value) — służący do zapisywania wartości.

Jest to istotne, ponieważ wiele frameworków, które będziemy omawiali w dalszej części kursu, będzie oczekiwało od programisty takich właśnie metod dostępowych w jego klasach. Przykład użycia akcesorów możesz zobaczyć poniżej.

<https://kodilla.com/pl/project-java/38638>

Kolekcję HashMap możemy więc potraktować jak kolekcję elementów typu Entry. Typ Entry posiada osobne właściwości: key oraz value. Do zawartości tych pól możemy odwołać się poprzez ich metody dostępowe, w tym gettery, na przykład:

Integer key = entry.getKey();

String value = entry.getValue();

Zbiór wszystkich elementów typu Entry, zawartych w danej kolekcji HashMap, możemy uzyskać przy pomocy metody kolekcji HashMap o nazwie entrySet().

Spójrz na przykładowy program wypełniający kolekcję HashMap wartościami, a następnie za pośrednictwem metody entrySet() oraz metod getKey() i getValue() wyświetlający na ekranie konsoli zawartość kolejnych obiektów kolekcji HashMap.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class EntryTest

{

public static void main (String[] args) throws java.lang.Exception

{

//Creating and filling the map

HashMap<Integer, String> theMap = new HashMap<Integer, String>();

theMap.put(1, "One");

theMap.put(2, "Two");

theMap.put(7, "Seven");

//Using entrySet() to retrieve and display content of the map

for(Map.Entry<Integer, String> entry :theMap.entrySet()){

System.out.println("Object: <" + entry.getKey() + ", " + entry.getValue() + ">");

}

}

}

<https://kodilla.com/pl/project-java/38639>

Jak możemy w praktyce połączyć poznaną wcześniej pętlę for z kolekcjami typu HashMap? Rozwińmy ostatnio napisany program tak, aby wyświetlał wszystkich pracowników wraz z ich parametrami wynagrodzeń:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Employee{

String firstName;

String lastName;

String peselId;

public Employee(String firstName, String lastName, String peselId){

this.firstName = firstName;

this.lastName = lastName;

this.peselId = peselId;

}

@Override

public boolean equals(Object o){

final Employee e = (Employee) o;

return this.firstName.equals(e.firstName) && this.lastName.equals(e.lastName) && this.peselId.equals(e.peselId);

}

@Override

public int hashCode(){

return Integer.parseInt(peselId.substring(0, 5));

}

@Override

public String toString(){

return firstName + " " + lastName + " (" + peselId + ")";

}

}

class SalaryParameters {

Double baseSalary;

Double regulatedBonus;

Double chefBonus;

Double functionAddSalary;

public SalaryParameters(Double baseSalary, Double regulatedBonus, Double chefBonus, Double functionAddSalary){

this.baseSalary = baseSalary;

this.regulatedBonus = regulatedBonus;

this.chefBonus = chefBonus;

this.functionAddSalary = functionAddSalary;

}

@Override

public String toString(){

return baseSalary + " + " + regulatedBonus + " + " +

chefBonus + " + " + functionAddSalary;

}

}

/\* Name of the class has to be "Main" only if the class is public. \*/

class HashMapAndLoopTest

{

public static void main (String[] args) throws java.lang.Exception

{

Employee worker1 = new Employee("Zachary", "Taylor", "12345678901");

Employee worker2 = new Employee("Monica", "Kowalski", "12345612345");

Employee worker3 = new Employee("Robert", "Greenwood", "12345698765");

Employee worker4 = new Employee("Robert", "Greenwood", "12345045678");

SalaryParameters params1 = new SalaryParameters(5000.0, 3000.0, 0.0, 0.0);

SalaryParameters params2 = new SalaryParameters(4000.0, 2000.0, 1000.0, 0.0);

SalaryParameters params3 = new SalaryParameters(4500.0, 3500.0, 0.0, 0.0);

SalaryParameters params4 = new SalaryParameters(11000.0, 0.0, 2000.0, 1000.0);

HashMap<Employee, SalaryParameters> salariesParameters = new HashMap<Employee, SalaryParameters>();

salariesParameters.put(worker1, params1);

salariesParameters.put(worker2, params2);

salariesParameters.put(worker3, params3);

salariesParameters.put(worker4, params4);

for(Map.Entry<Employee, SalaryParameters> entry : salariesParameters.entrySet()) {

System.out.println("Salary of " + entry.getKey() + " equals " +

entry.getValue());

}

}

}

<https://kodilla.com/pl/project-java/38641>

**Dla dociekliwych**

Zwróć uwagę, że w klasie SalaryParameters pojawiła się definicja metody toString(). Metoda o takiej nazwie udostępniana jest przez klasę Object i służy do wyświetlenia obiektu jako tekst (ciąg znaków typu String).

Na przykład:

System.out.println(“To jest obiekt: “ + myObject);

Taka linia kodu spowoduje, że system sięgnie do obiektu myObject i poszuka w nim metody toString(), odpowiedzialnej za skonwertowanie zawartości obiektu do łańcucha znaków. Jeżeli chcesz, aby obiekty Twojej klasy można było "sklejać" z napisami, musisz napisać swoją implementację metody toString() i nadpisać tę, która jest odziedziczona z klasy Object. W przykładowym programie używamy tej metody, aby zaprezentować poszczególne składniki wynagrodzenia w postaci napisu.

Możemy również zmodyfikować program tak, aby wyświetlał jedynie wynagrodzenia tych pracowników, których wynagrodzenie zasadnicze wynosi co najmniej 5 tysięcy złotych:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Employee{

private String firstName;

private String lastName;

private String peselId;

public Employee(String firstName, String lastName, String peselId){

this.firstName = firstName;

this.lastName = lastName;

this.peselId = peselId;

}

@Override

public boolean equals(Object o){

final Employee e = (Employee) o;

return this.firstName.equals(e.firstName) && this.lastName.equals(e.lastName) && this.peselId.equals(e.peselId);

}

@Override

public int hashCode(){

return Integer.parseInt(peselId.substring(0, 5));

}

@Override

public String toString(){

return firstName + " " + lastName + " (" + peselId + ")";

}

}

class SalaryParameters {

private Double baseSalary;

private Double regulatedBonus;

private Double chefBonus;

private Double functionAddSalary;

public SalaryParameters(Double baseSalary, Double regulatedBonus, Double chefBonus, Double functionAddSalary){

this.baseSalary = baseSalary;

this.regulatedBonus = regulatedBonus;

this.chefBonus = chefBonus;

this.functionAddSalary = functionAddSalary;

}

public Double getBaseSalary(){

return baseSalary;

}

@Override

public String toString(){

return baseSalary + " + " + regulatedBonus + " + " +

chefBonus + " + " + functionAddSalary;

}

}

/\* Name of the class has to be "Main" only if the class is public. \*/

class HashMapLoopAndConditions

{

public static void main (String[] args) throws java.lang.Exception

{

Employee worker1 = new Employee("Jennifer", "Taylor", "12345678901");

Employee worker2 = new Employee("Monica", "Kowalski", "12345612345");

Employee worker3 = new Employee("Robert", "Greenfield", "12345698765");

Employee worker4 = new Employee("Robert", "Greenfield", "12345045678");

SalaryParameters params1 = new SalaryParameters(5000.0, 3000.0, 0.0, 0.0);

SalaryParameters params2 = new SalaryParameters(4000.0, 2000.0, 1000.0, 0.0);

SalaryParameters params3 = new SalaryParameters(4500.0, 3500.0, 0.0, 0.0);

SalaryParameters params4 = new SalaryParameters(11000.0, 0.0, 2000.0, 1000.0);

HashMap<Employee, SalaryParameters> salariesParameters = new HashMap<Employee, SalaryParameters>();

salariesParameters.put(worker1, params1);

salariesParameters.put(worker2, params2);

salariesParameters.put(worker3, params3);

salariesParameters.put(worker4, params4);

for(Map.Entry<Employee, SalaryParameters> entry : salariesParameters.entrySet()) {

if (entry.getValue().getBaseSalary() >= 5000){

System.out.println("Salary of " + entry.getKey() + " equals " +

entry.getValue());

}

}

}

}

<https://kodilla.com/pl/project-java/84374>

**Zadanie: mapa uczniów i ocen**

1. Przy użyciu wewnętrznego edytora Kodilli stwórz program z użyciem HashMap, który będzie przechowywał dane uczniów oraz ich ocen.
2. Program ma wyświetlać średnią arytmetyczną ocen każdego ucznia.
3. Rozwiązane zadanie wyślij do mentora.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/479789)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162688)