**3. Kolekcje część druga**

**Wyzwania:**

* poznasz Set i HashSet,
* poznasz Queue,
* wykorzystasz w praktyce pętle for i while.

**Wprowadzenie**

W tym module poznasz kolejne rodzaje kolekcji z biblioteki java.util:

* zbiory implementujące interfejs Set (konkretnie implementację: HashSet),
* kolejki implementujące interfejs Queue, oraz dziedziczący z niego interfejs kolejki dwukierunkowej Deque (konkretnie jedną implementację: ArrayDeque).

W ten sposób poznasz przykładowe kolekcje z wszystkich czterech typów, które są udostępniane przez bibliotekę java.util. Zaczynajmy!

**Przypomnienie**

W poprzednim module zajmowaliśmy się kolekcjami dostarczanymi przez bibliotekę java.util (w ramach czegoś co nazywamy *JCF* – *Java Collections Framework*). Poznaliśmy dwa rodzaje kolekcji:

* listy, implementujące interfejs List (konkretnie dwie różne implementacje list: ArrayList oraz LinkedList),
* mapy, implementujące interfejs Map (konkretnie jedną implementację o nazwie HashMap).

**3.1. Zbiory - Interfejs Set i klasa HashSet**

Jak zapewne pamiętasz, Java udostępnia programistom interfejsy określające metody, które mają zostać umieszczone w implementacjach określonych struktur danych (w klasach, które implementują te interfejsy). Tym razem poznamy interfejs opisujący strukturę danych nazywaną Set (zbiór) oraz klasę implementującą ten interfejs – HashSet.

**Interfejs Set**

Interfejs Set definiuje podstawowe operacje na kolekcjach nazywanych zbiorami. Analogicznie jak poprzednio, interfejs definiuje jedynie **kontrakt**, czyli wskazuje, jakie operacje mają być możliwe do wykonania na obiektach klas implementujących ten interfejs. O tym, jak mają one być zrealizowane, decydują już poszczególne implementacje interfejsu, czyli klasy go implementujące.

Zanim przejdziemy do konkretnych implementacji, przyjrzyj się, jakie ogólne operacje udostępnia interfejs Set. Interfejs ten dziedziczy z interfejsu ogólnego dla kolekcji (czyli interfejsu Collection), zawiera więc wszystkie metody tego interfejsu.

Metody dodatkowo wprowadzane przez Set to między innymi:

* add(E e) – dodaje obiekt typu E do zbioru (jeżeli jeszcze nie jest w zbiorze zawarty),
* clear() – czyści zbiór (usuwa z niego wszystkie obiekty),
* contains(Object o) – sprawdza, czy obiekt jest już w kolekcji. Zwraca true, gdy obiekt znajduje się w zbiorze, i false w przeciwnym wypadku,
* isEmpty() – zwraca true, gdy zbiór jest pusty i nie zawiera żadnego obiektu, w przeciwnym razie zwraca false,
* remove(Object o) – usuwa ze zbioru obiekt o,
* size() – zwraca informację o rozmiarze zbioru (liczbie elementów w zbiorze),
* toArray() – zwraca wszystkie elementy zbioru w postaci tablicy, co umożliwia wykonanie na nich operacji charakterystycznych dla tablic,
* iterator() – zwraca tak zwany iterator zbioru (na razie nie wnikajmy w szczegóły, omówimy to dokładnie w osobnym submodule).

O zbiorach typu Set najłatwiej jest myśleć jak o zbiorach w rzeczywistości. Zbiorem obiektów może być na przykład pudełko z **unikalnymi** klockami. Pudełko zawiera zbiór (to jest właśnie Set), a jego elementami są klocki. Interfejs Set nie mówi nic o tym, jaka jest wewnętrzna struktura elementów w zbiorze. Nieprzezroczyste pudełko ze swobodnie wrzuconymi klockami jest więc dobrym przykładem – nie widać, co jest w środku, wiemy jednak, że są tam unikalne klocki :)

**Ważne**

W ramach jednego zbioru Set nie można przechowywać dwóch takich samych obiektów. Jeżeli dany obiekt znajduje się już w zbiorze, to próba ponownego jego dodania nie wywoła żadnej reakcji.

Równość obiektów sprawdzana jest przy pomocy metody equals() tych obiektów.

W ramach utrwalenia sobie wiedzy, porównaj interfejs Set z poznanym wcześniej interfejsem List. Podobna koncepcja, prawda? Widać, że Set to również kolekcja, tak samo jak List – można do niej dodawać obiekty, usuwać je oraz pobierać, a także sprawdzać jej rozmiar.

**Do czego nam to potrzebne?**

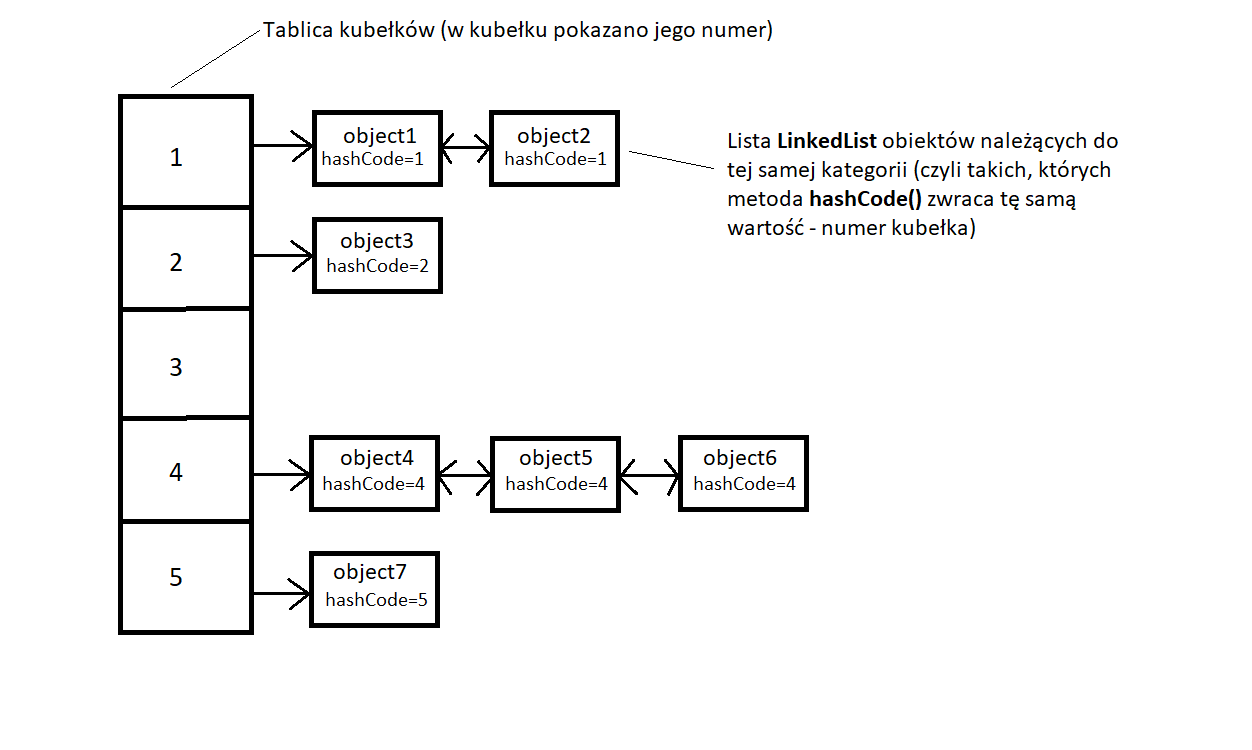
Podczas pracy jako programista, często możesz spotkać się z potrzebą upewnienia się, że dane, które wykorzystujesz w swoim programie, są unikalne. Możesz trafić na taki przypadek podczas pisania aplikacji wypłacającej pracownikom ich miesięczną pensję. Błąd w takiej aplikacji mógłby doprowadzić do sytuacji, w której program wypłaciłby temu samemu pracownikowi jego należność wiele razy, co przecież mogłoby kosztować firmę mnóstwo pieniędzy. Między innymi w takich sytuacjach Set przychodzi z pomocą, pozwalając nam obracać naszymi danymi w nieco bardziej uporządkowany sposób.

Innym przykładem może być system zbierający dane ze stacji meteorologicznych. Wskutek błędów w komunikacji, dane z niektórych czujników mogą dotrzeć do programu kilka razy. Podczas obliczania średniej temperatury powielone dane skutkowałyby niepoprawnymi obliczeniami. Przechowywanie ich w zbiorze Set zagwarantuje, że każdy pomiar znajdzie się w nim tylko raz, dzięki czemu obliczeniom nie grozi błąd.

**Klasa HashSet**

Czas przejść do konkretów, czyli do jednej z implementacji interfejsu Set udostępnianych przez bibliotekę java.util. Będzie to implementacja o nazwie HashSet.

Klasa HashSet, podobnie jak omawiana w poprzednim module mapa HashMap, przechowuje elementy zbioru w postaci kubełków (ang. *buckets*), zawierających listy LinkedList obiektów należących do tej samej kategorii. Kategoria danego obiektu wyznaczana jest przez metodę hashCode() tego obiektu. Różnica polega jedynie na tym, że w HashMap elementami kolekcji były pary klucz-wartość (czyli Entry), natomiast w HashSet elementami kolekcji są po prostu obiekty. Spójrz na poniższy rysunek.

[](https://kodilla.com/static/bootcamp-java/java-3_01.png)

Na powyższym rysunku pokazano zbiór typu HashSet zawierający 7 obiektów (o nazwach od object1 do object7), pogrupowanych na 5 kategorii o numerach od 1 do 5. Numer kategorii (czyli kubełka) dla każdego z obiektów wyznacza wynik wykonania jego metody hashCode().

**Przypomnienie**

Każda klasa dziedziczy w sposób niejawny z klasy Object, która posiada już zaimplementowaną metodę hashCode(). Aby HashSet poprawnie grupował obiekty w kategorie, musisz w swojej klasie nadpisać metodę hashCode() tak, aby zwracała numer kategorii (kubełka) zgodny z Twoim pomysłem na sposób organizacji zbioru HashSet.

**Praktyczne wykorzystanie hashCode()**

Nie koncentrowaliśmy się wcześniej na tym, jak dokładnie powinna działać metoda hashCode() (poza tym, że ma zwracać numer kubełka). Teraz nadszedł czas na to, aby wyraźnie powiedzieć, na co należy zwrócić uwagę podczas pisania własnej metody hashCode():

**Ważne**

Metoda hashCode() Twojej klasy powinna zwracać wartość obliczoną na podstawie wartości pól (czyli właściwości) Twojej klasy. Jeżeli dwa obiekty są sobie równe (w sensie porównania przy pomocy metody equals()), to ich metoda hashCode() musi zwracać taką samą wartość. Metoda hashCode() nie może zwracać wartości uzależnionej od tego, z którą instancją obiektu danej klasy mamy do czynienia, jeżeli obiekty są TAKIE SAME (ale nie TE SAME). Innymi słowy, TAKIE SAME obiekty zwracają TAKĄ SAMĄ wartość przy pomocy metody hashCode() – zapobiega to między innymi wstawianiu duplikatów obiektów do zbioru.

Czas przyjrzeć się zbiorom od strony praktycznej. Poniżej znajdziesz przykładowy program definiujący klasę Employee, która zawiera imię, nazwisko oraz datę urodzenia pracownika.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

import java.time.\*;

/\* Class Employee represents an employee in the company \*/

class Employee {

String firstName;

String lastName;

LocalDate birthDate;

public Employee(String firstName, String lastName, int yearOfBirth,

int monthOfBirth, int dayOfBirth){

this.firstName = firstName;

this.lastName = lastName;

// Class LocalDate provided by java.time library is used

// to store dates. Constructor of(int year, int month, int day) could be

// used to initialize LocalDate variable value

this.birthDate = LocalDate.of(yearOfBirth, monthOfBirth, dayOfBirth);

}

public int hashCode(){

// methods getYear() and getMonthValue() of LocalDate class could be used

// to retrieve a year and month of stored date

return birthDate.getYear() \* 100 + birthDate.getMonthValue();

}

}

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

class HashSetExample

{

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

{

Employee worker1 = new Employee("Steven", "Newmack", 1987, 6, 30);

System.out.println(worker1.hashCode());

}

}

<https://kodilla.com/pl/project-java/38643>

W klasie Employee zaimplementowano metodę hashCode(), która zwraca określoną wartość na podstawie roku i miesiąca urodzenia pracownika (linia 24).

Przykładowo dla osoby urodzonej 30.06.1987 wartość hashCode() wyniesie: 198706 (czyli 1987 \* 100 = 198700. Uzyskaliśmy na końcu dwa miejsca (zera) na wpisanie miesiąca i do tego dodajemy numer miesiąca, czyli 6, co w wyniku daje 198706).

Dzięki takiej metodzie hashCode() wszyscy pracownicy będą podzieleni na kubełki wg roku oraz miesiąca urodzania :)

**Przechowywanie dat w zmiennych**

Daty w języku Java mogą być przechowywane w zmiennych na kilka sposobów. Proponowanym w bootcampie podejściem jest skorzystanie z klasy LocalDate z biblioteki java.time. Aby zapisać datę w zmiennej typu LocalDate, można skorzystać z przypisania wartości do zmiennej przy użyciu “konstruktora statycznego" (w cudzysłowie, ponieważ to zwykła metoda statyczna, która w swoim wnętrzu wywołuje konstruktor, więc nie jest to taki "rasowy" konstruktor, o jakich mówiliśmy, tylko konstruktor opakowany w dodatkową metodę statyczną) o nazwie of klasy LocalDate:

LocalDate storedDay = LocalDate.of(2019, 2, 20);

Powyższa linia kodu tworzy zmienną o nazwie storedDay typu LocalDate i inicjuje jej wartość datą 20 lutego 2019 roku.

Pobieranie wartości poszczególnych składowych daty do zmiennych można zrealizować tak:

int year = storedDay.getYear();

int month = storedDay.getMonthValue();

int day = storedDay.getDayOfMonth();

Zwróć uwagę, że powyższe metody zwracają liczby, a nie zwracają np. nazw miesięcy.

Bibliotekę java.time należy dodać do sekcji zawierającej importy na początku programu (słowo kluczowe import w liniach 3-6 przykładu poniżej). Poinformuje to kompilator, że chcesz używać tej biblioteki.

Bez tego nie będzie on potrafił odnaleźć definicji klasy LocalDate i zgłosi błąd. W zamieszczonym przykładzie omawiany import umieszczono w wierszu nr 6.

Nie przejmuj się jednak – bardziej rozbudowane środowiska programistyczne (np. takie jak to, które zbudujesz w module 4) same podpowiadają programiście, że należy dodać stosowne importy w programie.

**Tworzenie kolekcji HashSet**

Stworzymy teraz pięć przykładowych obiektów klasy Employee, a następnie stworzymy zbiór klasy HashSet i umieścimy w nim utworzone obiekty. Na zakończenie przeprowadzimy mały test, aby zweryfikować, czy wszystko zadziałało jak należy – sprawdzimy, czy jeden z utworzonych uprzednio obiektów znajduje się w zbiorze.

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

import java.util.\*;

import java.lang.\*;

import java.io.\*;

import java.time.\*;

/\* Class Employee represents an employee in the company \*/

class Employee {

String firstName;

String lastName;

LocalDate birthDate;

// Constructor

public Employee(String firstName, String lastName, int yearOfBirth,

int monthOfBirth, int dayOfBirth){

this.firstName = firstName;

this.lastName = lastName;

// Class LocalDate provided by java.time library is used

// to store dates. Constructor of(int year, int month, int day) could be

// used to initialize LocalDate variable value

this.birthDate = LocalDate.of(yearOfBirth, monthOfBirth, dayOfBirth);

}

public int hashCode(){

// methods getYear() and getMonthValue() of LocalDate class could be used

// to retrieve a year and month of stored date

return birthDate.getYear() \* 100 + birthDate.getMonthValue();

}

public boolean equals(Object o){

Employee e = (Employee) o;

return (firstName.equals(e.getFirstName())) &&

(lastName.equals(e.getLastName())) &&

(birthDate.getYear() == e.getBirthDate().getYear()) &&

(birthDate.getMonthValue() == e.getBirthDate().getMonthValue()) &&

(birthDate.getDayOfMonth() == e.getBirthDate().getDayOfMonth());

}

// getters

public String getFirstName(){

return firstName;

}

public String getLastName(){

return lastName;

}

public LocalDate getBirthDate(){

return birthDate;

}

}

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

class HashSetExample

{

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

{

// Creating 5 example Employee objects

Employee worker1 = new Employee("Steven", "Newmack", 1987, 6, 30);

Employee worker2 = new Employee("Jennifer", "Nowak", 1988, 2, 28);

Employee worker3 = new Employee("Robert", "Greenfield", 1987, 6, 15);

Employee worker4 = new Employee("Monica", "Smith", 1984, 10, 18);

Employee worker5 = new Employee("Stephanie", "Jackson", 1985, 4, 3);

// Creating a HashSet and filling it with objects

HashSet<Employee> workersSet = new HashSet<Employee>();

workersSet.add(worker1);

workersSet.add(worker2);

workersSet.add(worker3);

workersSet.add(worker4);

workersSet.add(worker5);

// Testing if example object exists in the set

System.out.println("worker3 exists in the set: " +

workersSet.contains(worker3));

}

}

<https://kodilla.com/pl/project-java/38647>

Zwróć uwagę, że przy wywołaniu konstruktora klasy HashSet (linia 68), w nawiasach ostrokątnych podajemy nazwę klasy, której obiekty będą w kolekcji.

**Operacje na HashSet**

Jak już zapewne wiesz, kolekcje dają nam duże możliwości operowania na danych. Możemy na przykład wyświetlić wszystkich pracowników, używając pętli np. for each. HashSet również daje nam taką możliwość, co widać na przykładzie poniżej.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

import java.time.\*;

/\* Class Employee represents an employee in the company \*/

class Employee {

String firstName;

String lastName;

LocalDate birthDate;

// Constructor

public Employee(String firstName, String lastName, int yearOfBirth,

int monthOfBirth, int dayOfBirth){

this.firstName = firstName;

this.lastName = lastName;

// Class LocalDate provided by java.time library is used

// to store dates. Constructor of(int year, int month, int day) could be

// used to initialize LocalDate variable value

this.birthDate = LocalDate.of(yearOfBirth, monthOfBirth, dayOfBirth);

}

public int hashCode(){

// methods getYear() and getMonthValue() of LocalDate class could be used

// to retrieve a year and month of stored date

return birthDate.getYear() \* 100 + birthDate.getMonthValue();

}

public boolean equals(Object o){

Employee e = (Employee) o;

return (firstName.equals(e.getFirstName())) &&

(lastName.equals(e.getLastName())) &&

(birthDate.getYear() == e.getBirthDate().getYear()) &&

(birthDate.getMonthValue() == e.getBirthDate().getMonthValue()) &&

(birthDate.getDayOfMonth() == e.getBirthDate().getDayOfMonth());

}

public String toString(){

return firstName + ", " + lastName + ", born: " + birthDate;

}

// getters

public String getFirstName(){

return firstName;

}

public String getLastName(){

return lastName;

}

public LocalDate getBirthDate(){

return birthDate;

}

}

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

class HashSetExample

{

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

{

// Creating 5 example Employee objects

Employee worker1 = new Employee("Steven", "Newmack", 1987, 6, 30);

Employee worker2 = new Employee("Jennifer", "Nowak", 1988, 2, 28);

Employee worker3 = new Employee("Robert", "Greenfield", 1987, 6, 15);

Employee worker4 = new Employee("Monica", "Smith", 1984, 10, 18);

Employee worker5 = new Employee("Stephanie", "Jackson", 1985, 4, 3);

// Creating a HashSet and filling it with objects

HashSet<Employee> workersSet = new HashSet<Employee>();

workersSet.add(worker1);

workersSet.add(worker2);

workersSet.add(worker3);

workersSet.add(worker4);

workersSet.add(worker5);

// Print out every worker from the set

for(Employee theWorker: workersSet){

System.out.println(theWorker);

}

}

}

<https://kodilla.com/pl/project-java/38653>

**Przypomnienie**

Metoda toString() dowolnej klasy wykorzystywana jest przez Javę podczas wyświetlania obiektów tej klasy na ekranie konsoli, oraz podczas konkatenacji (łączenia) tekstów. Na przykład:

String variable = "Dane pracownika: " + employee;

Na powyższym przykładzie widzimy konkatenację dwóch obiektów: stałej typu String o wartości "Dane pracownika: " oraz obiektu employee. W celu przedstawienia obiektu employee jako String Java poszukuje metody toString() tego obiektu i ją wywołuje.

Możemy również sprawdzać warunki. Dla przykładu wyświetlimy tych pracowników, którzy urodzili się nie wcześniej niż w 1986 roku:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

import java.time.\*;

/\* Class Employee represents an employee in the company \*/

class Employee {

String firstName;

String lastName;

LocalDate birthDate;

// Constructor

public Employee(String firstName, String lastName, int yearOfBirth,

int monthOfBirth, int dayOfBirth){

this.firstName = firstName;

this.lastName = lastName;

// Class LocalDate provided by java.time library is used

// to store dates. Constructor of(int year, int month, int day) could be

// used to initialize LocalDate variable value

this.birthDate = LocalDate.of(yearOfBirth, monthOfBirth, dayOfBirth);

}

public int hashCode(){

// methods getYear() and getMonthValue() of LocalDate class could be used

// to retrieve a year and month of stored date

return birthDate.getYear() \* 100 + birthDate.getMonthValue();

}

public boolean equals(Object o){

Employee e = (Employee) o;

return (firstName.equals(e.getFirstName())) &&

(lastName.equals(e.getLastName())) &&

(birthDate.getYear() == e.getBirthDate().getYear()) &&

(birthDate.getMonthValue() == e.getBirthDate().getMonthValue()) &&

(birthDate.getDayOfMonth() == e.getBirthDate().getDayOfMonth());

}

public String toString(){

return firstName + ", " + lastName + ", born: " + birthDate;

}

// getters

public String getFirstName(){

return firstName;

}

public String getLastName(){

return lastName;

}

public LocalDate getBirthDate(){

return birthDate;

}

}

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

class HashSetExample

{

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

{

// Creating 5 example Employee objects

Employee worker1 = new Employee("Steven", "Newmack", 1987, 6, 30);

Employee worker2 = new Employee("Jennifer", "Nowak", 1988, 2, 28);

Employee worker3 = new Employee("Robert", "Greenfield", 1987, 6, 15);

Employee worker4 = new Employee("Monica", "Smith", 1984, 10, 18);

Employee worker5 = new Employee("Stephanie", "Jackson", 1985, 4, 3);

// Creating a HashSet and filling it with objects

HashSet<Employee> workersSet = new HashSet<Employee>();

workersSet.add(worker1);

workersSet.add(worker2);

workersSet.add(worker3);

workersSet.add(worker4);

workersSet.add(worker5);

// Print out every worker from the set

for(Employee theWorker: workersSet){

if(theWorker.getBirthDate().getYear() >= 1986){

System.out.println(theWorker);

};

}

}

}

<https://kodilla.com/pl/project-java/38654>

**Korzyści wynikające z używania kolekcji typu HashSet**

Kolekcji Set należy używać wówczas, gdy pracujemy z kolekcjami, do których często zapisujemy nowe obiekty lub zmieniamy istniejące, a także często wyszukujemy obiekty. Dzięki zastosowaniu tablicy mieszającej (tablica podzielona na kubełki), złożoność czasowa operacji wstawiania, zmiany i wyszukiwania w Setach jest prawie zbliżona do stałej (czyli niezależnej od liczby obiektów w kolekcji).

Pamiętasz HashMap z poprzedniego modułu? Kolekcja HashSet jest bardzo podobna. Pod względem wydajności zachowuje się dokładnie tam samo. Różnica między nimi polega na tym, że kolekcja HashSet przechowuje unikalne obiekty, natomiast kolekcja HashMap przechowuje pary typu Entry o postaci klucz-wartość. W przypadku HashMap obiekty reprezentujące wartość nie muszą być unikalne (jednak obiekty reprezentujące klucz muszą być unikalne).

**Zadanie: Filtrowanie zbioru książek**

1. Przy użyciu wewnętrznego edytora Kodilla napisz program, który będzie przechowywał w zbiorze HashSet zestaw kilku obiektów reprezentujących książki.
2. Każda książka powinna mieć tytuł, autora i rok wydania.
3. Nie zapomnij również o zdefiniowaniu metod hashCode() oraz equals() dla swoich książek.
4. Następnie, przy pomocy poznanej w poprzednim module pętli for-each, wyświetl w oknie konsoli tytuły wszystkich książek znajdujących się w zbiorze, które zostały wydane przed 2010 rokiem.
5. Rozwiązane zadanie wyślij do mentora.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/483241)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/163093)

**3.2. Kolejki - Interfejs Queue i klasa ArrayDeque**

W tym submodule zajmiemy się kolejkami. Tę strukturę będzie Ci dosyć łatwo zrozumieć ze względu na to, jak często występuje "w naturze" :) Najpierw przeanalizujmy trzy przykłady kolejek z życia codziennego.

**Kolejka typu FIFO**

Wyobraź sobie kolejkę w sklepie – formuje się ona przy kasie, która "konsumuje elementy kolejki" (czyli obsługuje kolejne osoby). Kolejka posiada głowę (osoba najbliżej kasy) oraz ogon (osoba na końcu kolejki). Jest to kolejka typu **FIFO** (ang. *first in, first out*), co oznacza, że klient wchodzący do kolejki jako pierwszy, również jako pierwszy ją opuści.

**Kolejka typu LIFO**

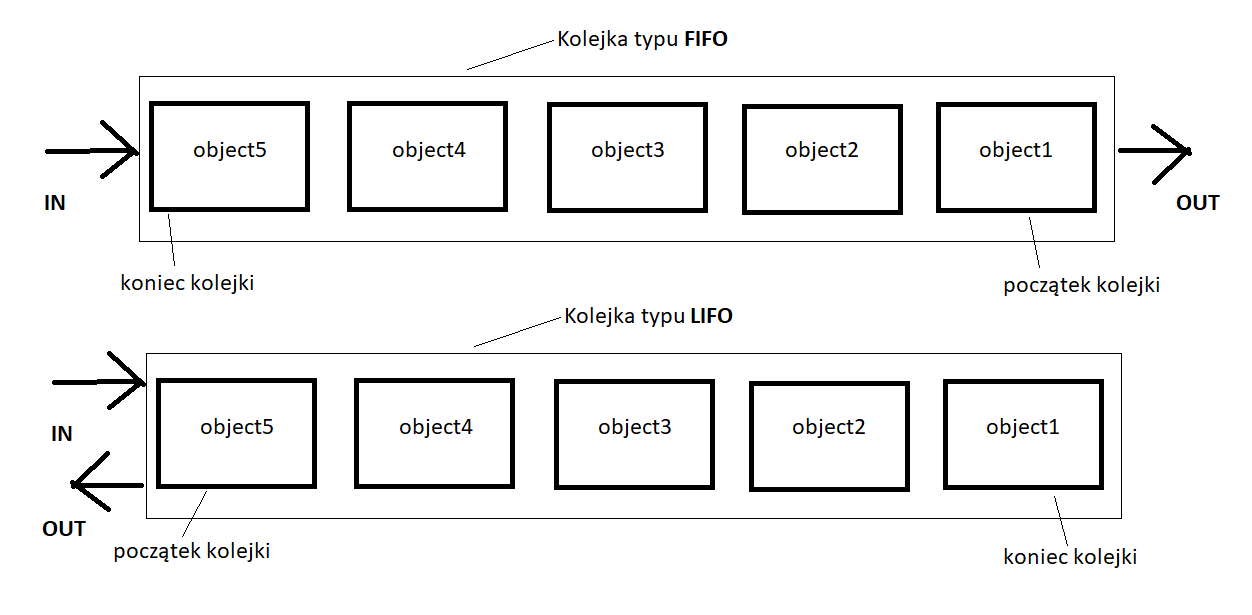
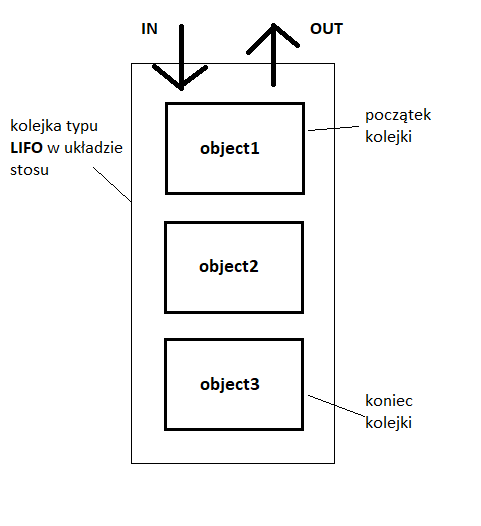
To kolejny przykład kolejki występującej w otaczającej nas rzeczywistości to kolejka w wąskiej windzie. Kolejka taka formuje się wewnątrz windy podczas jej wędrowania pomiędzy piętrami: osoba, która wchodzi do windy jako pierwsza, musi się przesunąć nieco głębiej, aby wpuścić kolejne wchodzące osoby. Podczas wychodzenia z windy kolejność jest odwrócona: osoba, która weszła do windy jako ostatnia, wyjdzie jako pierwsza. Natomiast osoba, która weszła jako pierwsza, wyjdzie ostatnia. Jest to kolejka typu **LIFO** (ang. *last in, first out*) – możesz się również spotkać z nazwą FILO, która oznacza to samo.

**Kolejka LIFO typu STOS**

Pewnie zdarzyło Ci się układać książki w pudle, np. podczas przeprowadzki. Wyobraź sobie wąskie pudło, do którego wkładasz książki, umieszczając je jedna na drugiej. Aby wyjąć książkę włożoną jako pierwszą, musisz najpierw zdjąć z niej wszystkie książki położone później. Ten przykład powinien Ci zobrazować, dlaczego kolejkę **LIFO** nazywamy też często STOSEM (and. *stack*).

**O kolejkach graficznie**

Spójrz na poniższy rysunek, pozwalający łatwiej wyobrazić sobie jak działają kolejki. Zwróć uwagę, że kolejkę **LIFO** oraz kolejkę **LIFO** typu **STOS** odróżnia od siebie jedynie to, że wyobrażamy je sobie "poziomo" lub "pionowo" - poza tym praktycznie niczym się nie różnią.

[](https://kodilla.com/static/bootcamp-java/java-3_02.png)[](https://uploads.kodilla.com/bootcamp/java/03/java-3_03.png)

Nie mówimy na razie nic o tym, jaka jest **wewnętrzna budowa kolejki** – będzie to wynikało z tego, którą jej konkretną implementację wybierzemy. Wrócimy do tego za chwilę.

**Zastosowanie kolejek**

Typowe zastosowanie kolejek **FIFO** związane jest ze współpracą systemów, wykonujących swoje działania z różną szybkością. Aby wolniejszy system nie spowalniał szybszego, często komunikację pomiędzy nimi organizuje się przy pomocy kolejki. Szybszy system wstawia do kolejki obiekty reprezentujące żądania wykonania jakichś działań przez drugi system, drugi natomiast – w swoim tempie – podejmuje obiekty z kolejki i realizuje związane z nimi żądania obsługi.

**Dla dociekliwych**

Komunikacja pomiędzy różnymi systemami realizowana za pośrednictwem kolejek to tak zwana komunikacja **asynchroniczna** (nie wymagająca synchronizacji działań w czasie pomiędzy systemami). Do realizacji takich zadań często wykorzystuje się dedykowane biblioteki obsługujące kolejki, zwane w skrócie **JMS** (ang. *Java Messaging Service*).

Kolejki **FIFO** mogą być również używane wewnętrznie w ramach jednego programu. Obiekt jednej klasy jest wówczas producentem wpisów w kolejce, a obiekt innej klasy jest konsumentem tych wpisów.

Innym przykładem zastosowania kolejek, który na pewno kojarzysz, może być system do obsługi zamówień w restauracjach fast-foodowych. Takie miejsca często mają ekrany (tzw. KDSy) wyświetlające program, który informuje o numerach zamówień, które są obecnie w przygotowaniu lub do odebrania – taki system również może zostać napisany przy użyciu kolejek.

Nieco inne zastosowania mają kolejki typu **LIFO** – używane są wszędzie tam, gdzie należy zapamiętywać jakieś elementy, a następnie odtwarzać je w odwrotnej kolejności. Wyobraź sobie program, który ma przeprowadzić użytkownika przez labirynt – każdy zakręt podczas pokonywanej drogi można zapamiętać w kolejce **LIFO**. Aby z dowolnego miejsca wrócić do punktu startu, wystarczy wyciągać z kolejki **LIFO** zapamiętane informacje o zakrętach w odwrotnej kolejności niż były w kolejce umieszczane i sukcesywnie cofać się aż do punktu rozpoczęcia.

**Dla dociekliwych**

Kolejka **LIFO** w postaci **STOSU** wykorzystywana jest przez wirtualną maszynę Javy podczas wykonywania programu. W momencie gdy program natrafi na wywołanie metody jakiegoś obiektu, zapamiętuje miejsce programu (jego adres w pamięci komputera) w którym aktualnie się znajduje – właśnie w kolejce typu **LIFO**. Następnie przeskakuje do miejsca w pamięci, w którym znajduje się kod wywoływanej metody. Po zakończeniu wykonywania metody pobiera z kolejki **LIFO** wcześniej zapamiętany punkt, z którego nastąpiło wywołanie metody, i skacze w to miejsce z powrotem, aby wznowić i kontynuować wykonanie programu. Takie rozwiązanie pozwala na dowolne zagnieżdżanie wywołań (czyli z wnętrza jednej metody możemy wywołać inną metodę, z której możemy wywołać kolejną metodę itd.)

Przy tego typu operacjach mówimy, że komputer *odkłada na stos adres powrotu*.

**Interfejs Queue**

Podstawowym interfejsem do obsługi kolejek, bez względu na ich konkretną implementację i rodzaj, jest interfejs dostarczany przez bibliotekę java.util o nazwie Queue (pol. *kolejka*). Jak zawsze, interfejs ten jest tylko deklaracją kontraktu, czyli określa, jakie metody muszą być zaimplementowane w konkretnych, implementujących go klasach kolejek. Interfejs Queue – tak jak poprzednio omawiane List, Map oraz Set – również wchodzi w skład *JCF* (*Java Collections Framework*).

Podstawowe operacje wymagane przez interfejs Queue, poza operacjami odziedziczonymi z interfejsu Collection, to:

* offer(E e) – umieszcza obiekt klasy E w kolejce,
* poll() – pobiera i usuwa obiekt z kolejki,
* peek() – pobiera obiekt z kolejki, ale go z niej nie usuwa (w celu sprawdzenia obiektu – na przykład gdy program chce "zajrzeć" do kolejki, aby sprawdzić co go czeka za chwilę, aby np. wyświetlić listę nadchodzących zadań).

**Interfejs Deque**

Sam omówiony przed chwilą interfejs Queue jest bardzo ogólny – nie umożliwia on nawet wskazania, czy wstawianie lub pobieranie elementu ma się odbywać od strony końca, czy też od strony początku kolejki. Dlatego też biblioteka java.util udostępnia nieco bardziej konkretny interfejs, tak zwany **interfejs kolejki dwukierunkowej** o nazwie Deque (dziedziczący, czyli rozszerzający interfejs Queue). Przy pomocy interfejsu Deque możliwe jest już implementowanie klas służących do obsługi zarówno kolejek **FIFO** jak i kolejek LIFO.

Podstawowe operacje udostępniane przez interfejs Deque, poza operacjami odziedziczonymi z Queue oraz Collection, przedstawiono poniżej:

* offerFirst(E e) – dodaje obiekt klasy E na początku kolejki,
* offerLast(E e) – dodaje obiekt klasy E na końcu kolejki,
* pollFirst() – pobiera i usuwa obiekt z początku kolejki,
* pollLast() – pobiera i usuwa obiekt z końca kolejki,
* peekFirst() – pobiera obiekt z początku kolejki bez usuwania go z kolejki (w celu sprawdzenia obiektu i np. wyświetlenia, jaki obiekt jest następny w kolejce),
* peekLast() – pobiera obiekt z końca kolejki bez usuwania go z kolejki (w celu sprawdzenia obiektu i np. wyświetlenia, jaki obiekt jest następny w kolejce).

Klasy realizujące kolejki **FIFO** oraz **LIFO**, które implementują interfejs Deque, wykorzystują następujące jego metody w zależności od rodzaju kolejki:

| **Rodzaj operacji** | **Realizacja kolejki typu FIFO** | **Realizacja kolejki typu LIFO** |
| --- | --- | --- |
| Wstawianie obiektu do kolejki | offerLast(E e) | offerFirst(E e) |
| Pobieranie i usunięcie obiektu z kolejki | pollFirst() | pollFirst() |
| Pobranie obiektu z kolejki bez usuwania obiektu (sprawdzenie obiektu) | peekFirst() | peekFirst() |

Czas zakończyć rozważania teoretyczne i przejść do praktyki. Zajmiemy się teraz przykładową implementacją interfejsu Deque.

**Klasa ArrayDeque**

Klasa ArrayDeque jest jedną z implementacji interfejsu Deque, dostarczaną przez bibliotekę java.util. Jest ona o tyle ciekawa, że za jej pomocą można zrealizować zarówno kolejkę **FIFO** jak i kolejkę **LIFO** (w postaci **stosu**). Na początek utworzymy klasę Order, która będzie reprezentowała zamówienie w księgarni internetowej. Oprócz tego utworzymy klasę Book, reprezentującą książkę. Spójrz na kod poniżej:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

import java.time.\*;

class Book{

String title;

String author;

int year;

// constructor with arguments

public Book(String title, String author, int year){

this.title = title;

this.author = author;

this.year = year;

}

// toString method used for printout the object

public String toString(){

return "Title: \"" + title + "\", author: " + author + ", (" + year + ")";

}

// getters

public String getTitle(){

return title;

}

public String getAuthor(){

return author;

}

public int getYear(){

return year;

}

}

class Order{

Book book;

LocalDate dateOfOrder;

// constructor with arguments

public Order(Book book, int year, int month, int day){

this.book = book;

this.dateOfOrder = LocalDate.of(year, month, day);

}

// method used to printout the order

public String toString(){

return "Order created: " + dateOfOrder + "\n" +

" ordered book: " + book;

}

//getters

public Book getBook(){

return book;

}

public LocalDate getDateOfOrder(){

return dateOfOrder;

}

}

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

class OrdersQueueExample

{

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

{

// Create sample book

Book theBook = new Book("Assassins' Creed the Book", "William Shakespeare", 2016);

// Create sample order

Order theOrder = new Order(theBook, 2017, 6, 30);

// Check what is inside the order

System.out.println(theOrder);

}

}

<https://kodilla.com/pl/project-java/38657>

**Ważne**

Aby w zmiennej typu String umieścić znak cudzysłowu ("), należy podczas wstawiania wartości do zmiennej poprzedzić go znakiem *backslash* (\), czyli wstawić ciąg znaków \".

Dzięki temu kompilator wie, że ten cudzysłów nie kończy ciągu znaków, lecz jest jego elementem (spójrz na powyższy przykład kodu, linia 20).

Analogicznie, aby w zmiennej typu String umieścić znak przejścia do kolejnej linii (czyli potocznie "enter"), należy do tej zmiennej wstawić ciąg znaków \n – symbol specjalny mówiący o tym, że w danym miejscu ma nastąpić przejście do nowego wiersza konsoli.

W następnym kroku będziemy chcieli utworzyć kilka zamówień (obiektów klasy Order) i umieścić je w kolejce **FIFO**. Następnie będziemy chcieli te zamówienia z kolejki pobrać. Nim to zrobimy, spójrzmy jeszcze na to, jakie metody udostępnia klasa ArrayDeque w zależności od tego, czy jest używana w trybie kolejki **FIFO** czy też w trybie **stosu**, czyli kolejki **LIFO.**

Niektóre metody udostępniane przez klasę ArrayDeque w trybie kolejki **FIFO:**

* offer(E e) – wstawia obiekt typu E do kolejki,
* poll() – pobiera i usuwa obiekt z kolejki,
* peek() – pobiera obiekt z kolejki bez usuwania go z kolejki (sprawdzenie obiektu np. w celu wyświetlenia, jaki element jest następny w kolejce),
* size() – zwraca liczbę elementów w kolejce,
* toArray() – zwraca całą kolejkę w postaci tablicy.

Niektóre metody udostępniane przez klasę ArrayDeque w trybie **stosu** (kolejki **LIFO**):

* push(E e) – wstawia element na stos,
* pop() – pobiera i usuwa element ze stosu,
* peek() – pobiera obiekt ze stosu bez usuwania go (sprawdzenie obiektu np. w celu wyświetlenia, jaki element jest następny w kolejce),
* size() – zwraca liczbę elementów na stosie
* toArray() – zwraca całą zawartość stosu w postaci tablicy.

Stos jest bardzo starą strukturą danych – pochodzi z czasów, gdy nie istniały jeszcze języki programowania wysokiego poziomu. Nazwy metod pop() oraz push(E e) pochodzą jeszcze z czasów, gdy wszystko programowało się w assemblerze (języku maszynowym) i są zwyczajowo stosowane do dzisiaj.

**Tworzenie kolejki i wstawianie elementów**

Zmodyfikujmy teraz program z ostatniego przykładu tak, aby wstawić 5 przykładowych zamówień do kolejki, a następnie je z tej kolejki pobrać.

Kolejkę – tak samo jak wszystkie inne kolekcje – tworzymy z użyciem nawiasów ostrokątnych po nazwie klasy, wewnątrz których podajemy nazwę klasy, której obiekty będą przechowywane w kolejce (linia 82 w prykładzie poniżej).

import java.util.\*;

import java.lang.\*;

import java.io.\*;

import java.time.\*;

class Book{

String title;

String author;

int year;

// constructor with arguments

public Book(String title, String author, int year){

this.title = title;

this.author = author;

this.year = year;

}

// toString method used for printout the object

public String toString(){

return "Title: \"" + title + "\", author: " + author + ", (" + year + ")";

}

// getters

public String getTitle(){

return title;

}

public String getAuthor(){

return author;

}

public int getYear(){

return year;

}

}

class Order{

Book book;

LocalDate dateOfOrder;

// constructor with arguments

public Order(Book book, int year, int month, int day){

this.book = book;

this.dateOfOrder = LocalDate.of(year, month, day);

}

// method used to printout the order

public String toString(){

return "Order created: " + dateOfOrder + "\n" +

" ordered book: " + book;

}

//getters

public Book getBook(){

return book;

}

public LocalDate getDateOfOrder(){

return dateOfOrder;

}

}

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

class OrdersQueueExample

{

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

{

// Create 3 sample books

Book theBook1 = new Book("Assassins' Creed the Book", "William Shakespeare", 2016);

Book theBook2 = new Book("Book of jungle", "Rudyard Kipling", 1894);

Book theBook3 = new Book("50 shades of gray - lexicon for graphicians", "Dante Alighieri", 1572);

// Create 5 sample orders

Order theOrder1 = new Order(theBook1, 2017, 6, 28);

Order theOrder2 = new Order(theBook2, 2017, 6, 28);

Order theOrder3 = new Order(theBook1, 2017, 6, 29);

Order theOrder4 = new Order(theBook2, 2017, 6, 30);

Order theOrder5 = new Order(theBook3, 2017, 6, 30);

// Create the FIFO queue

ArrayDeque<Order> theOrders = new ArrayDeque<Order>();

// Check size of the queue

System.out.println("Queue is created. It's size: " + theOrders.size());

// Put the orders into the queue

theOrders.offer(theOrder1);

theOrders.offer(theOrder2);

theOrders.offer(theOrder3);

theOrders.offer(theOrder4);

theOrders.offer(theOrder5);

// Check size of the queue

System.out.println("Queue is filled. It's size: " + theOrders.size());

System.out.println();

// Examine the order at beginning of the queue

System.out.println("First order in the queue is: \n" + theOrders.peek());

System.out.println();

// Get objects from queue

Order temporaryOrder;

temporaryOrder = theOrders.poll();

// here you can do something with order taken from the queue

temporaryOrder = theOrders.poll();

temporaryOrder = theOrders.poll();

temporaryOrder = theOrders.poll();

temporaryOrder = theOrders.poll();

// Check size of the queue

System.out.println("Queue is emptied. It's size: " + theOrders.size());

System.out.println();

// Check what was taken as last

System.out.println("Last order taken from queue was:\n" + temporaryOrder);

System.out.println();

}

}

<https://kodilla.com/pl/project-java/38658>

W jaki sposób można wykorzystać klasę ArrayDeque, aby reprezentowała **stos**? Wystarczy po prostu używać metod push(E e) oraz pop() w miejscu offer(E e) oraz poll(). Cała reszta praktycznie się nie zmienia!

**Zadanie: Książki na stos!**

Przy użyciu wewnętrznego edytora Kodilli napisz program, który:

1. tworzy stos i umieszcza w nim pięć obiektów typu Book (możesz użyć klasy Book z przykładu),
2. sprawdza rozmiar stosu,
3. następnie usuwa wszystkie książki,
4. ponownie sprawdza rozmiar stosu,
5. wyświetla, która książka została usunięta ze stosu jako ostatnia.

Rozwiązane zadanie wyślij do mentora.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/483242)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/163094)

**3.3. Interfejs Iterator oraz pętla while**

**Interfejs Iterator**

Wspominaliśmy wcześniej o tym, że każda kolekcja implementuje interfejs Iterable, dzięki czemu możemy jej używać w pętlach for-each. Przyjrzymy się temu zagadnieniu nieco bliżej.

Interfejs Iterable narzuca, aby każda implementująca go klasa udostępniała klasę implementującą interfejs Iterator. Przykładowo, w klasie ArrayList mamy do dyspozycji metodę o nazwie iterator(), która zwraca obiekt implementujący interfejs Iterator. Za chwilę rozjaśnimy :)

**Czym jest iterator?**

Iterator kolekcji najłatwiej jest wyobrazić sobie jako kursor w edytorze, na dokumencie tekstowym. Na początku, po otwarciu dokumentu, kursor ustawiony jest przed pierwszym znakiem tekstu. Wyobraźmy sobie, że nasz kursor potrafi wykonać dwa polecenia:

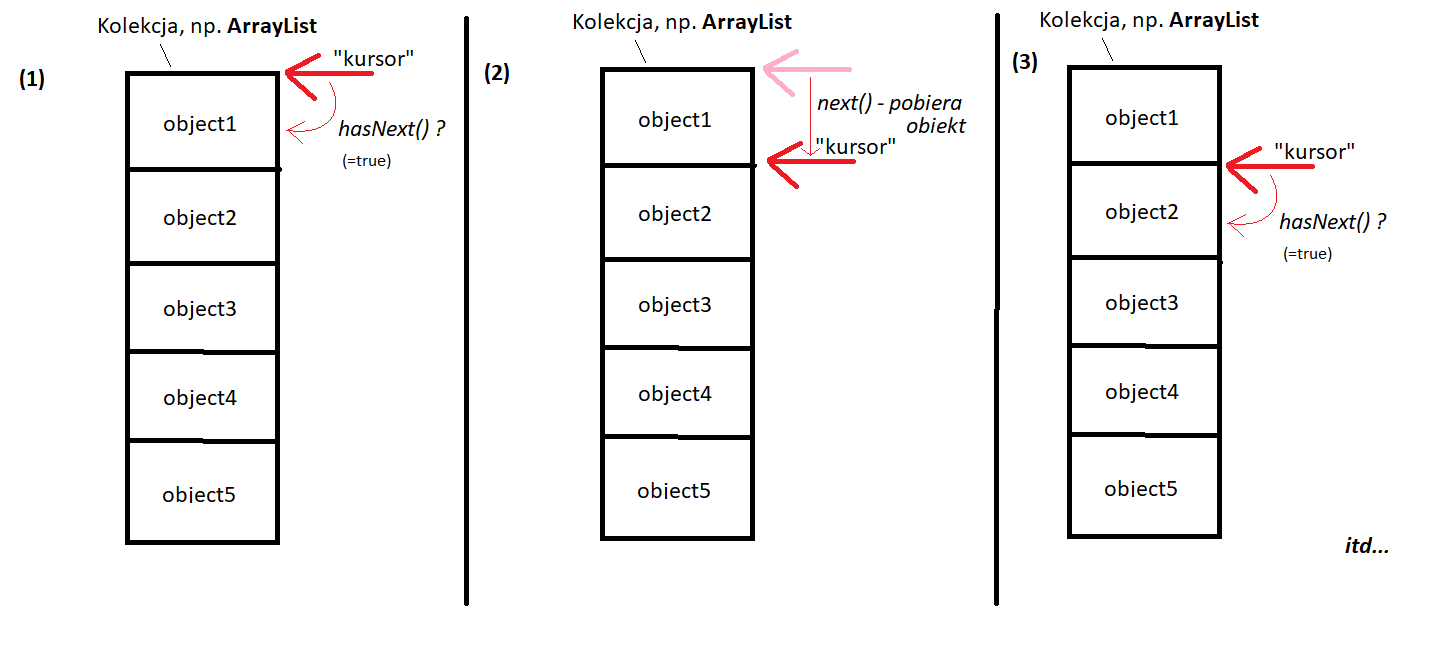
* odpowiedzieć na pytanie, czy z jego prawej strony znajduje się jeszcze jakiś znak, czy też osiągnął już koniec tekstu,
* pobrać znak z prawej strony kursora i przesunąć się o jedno miejsce w prawo.

Możemy sobie wyobrazić, że dysponując tymi dwoma poleceniami jesteśmy w stanie przejść kursorem przez cały tekst. Za każdym razem sprawdzamy, czy po prawej stronie kursora jest jeszcze jakiś znak. Jeżeli tak, odczytujemy ten znak i przesuwamy kursor o jedno miejsce w prawo.

Dokładnie tak działa interfejs Iterator ‐ oczywiście nie operuje on na tekście, a na obiektach zawartych w kolekcji. Interfejs Iterator posiada tak naprawdę kilka metod, ale dwie z nich są najbardziej istotne:

* boolean hasNext() ‐ zwraca true, gdy kolekcja zawiera jeszcze kolejne elementy do odczytania,
* E next() ‐ pobiera z kolekcji wartość kolejnego obiektu typu E i przesuwa "kursor" iteratora o jedno miejsce w prawo.

Jak używać iteratora? Spójrz na rysunek:

[](https://kodilla.com/static/bootcamp-java/java-3_04.png)

Na rysunku widzimy jak działa Iterator. Opiszmy to nieco dokładniej:

1. W pierwszym kroku kursor jest ustawiony przed pierwszym elementem kolekcji.
2. Wywołujemy metodę hasNext(), która zwraca true, ponieważ w kolekcji jest kolejny obiekt do pobrania (rysunek ‐ sekcja **1**).
3. Pobieramy kolejny element z kolekcji przy pomocy metody next(). Powoduje to przesunięcie kursora o jedno miejsce w kolekcji (rysunek ‐ sekcja **2**).
4. Kursor ustawiony jest przed kolejnym obiektem. Możemy więc powtórzyć sprawdzenie, czy jest co pobrać, oraz wykonać pobranie kolejnego obiektu (rysunek ‐ sekcja **3**).
5. Operacje w punktach 2-4 wykonujemy dopóty, dopóki w kolekcji są kolejne elementy do pobrania. Kończymy działanie iteratora dopiero w momencie, gdy metoda hasNext() zwróci wartość false.

Wygląda prosto, prawda? Bo i takie w rzeczywistości jest :) Pętla for-each wykorzystuje wewnętrznie ten właśnie mechanizm dla kolekcji implementujących interfejs Iterable, a więc i interfejs Iterator.

W tym miejscu czas poznać nową pętlę. Dotychczas mieliśmy okazję poznać dwie pętle: pętlę for oraz jej odmianę ‐ pętlę for-each. Teraz czas na pętlę while.

**Pętla while**

Ogólna struktura i składnia pętli while wygląda następująco:

while(<condition>){

<loop\_body>

}

Gdzie:

* <condition> **warunek pętli** ‐ wyrażenie zwracające true lub false (innymi słowy, typu boolean). Pętla wykonuje się dopóty, dopóki zwraca ono wartość true. Gdy <condition> zwróci wartość false, wykonanie pętli jest przerywane i program przechodzi do wykonywania kodu znajdującego się za pętlą,
* **ciało pętli** (<loop\_body>) ‐ instrukcje, które są wykonywane podczas każdej iteracji pętli.

Pętla while wykonuje się tak długo, dopóki warunek pętli (czyli <condition>) zwraca wartość true. Aby niepotrzebnie nie teoretyzować, spójrzmy od razu na przykład:

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

import java.util.\*;

import java.lang.\*;

import java.io.\*;

import java.time.\*;

/\* Class Employee represents an employee in the company \*/

class Employee {

String firstName;

String lastName;

LocalDate birthDate;

// Constructor

public Employee(String firstName, String lastName, int yearOfBirth,

int monthOfBirth, int dayOfBirth){

this.firstName = firstName;

this.lastName = lastName;

// Class LocalDate provided by java.time library is used

// to store dates. Constructor of(int year, int month, int day) could be

// used to initialize LocalDate variable value

this.birthDate = LocalDate.of(yearOfBirth, monthOfBirth, dayOfBirth);

}

public int hashCode(){

// methods getYear() and getMonthValue() of LocalDate class could be used

// to retrieve a year and month of stored date

return birthDate.getYear() \* 100 + birthDate.getMonthValue();

}

public boolean equals(Object o){

Employee e = (Employee) o;

return (firstName.equals(e.getFirstName())) &&

(lastName.equals(e.getLastName())) &&

(birthDate.getYear() == e.getBirthDate().getYear()) &&

(birthDate.getMonthValue() == e.getBirthDate().getMonthValue()) &&

(birthDate.getDayOfMonth() == e.getBirthDate().getDayOfMonth());

}

public String toString(){

return firstName + ", " + lastName + ", born: " + birthDate;

}

// getters

public String getFirstName(){

return firstName;

}

public String getLastName(){

return lastName;

}

public LocalDate getBirthDate(){

return birthDate;

}

}

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

class HashSetExample

{

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

{

// Creating 5 example Employee objects

Employee worker1 = new Employee("Steven", "Newmack", 1987, 6, 30);

Employee worker2 = new Employee("Jennifer", "Nowak", 1988, 2, 28);

Employee worker3 = new Employee("Robert", "Greenfield", 1987, 6, 15);

Employee worker4 = new Employee("Monica", "Smith", 1984, 10, 18);

Employee worker5 = new Employee("Stephanie", "Jackson", 1985, 4, 3);

// Creating a HashSet and filling it with objects

HashSet<Employee> workersSet = new HashSet<Employee>();

workersSet.add(worker1);

workersSet.add(worker2);

workersSet.add(worker3);

workersSet.add(worker4);

workersSet.add(worker5);

// Print out every worker from the set using while loop

Iterator<Employee> iteratorWorkersSet = workersSet.iterator();

while(iteratorWorkersSet.hasNext()){

Employee currentWorker = iteratorWorkersSet.next();

System.out.println(currentWorker);

}

System.out.println("Loop has finished its work");

}

}

<https://kodilla.com/pl/project-java/38659>

Przeanalizujmy, co dzieje się w przedstawionym programie:

1. W wierszu 79 tworzony jest obiekt typu Iterator<Employee> o nazwie iteratorWorkersSet. Przypisywany jest do niego obiekt zwracany przez metodę iterator() kolekcji. Musimy tę czynność wykonać przed pętlą, ponieważ każdorazowe wywołanie metody iterator() w kolekcji powoduje utworzenie nowego iteratora i ustawienie jego kursora przed pierwszym obiektem w kolekcji.
2. W wierszu 80 sprawdzany jest warunek naszej pętli while, czyli wyrażenie iteratorWorkersSet.hasNext(). Zwraca ono wartość true wówczas, gdy za kursorem iteratora znajduje się jakiś obiekt, który można pobrać z kolekcji.
3. W wierszu 81 do zmiennej lokalnej pętli o nazwie currentWorker pobierany jest z kolekcji obiekt reprezentujący pracownika przy pomocy metody next() iteratora. Kursor iteratora przesuwany jest o jedno miejsce ‐ teraz znajduje się przed kolejnym obiektem kolekcji.
4. W wierszu 82 wyświetlane są dane pracownika zapisanego w zmiennej currentWorker, czyli wykonywana jest metoda toString() z klasy Employee dla pracownika zapamiętanego w zmiennej currentWorker.
5. Wykonanie pętli wraca do wiersza 80, gdzie ponownie następuje sprawdzenie, czy w kolekcji są jeszcze jakieś dane, czyli przechodzimy z powrotem do punktu 2 powyżej.
6. W chwili, gdy metoda iteratora hasNext() zwróci wartość false, wykonanie pętli ulega zakończeniu i program przechodzi do instrukcji zawartych bezpośrednio za pętlą ‐ tu, do wyświetlenia komunikatu w wierszu 84.

Pętla while ma bardzo wiele zastosowań ‐ jest wykorzystywana praktycznie równie często, jak pętla for. Typowym jej zastosowaniem (oprócz iterowania po obiektach kolekcji) jest oczekiwanie na wystąpienie jakiegoś zdarzenia.

Wyobraź sobie, że Twój program ma poczekać w pętli, aż do systemu zaloguje się użytkownik. Aby było widać, że program się nie "zawiesił", będzie on informował komunikatem o tym, że oczekuje na zalogowanie się użytkownika oraz poda informację o tym, ilu prób sprawdzenia już dokonał.

**Operatory logiczne w wyrażeniach ‐ przypomnienie**

W trakcie pisania programów nieraz będzie konieczność sprawdzenia warunków złożonych, np. jeżeli a > 10 i b > 10. Albo na przykład jeżeli a > 10 lub (b > 10 i a < 5). Do zapisywania takich wyrażeń wykorzystuje się operatory logiczne:

* && ‐ operator i(ang. *and*). Zwraca wartość true, gdy wyrażenie po lewej stronie, jak i wyrażenie po prawej stronie zwraca true. Na przykład: (a > 10) && (b > 15) ‐ wyrażenie zwróci true, gdy zarówno a będzie większe od 10, jak i b będzie większe od 15. W przeciwnym przypadku zwróci false.
* || ‐ operator lub (ang. *or*). Zwraca wartość true, gdy choć jedno z wyrażeń po lewej i prawej stronie zwraca wartość true (lub gdy oba zwracają true). Na przykład (a > 10) || (b > 15) ‐ wyrażenie zwróci true, gdy a będzie większe od 10 lub b będzie większe od 15. W przeciwnym wypadku zwróci false.
* ! ‐ operator nie (ang. *not*). Zwraca true, gdy wyrażenie po prawej stronie zwraca false oraz zwraca false, gdy wyrażenie jest równe true. Na przykład: !(a > 10) ‐ wyrażenie zwróci true, gdy a nie będzie większe od 10.

Operatorów logicznych używamy najczęściej w wyrażeniach sprawdzających warunki, np. w instrukcji if(). Wynik ich działania możemy również zapamiętywać w zmiennych typu boolean, na przykład:

boolean result = ((a > 5) && (b < 10)) || (a> 100);

Aby zasymulować zdarzenie zalogowania się użytkownika, będziemy losowali liczbę z zakresu 0-99. Jeżeli wylosowana liczba będzie większa od 90, przyjmiemy, że użytkownik się zalogował. Spójrz na przykładowy kod:

import java.util.\*;

class WhileLoopTest {

public static void main(String[] args) {

boolean loggedIn = false;

int tryCounter = 0;

Random randomGenerator = new Random();

while (!loggedIn){

tryCounter++;

System.out.println("Waiting for the User to sign in : [attempt # " + tryCounter + "]");

if (randomGenerator.nextInt(100) > 90) {

loggedIn = true;

};

}

System.out.println("User logged in!");

}

}

<https://kodilla.com/pl/project-java/38660>

Jak działa program?

1. Na początku zmienna jest inicjowana loggedIn na wartość false, a licznik prób o nazwie tryCounter na wartość 0. Tworzony jest też generator liczb pseudolosowych (linie 6 do 8).
2. Program rozpoczyna wykonywanie pętli while. Warunkiem pętli jest wyrażenie !loggedIn (linia 9). Zmienna loggedIn na początku ma wartość false, czyli wyrażenie !loggedIn ma wartość true, a więc warunek pętli jest spełniony i program może wykonać jedną iterację pętli.
3. Zwiększana jest wartość licznika prób oraz wyświetlany jest komunikat o oczekiwaniu na zalogowanie się użytkownika (linie 10 do 11).
4. Następnie generowana jest liczba pseudolosowa z zakresu 0-99 i porównywana jest z ustaloną przez nas wartością graniczną równą 90. Jeżeli wylosowana liczba jest większa od 90, wówczas zmiennej loggedIn przypisujemy wartość true, czyli symulujemy, że użytkownik się zalogował (linie 12 do 14)
5. Program ponownie przechodzi do sprawdzenia warunku pętli (linia 9). Jeżeli nadal jest spełniony, czyli loggedIn = false, to pętla wykona kolejną iterację. Jeżeli natomiast warunek nie jest już spełniony, to program przejdzie do pierwszej linii za pętlą i wyświetli komunikat o tym, że użytkownik się zalogował (linia 16).

**Liczby pseudolosowe**

Często podczas programowania spotkasz się z potrzebą wylosowania jakiejś liczby. W bibliotece java.util znajduje się przeznaczona do tego celu klasa o nazwie Random.

Aby wylosować jakąś liczbę, należy stworzyć generator liczb pseudolosowych, na przykład tak:

Random randomGenerator = new Random();

w linii tej tworzony jest obiekt o nazwie randomGenerator klasy Random. Generator taki udostępnia szereg metod, nas interesuje jednak najprostsza metoda zwracająca wylosowaną liczbę typu int, której wartości mieszczą się w podanym zakresie. Na przykład:

int n = randomGenerator.nextInt(100);

W przykładzie tym generowana liczba jest z zakresu 0-99 (nie większa niż wartość 100 podana w argumencie metody nextInt(int range).

Dlaczego nazywamy generowane w ten sposób liczby "pseudolosowymi"? Otóż mimo naszych najszczerszych chęci, nie są one w pełni losowe. Aby je wygenerować, komputer wykonuje zestaw instrukcji przekazanych mu w kodzie, który z kolei napisał jakiś programista – to eliminuje czynnik pełnej "losowości" z naszych generatorów. Niemniej jednak, losowość wartości generowanych w ten sposób jest zazwyczaj w pełni wystarczająca – poza tym, nie mamy lepszej możliwości.

**Zadanie: Wyścigi sum**

Napisz program, który będzie posiadał:

* zmienną int sumaA (zainicjowana z wartością 1000),
* zmienną int SumaB (zainicjowana z wartością 0),
* pętlę while, wewnątrz której:
  + losujemy liczbę typu int z zakresu 0-9 i dodajemy ją do zmiennej sumaA,
  + losujemy liczbę typu int z zakresu 0-49 i dodajemy ją do zmiennej sumaB.

Pętlę wykonuj tak długo, jak długo spełniony jest warunek sumaA > sumaB.

W każdej iteracji wyświetl numer przebiegu pętli oraz aktualne wartości obu zmiennych.

Rozwiązane zadanie wyślij do mentora.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/483243)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/163095)

**3.4. Przykłady użycia pętli for i while na kolekcjach**

Wiesz już całkiem sporo na temat kolekcji. Znasz też potężne narzędzia programistyczne, jakimi są pętle (for oraz while), a także instrukcja warunkowa if-else. Czas wykorzystać w praktyce zdobytą wiedzę. Niniejszy submoduł będzie w całości poświęcony rozwiązywaniu przykładowych zadań z użyciem pętli oraz kolekcji.

**Generowanie danych testowych przy pomocy pętli for**

Jako pierwszy przykład zadania rozważmy następującą sytuację: wyobraź sobie, że piszesz naprawdę ważny program. Aby mieć pewność, że dobrze działa, musisz przetestować, jak będzie się on zachowywał, gdy otrzyma duży zbiór danych wejściowych typu int w postaci tablicy ArrayList. W tej sytuacji stajesz przed wyzwaniem: jak stworzyć taki zbiór? "Duży" oznacza, że liczba elementów będzie liczona w milionach. Ręcznie nie da się tego zrobić – trzeba zaprząc program do wygenerowania tych danych. Spójrz na poniższy program.

/\* 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 BigTestData

{

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

{

ArrayList<Integer> theBigList = new ArrayList<Integer>();

Random theGenerator = new Random();

for (int n=0; n<2000000; n++){

theBigList.add(theGenerator.nextInt(50000));

}

// Check the list size

System.out.println("The Big List contains now " + theBigList.size() + " objects");

}

}

<https://kodilla.com/pl/project-java/42341>

Prześledźmy zaprezentowany kod linijka po linijce:

1. W linii 12 tworzona jest lista typu ArrayList o nazwie theBigList. Lista będzie zawierała obiekty typu Integer (potrzebujemy co prawda liczb typu int, ale na szczęście poznaliśmy już wrappery).
2. Wiersz 13 to utworzenie oraz zainicjowanie generatora liczb pseudolosowych o nazwie theGenerator przy pomocy klasy Random z biblioteki java.util.
3. W liniach od 14 do 16 wykonywana jest pętla wypełniająca kolekcję danymi. Wykonywanych jest 2 miliony iteracji pętli – w trakcie każdej z nich, dodajemy jeden obiekt do kolekcji.
4. Wewnątrz pętli, w wierszu 15, losowana jest wartość wstawianego do kolekcji obiektu. Obiekty będą miały wartości od 0 do 50 tysięcy minus 1 (nextInt(50000) oznacza liczbę mniejszą (nie większą i nie równą) niż 50 tysięcy) i będą typu int. W wierszu tym obiekty wstawiane są od razu do kolekcji przy pomocy metody add(Object o). Mimo że losowane wartości są typu int, a kolekcja oczekuje obiektów typu Integer, nic złego się nie dzieje – wrapper dba o automatyczną konwersję typu "w locie".
5. Po wypełnieniu kolekcji danymi, sprawdzamy jej rozmiar przy pomocy metody size() udostępnianej przez kolekcję, a następnie go wyświetlamy (linia 18).

**Przekazywanie kolekcji jako argument do metody**

Czas na inny przykład. Twój program zawiera listę wypełnioną liczbami. Chcesz teraz przekazać tę listę do metody innej klasy, która wyświetli na ekranie konsoli tylko te elementy listy, które zawierają wartości parzyste. Spójrz na program:

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

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class ArrayTester{

//method prints out only the even elements of an ArrayList

public void printEven(ArrayList<Integer> theList){

int temporaryValue = 0;

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

temporaryValue = theList.get(n);

//checking if an element is even or odd

if (temporaryValue % 2 == 0){

//if reminder of dividing the element by two equals 0,

//the element is even, so we can print out it.

System.out.println(temporaryValue);

}

}

}

}

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

class CollectionAsParameter

{

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

{

//initialising the collection

ArrayList<Integer> theBigList = new ArrayList<Integer>();

Random theGenerator = new Random();

for(int n=0; n < 200; n++){

//inserting a random number of range 0-99 into the list

theBigList.add(theGenerator.nextInt(100));

};

ArrayTester tester = new ArrayTester();

tester.printEven(theBigList);

}

}

<https://kodilla.com/pl/project-java/38662>

Prześledzimy kod krok po kroku, aby wyjaśnić, jak działa:

1. Działanie programu rozpoczyna się od metody main(String[] args) klasy CollectionAsParameter – linia 27.
2. Tworzona jest kolekcja o nazwie theBigList klasy ArrayList, zawierająca obiekty typu Integer, a także tworzony jest i inicjowany generator liczb pseudolosowych o nazwie theGenerator (linie 30 do 31).
3. Lista theBigList jest wypełniana wartościami pseudolosowymi z zakresu 0-99 przy pomocy pętli for (linie 32 do 35).
4. Tworzony jest obiekt klasy ArrayTester (definicja tej klasy znajduje się w liniach 7 do 22) o nazwie tester.
5. Wywoływana jest metoda printEven(ArrayList<Integer> theList) obiektu tester. Jako argument przekazywana jest kolekcja theBigList.
6. Rozpoczyna się wykonywanie metody printEven(ArrayList<Integer> theList). Jako parametr otrzymuje ona obiekt klasy ArrayList. Wewnątrz metody obiekt ten będzie widoczny pod nazwą theList (linia 10).
7. Wykonywana jest pętla for na całej przekazanej jako argument kolekcji. Rozmiar kolekcji sprawdzany jest w warunku pętli przy pomocy metody size() kolekcji (linie 12 do 20).
8. W każdej iteracji pętli:
   * pobierana jest wartość kolejnego elementu kolekcji do zmiennej lokalnej o nazwie temporaryValue – linia 13,
   * następuje sprawdzenie, czy reszta z dzielenia elementu przez 2 wynosi 0, czyli czy element ma wartość parzystą. Jeżeli tak, wówczas wykonane zostaną instrukcje wewnątrz if() – linia 15,
   * wyświetlana jest wartość sprawdzanego elementu kolekcji.

**Przepisywanie obiektów z kolekcji do kolekcji**

Spróbujmy teraz przedstawić jakiś przykład z pętlą while oraz kolejkami. Wyobraź sobie, że piszesz system obsługujący zadania. Zadania do wykonania są przechowywane w kolejce **FIFO** typu ArrayDeque. Twoim zadaniem jest napisanie klasy, która zajmie się obsługą tych zadań. Każde zadanie po wykonaniu powinno być zapamiętywane w wewnętrznej kolekcji typu ArrayList Twojej klasy, w celu archiwizacji. Spójrz na program poniżej.

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

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class TaskManager {

ArrayList<String> executedTasks;

//constructor

public TaskManager(){

executedTasks = new ArrayList<String>();

}

//method serving the list of tasks

public void executeTasks(ArrayDeque<String> theQueue){

while (theQueue.size() > 0){

String theTask = theQueue.poll();

System.out.println("Processing the task: " + theTask);

executedTasks.add(theTask);

}

System.out.println("\nExecuted total " + executedTasks.size() + " tasks\n");

}

}

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

class QueuesAndLoops

{

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

{

//the first task's list

ArrayDeque<String> taskQueue1 = new ArrayDeque<String>();

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

taskQueue1.offer("The first task number " + (n+1));

}

//the second task's list

ArrayDeque<String> taskQueue2 = new ArrayDeque<String>();

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

taskQueue2.offer("The second task number " + (n+1));

}

TaskManager taskExecutor = new TaskManager();

taskExecutor.executeTasks(taskQueue1);

taskExecutor.executeTasks(taskQueue2);

}

}

<https://kodilla.com/pl/project-java/38663>

Poddajmy powyższy kod analizie:

1. Wykonanie programu rozpoczyna się od metody main(String[] args) klasy QueuesAndLoops (linia 29).
2. Tworzone są dwie kolejki o nazwach taskQueue1 oraz taskQueue2, a następnie są one wypełniane tekstami reprezentującymi zadania (po 5 na kolejkę) – linie 32 do 41.
3. Tworzony jest obiekt taskExecutor klasy TaskManager, której definicja znajduje się w programie w liniach 7 do 24.
4. Wykonywana jest dwukrotnie metoda executeTasks(ArrayDeque theQueue) obiektu taskExecutor - najpierw dla kolejki taskQueue1, a następnie dla kolejki taskQueue2.

Wykonanie metody executeTasks(ArrayDeque theQueue) przebiega następująco:

1. Rozpoczyna się wykonanie metody. Przekazany argument typu ArrayDeque otrzymuje lokalną nazwę theQueue – linia 16.
2. Rozpoczyna się wykonywanie pętli while. Pętla wykonuje się dopóty, dopóki w kolejce znajdują się jakieś zadania do wykonania (warunek: theQueue.size() > 0) – linia 17.
3. W ramach każdej iteracji pętli while:
   * zadanie jest pobierane z kolejki przy pomocy jej metody poll() i jest konwertowane do lokalnej zmiennej theTask typu String – linia 18,
   * wyświetlana jest informacja o wykonaniu zadania – linia 19,
   * wykonane zadanie theTask jest zapisywane w lokalnej kolekcji typu ArrayList – linia 20,
   * po zakończeniu wykonywania pętli, wyświetlana jest informacja o tym, ile zadań wykonano od początku powołania obiektu klasy TaskManager do życia (czyli od utworzenia obiektu).

Warto zwrócić uwagę na fakt, że po wykonaniu zadań z pierwszej kolejki obiekt taskExecutor nie jest tworzony na nowo, lecz jest używany ponownie – tym razem dla kolejnej kolejki zadań. Dzięki temu kolekcja z wykonanymi już zadaniami, przechowywana wewnątrz klasy TaskManager, jest ciągle uzupełniana o kolejne zadania.

**Sprawdzenie LinkedList pod kątem występowania duplikatów**

W ostatnim przykładzie w ramach tego submodułu, użyjemy równocześnie pętli for, pętli while oraz interfejsu Iterator. Naszym zadaniem będzie utworzenie listy LinkedList zawierającej 100 elementów, a następnie wypełnienie tej listy wartościami losowymi z zakresu 0-99 i sprawdzenie, które z wartości powtarzają się wewnątrz listy.

W tym celu stworzymy prostą klasę MyObject, która będzie posiadała tylko pole o nazwie value typu int oraz metodę equals(MyObject o) pozwalającą na porównania. Jak widzisz, nie nadpisaliśmy metody hashCode() i mamy w tym swój cel. Wykorzystamy jej domyślną, dziedziczoną z klasy Object implementację do sprawdzenia, czy porównywane obiekty to różne obiekty – jest to możliwe dzięki temu, że domyślnie hashCode() zwraca adres obiektu w pamięci.

Spójrz na program:

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

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class MyObject {

int value = 0;

//constructor

public MyObject(int value){

this.value = value;

}

public boolean equals(Object o){

MyObject myObj = (MyObject) o;

if (myObj.getValue() == value){

return true;

} else {

return false;

}

}

//getters

public int getValue(){

return value;

}

}

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

class UniqueTest

{

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

{

LinkedList<MyObject> listOfNumbers = new LinkedList<MyObject>();

Random theGenerator = new Random();

//filling the list with random values

for (int n=0; n < 100; n++){

listOfNumbers.add(new MyObject(theGenerator.nextInt(100)));

}

//checking for duplicates

Iterator<MyObject> iterator = listOfNumbers.iterator();

while(iterator.hasNext()){

MyObject examinedObject = iterator.next();

//comparing with every element of list

for (int k=0; k < listOfNumbers.size(); k++){

//preventing to compare the object with himself

if(listOfNumbers.get(k).hashCode() != examinedObject.hashCode()){

//if duplicated...

if(listOfNumbers.get(k).equals(examinedObject)){

System.out.println("Duplicated: " +

examinedObject.getValue());

}

}

}

}

}

}

<https://kodilla.com/pl/project-java/38665>

Dokonajmy analizy działania programu:

1. Działanie programu rozpoczyna się od metody main(String[] args) klasy UniqueTest – linia 34.
2. Tworzona jest lista typu LinkedList o nazwie listOfNumbers, która następnie jest wypełniana setką obiektów typu MyObject z wartością pola value wylosowaną przez generator theGenerator – linie 36 do 41.
3. Tworzony jest iterator i w pętli while przechodzimy po liście tak długo, jak długo są w niej kolejne elementy – linie 44 do 45.
4. Dla każdej iteracji pętli while:
5. Do zmiennej lokalnej examinedObject przypisujemy kolejny element z listy – linia 46.
6. Każdy obiekt porównujemy z wszystkimi obiektami przechowywanymi w liście przy pomocy pętli for – linie 48 do 59.
7. Dla każdego porównywanego obiektu w ramach pętli for:
   * sprawdzamy, czy porównywany obiekt nie jest tym samym, z którym go porównujemy poprzez sprawdzenie, czy hashCode() obiektów są różne – linia 52,
   * sprawdzamy, czy porównywane obiekty są sobie równe pod względem wartości pola value za pomocą metody equals(MyObject o) – linia 55,
   * jeżeli obiekty są sobie równe, wyświetlamy komunikat o znalezieniu duplikatu – linia 56.

**Zadanie: Litery wężykiem!**

Zadanie składa się z dwóch części. Należy je zrealizować przy pomocy wewnęrznego edytora Kodilla w ramach jednego projektu.

**Część 1**

1. Napisz program, który w kolejce ArrayDeque umieści 50 obiektów przechowujących napisy, składające się z litery 'a' powtórzonej losową ilość razy (zakres powtórzeń: 1-50).
2. Wypełnienie obiektu powtórzeniami litery 'a' zrealizuj przy pomocy pętli for.

**Część 2**

1. Program z części pierwszej rozbuduj w taki sposób, że przekażesz utworzoną kolejkę do metody Twojej klasy, która dokona rozdzielenia obiektów z kolejki na dwie kolekcje ArrayList.
2. Jedna z nich będzie przechowywała obiekty o parzystej liczbie znaków 'a', a druga o nieparzystej.

Rozwiązane zadanie wyślij do mentora.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/483244)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/163096)

**3.5. Uzupełnienie - pomiary w kolekcjach**

Jeżeli ciekawi Cię, jak duże są różnice w szybkości działania poszczególnych rodzajów kolekcji, możesz zapoznać się z poniższym materiałem uzupełniającym. Wykonamy w nim pomiar czasu realizacji podstawowych operacji na różnych kolekcjach.

W naszym przykładzie w punkcie dotyczącym kolekcji typu ArrayList lista książek zawierała 15 pozycji – co by było gdyby kolekcja zawierała powiedzmy 30 tysięcy pozycji? Możemy się o tym przekonać w praktyce.

Wpierw przygotujmy listę z 30 tysiącami pozycji. Nie będziemy tego robili "ręcznie" – wykorzystamy do tego celu pętlę for. Nasza lista będzie zawierała liczby, a kolejne jej obiekty będą posiadały wartość równą indeksowi. Skorzystamy z kolekcji ArrayList<Integer> (pamiętaj, że używamy obiektów – czyli typu Integer zamiast typu prymitywnego int).

Program tworzący kolekcję z 30 tysiącami elementów 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 CollectionCreate

{

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

{

//Creating collection with 30000 elements of type Integer

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

for(int n = 0; n < 30000; n++) {

list.add(n);

}

//Displaying quantity of elements in the collection

System.out.println("Quantity of elements in the collections: " + list.size());

}

}

<https://kodilla.com/pl/project-java/38666>

Sprawdźmy teraz, ile czasu zajmuje usunięcie ostatniego elementu kolekcji. Aby to zrealizować, skorzystamy z metody currentTimeMillis() biblioteki System. Dany moment w czasie możemy zapamiętać w zmiennej w następujący sposób:

long momentOfTime = System.currentTimeMillis();

Powyższa operacja zapamięta w zmiennej momentOfTime liczbę milisekund, jaka upłynęła od 1 stycznia 1970 roku.

**Warto wiedzieć**

Jeżeli rozdzielczość czasu w milisekundach nie jest wystarczająco dokładna, można skorzystać z metody zwracającej czas w nanosekundach.

Metoda ta to System.nanoTime(). Przykład jej użycia znajdziesz poniżej:

<https://kodilla.com/pl/project-java/38667>

Aby sprawdzić, jak długo wykonuje się jakaś operacja, wystarczy "zapamiętać" w zmiennej czas systemowy w milisekundach przed wykonaniem operacji oraz po wykonaniu operacji. Różnica tych czasów odpowiada długości wykonania operacji (w milisekundach).

Czas potrzebny na usunięcie ostatniego elementu z kolekcji liczącej 30 tysięcy elementów możemy zbadać przy pomocy poniższego programu:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

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

class ArraySpeedTest

{

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

{

//Creating collection with 30000 elements of type Integer

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

for(int n = 0; n < 30000; n++) {

list.add(n);

}

//Displaying quantity of elements

System.out.println("Quantity of elements in the collection: " + list.size());

//Deleting last element from the collection

long begin = System.currentTimeMillis();

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

long end = System.currentTimeMillis();

//Displaying time of the delete operation

System.out.println("Removing last element has taken: " + (end - begin) + "ms");

}

}

<https://kodilla.com/pl/project-java/38668>

A teraz spróbujemy usunąć pierwszy element kolekcji (z indeksem zero). Program, który realizuje tę operację, przedstawiono poniżej. Tym razem jednak w kolekcji umieścimy 8 milionów elementów i na próbę usuniemy wpierw ostatni, a następnie pierwszy element kolekcji:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

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

class ArraySpeedTest

{

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

{

//Creating collection with 8000000 elements of type Integer

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

for(int n = 0; n < 8000000; n++) {

list.add(n);

}

//Displaying quantity of elements

System.out.println("Quantity of elements in the collection: " + list.size());

//Deleting last element in the collection

long begin = System.currentTimeMillis();

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

long end = System.currentTimeMillis();

//Displaying time of deletion

System.out.println("Removing last element has taken: " + (end - begin) + "ms");

//Deleting first element from collection

begin = System.currentTimeMillis();

list.remove(0);

end = System.currentTimeMillis();

//Displaying time of deletion

System.out.println("Removing first element has taken: " + (end - begin) + "ms");

}

}

<https://kodilla.com/pl/project-java/38669>

Różnica w czasie wykonania jest dość znaczna. Gdyby taki program uruchomić na Twoim komputerze, to jego wykonanie zajęłoby prawdopodobnie kilkadziesiąt sekund (w przypadku usuwania pierwszego elementu listy).

**Przypomnienie**

Operacje wstawiania i usuwania danych z początku lub środka ArrayList wymaga przesunięcia wszystkich pozostałych elementów. Z tego powodu jest ono wolniejsze niż wstawianie i usuwanie elementu z końca ArrayList.

Wyszukiwanie w LinkedList ostatniego elementu wymaga przejścia przez wszystkie elementy listy – dlatego jest wolniejsze niż wyszukanie elementu z początku listy.

Sprawdźmy jeszcze, jak wyglądają czasy potrzebne na wstawienie nowego obiektu do kolekcji liczącej 8 milionów obiektów – w przypadku wstawiania nowego obiektu na końcu listy oraz w przypadku wstawiania nowego obiektu na początku listy (na pozycji z indeksem 0). Do pomiaru czasu skorzystamy tym razem z metody **System.nanoTime()**, która działa analogicznie do metody **System.currentTimeMillis()**, jednak wynik podawany jest w nanosekundach:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class ArrayListSpeedTest

{

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

{

//Creating collection with 8000000 elements of type Integer

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

for(int n = 0; n < 8000000; n++) {

list.add(n);

}

//Displaying quantity of elements

System.out.println("Quantity of elements in the collection: " + list.size());

//Adding element at the end of the collection

long begin = System.nanoTime();

list.add(list.size());

long end = System.nanoTime();

//Displaying time of the operation

System.out.println("Adding element at the od of the collection has taken: " + (end - begin) + "ns");

//Adding element at the beginning of the collection

begin = System.nanoTime();

list.add(0, list.size());

end = System.nanoTime();

//Displaying time of the operation

System.out.println("Adding element at the beginning of the collection has taken: " + (end - begin) + "ns");

}

}

<https://kodilla.com/pl/project-java/84384>

Ponownie widać, że czasy wykonania operacji istotnie się różnią. I znowu należy pamiętać o tym, że ten sam program wykonywany na Twoim komputerze wykonywałby się prawdopodobnie kilkadziesiąt sekund.

**Podsumowanie pomiarów w kolekcjach**

Omawiane typy kolekcji mogą być często stosowane zamiennie. Jednak w przypadku większych zbiorów danych warto zwrócić uwagę na to, które typy kolekcji lepiej się sprawdzą dla poszczególnych operacji.

I tak:

ArrayList posiada złożoność obliczeniową:

* wynoszącą O(n) dla operacji wstawiania i usuwania
* wynoszącą O(c) dla operacji wyszukiwania po indeksie

LinkedList posiada złożoność obliczeniową:

* wynoszącą O(c) dla operacji wstawiania i usuwania
* wynoszącą O(n) dla operacji wyszukiwania po indeksie

HashMap posiada złożoność obliczeniową zależną od sposobu implementacji metody hashCode():

* wynoszącą O(n/c) dla operacji wstawiania i usuwania
* wynoszącą również O(n/c) dla operacji wyszukiwania (gdzie stała c zależy od tego, na ile (w miarę równolicznych) kategorii podzielony został zbiór obiektów przez metodę hashCode().)

HashSet posiada złożoność obliczeniową zależną od sposobu implementacji metody hashCode():

* wynoszącą O(n/c) dla operacji wstawiania i wyszukiwania
* wynoszącą również O(n/c) dla operacji wyszukiwania (gdzie stała c zależy od tego, na ile (w miarę równolicznych) kategorii podzielony został zbiór obiektów przez metodę hashCode().)

ArrayDeque posiada złożoność obliczeniową:

* wynoszącą O(n) dla operacji wstawiania i usuwania
* wynoszącą O(c) dla operacji odczytu bez usuwania

Jeżeli wiemy, że operacją dominującą będzie wyszukiwanie albo wstawianie i usuwanie, to lepiej zastosować ArrayList (gdy najczęściej odczytujemy) lub LinkedList (gdy najczęściej modyfikujemy). Jeżeli natomiast modyfikacje będą tak samo częste jak odczyty z kolekcji, wówczas najefektywniejszym rodzajem kolekcji będzie HashSet lub HashMap.

**Zadanie: Pomiar szybkości LinkedList oraz HashMap**

Zadanie składa się z dwóch części. Jedna dotyczy LinkedList, a druga HashMap. Oba programy napisz w wewnętrznym edytorze Kodilla i zapisz je jako jeden projekt.

**Część 1**

Stwórz klasę reprezentującą książkę o nazwie Book. Klasa powinna mieć dwa pola: author oraz title. Pamiętaj o implementacji metod hashCode() oraz equals(Object o). Będziemy jej używali jako obiektów kolekcji LinkedList w tej części zadania, oraz jako obiektów kolekcji HashMap w drugiej części zadania.

Stwórz program, który zmierzy czas operacji wyszukiwania i usunięcia obiektu na początku (z użyciem metody remove(Object o)), wyszukiwania i usunięcia obiektu na końcu (z użyciem metody remove(Object o)), wstawiania na początku oraz wstawiania na końcu listy LinkedList liczącej kilkaset tysięcy obiektów.

**Część 2**

Stwórz program, który zmierzy czas operacji wyszukiwania po kluczu, a także czas dodawania i usuwania obiektu z mapy HashMap liczącej kilkaset tysięcy obiektów. Postaraj się, aby kluczem w mapie nie była wartość liczbowa.

Rozwiązane zadanie wyślij do mentora.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/483245)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/163097)