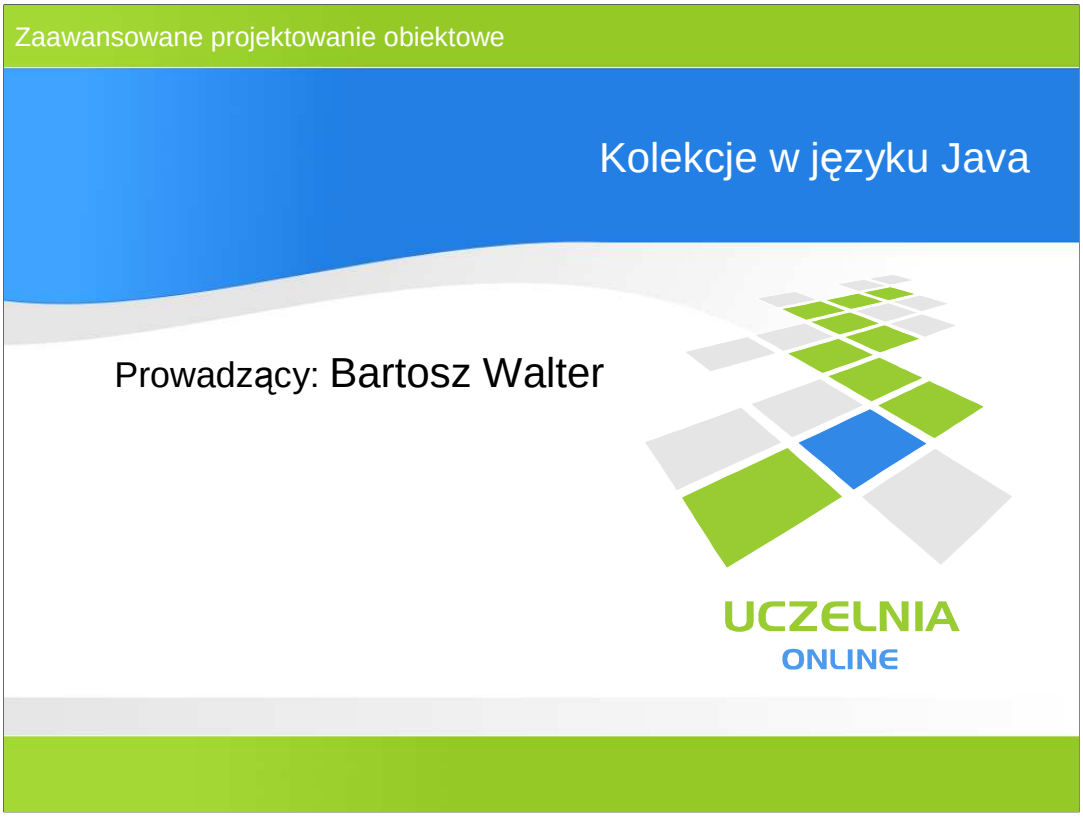
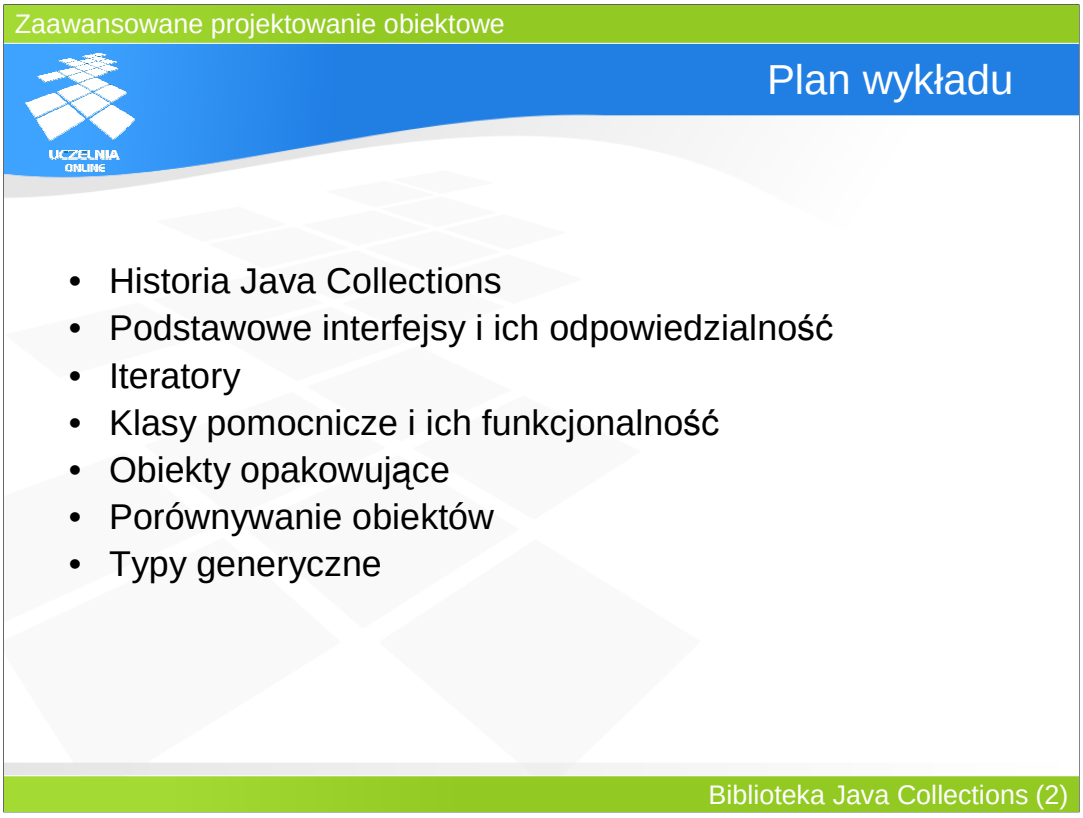

Zaawansowane projektowanie obiektowe
Kolekcje w języku Java
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE



Podczas wykładu zostanie zaprezentowana biblioteka Java Collections. Stanowi ona przykład ewolucji biblioteki obiektowej wraz z pojawiającymi się nowymi elementami języka. Spośród innych bibliotek wyróżnia ją wysoka jakość projektu oraz dobór zastosowanych rozwiązań.

Omówiona zostanie pokrótce historia ewolucji tej biblioteki i wprowadzane do niej w kolejnych wersjach zmiany. Następnie przedstawione będą podstawowe interfejsy wchodzące w skład biblioteki, ich odpowiedzialność oraz konwencje i ograniczenia, jakie nakładają. Potem zaprezentowane zostaną iteratory stanowiące abstrakcyjny mechanizm dostępu do elementów kolekcji i ich implementacja w tej bibliotece. Tematem kolejnej części wykładu będą klasy pomocnicze, zawierające metody, które nie należą do żadnego typu kolekcji, a pozwalające na wykonywanie na nich pewnych wspólnych operacji. Na szczególną uwagę zasługuje koncepcja obiektów opakowujących, które pozwalają dynamicznie zmieniać zachowanie klasy. Ostatnim zagadnieniem będą typy generyczne wprowadzone do języka Java w wersji 5.0 i ich zastosowania.

Historia

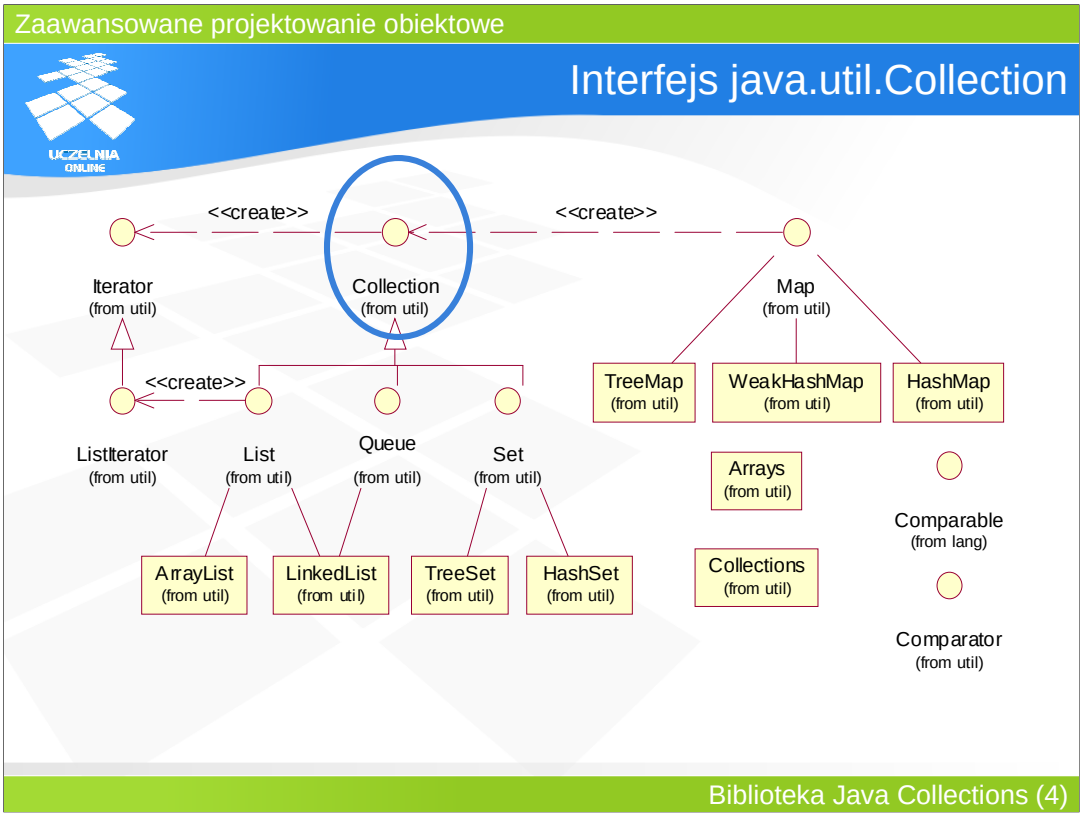
- **1995: JDK 1.0**
  - niezależne klasy kolekcji: tablice *Vector*, *Hashtable* o niespójnym sposobie dostępu do elementów
  - brak wydzielonych interfejsów
- **1998: JDK 1.2**
  - oparcie kolekcji na interfejsach *Collection*, *Set*, *List* i *Map* oraz różnorodnych implementacjach
  - iteratory, algorytmy, porównania
- **2005: JDK 1.5**
  - wprowadzenie typów generycznych

Kolekcje są obecne w JDK od początku istnienia Javy. Jednak, wydana w roku 1995 wersja 1.0 zawierała zestaw kilku słabo związanych ze sobą klas, które reprezentowały różne rodzaje kolekcji: prostą tablicę lub klasę Vector (jako listę), Hashtable (jako tablicę asocjacyjną). Klasy te były gotowymi implementacjami o sfinalizowanych metodach (co utrudniało ich rozszerzanie), brakowało abstrakcyjnych interfejsów, które pozwoliłyby lepiej modularyzować kod i hermetyzować jego funkcjonalność.

Dlatego w wersji 1.2 JDK pojawiła się zaprojektowana od nowa biblioteka Java Collections, która rozwiązywała większość istniejących problemów. Wprowadzono nowe interfejsy, które tworzyły szkielet biblioteki: Collection, Set, List i Map, dzięki czemu implementacje nie były ze sobą bezpośrednio związane. Dostęp do elementów kolekcji został ujednolicony poprzez wprowadzenie interfejsu Iterator, implementowanego we wszystkich kolekcjach. Ponadto, w bibliotece zaimplementowano typowe algorytmy wyszukiwania, sortowania, a także wymienny mechanizm porównywania obiektów.

Kolejna duża zmiana wiązała się z wprowadzeniem do języka Java typów generycznych (ang. *generic types*). Pozwalają one sprawdzać w momencie kompilacji, czy typy obiektów są zgodne z deklaracją. Ma to ogromny wpływ przede wszystkim na kolekcje, które dotąd przechowywały obiekty typu java.lang.Object, a więc "zapominały" o typach. Wraz z pojawieniem się typów generycznych możliwe stało się określenie typu elementów i weryfikacja w momencie kompilacji.

Wprowadzane zmiany miały ogromny wpływ na projekt, funkcjonalność i sposób wykorzystania biblioteki Java Collections. Warto zauważyć, że biblioteka ta zachowała jednak wsteczną zgodność z poprzednimi wersjami, co miało istotny wpływ na drogi jej ewolucji.



Interfejs Collection stanowi podstawę projektu biblioteki Java Collections. Bezpośrednio dziedziczą po nim interfejsy List, Queue oraz Set, a intensywnie korzysta z niego interfejs Map. Każda kolekcja produkuje swój własny obiekt o interfejsie Iterator.

Pozostałe klasy są implementacjami wymienionych interfejsów lub niezależnymi klasami pomocniczymi, blisko związanymi z kolekcjami.

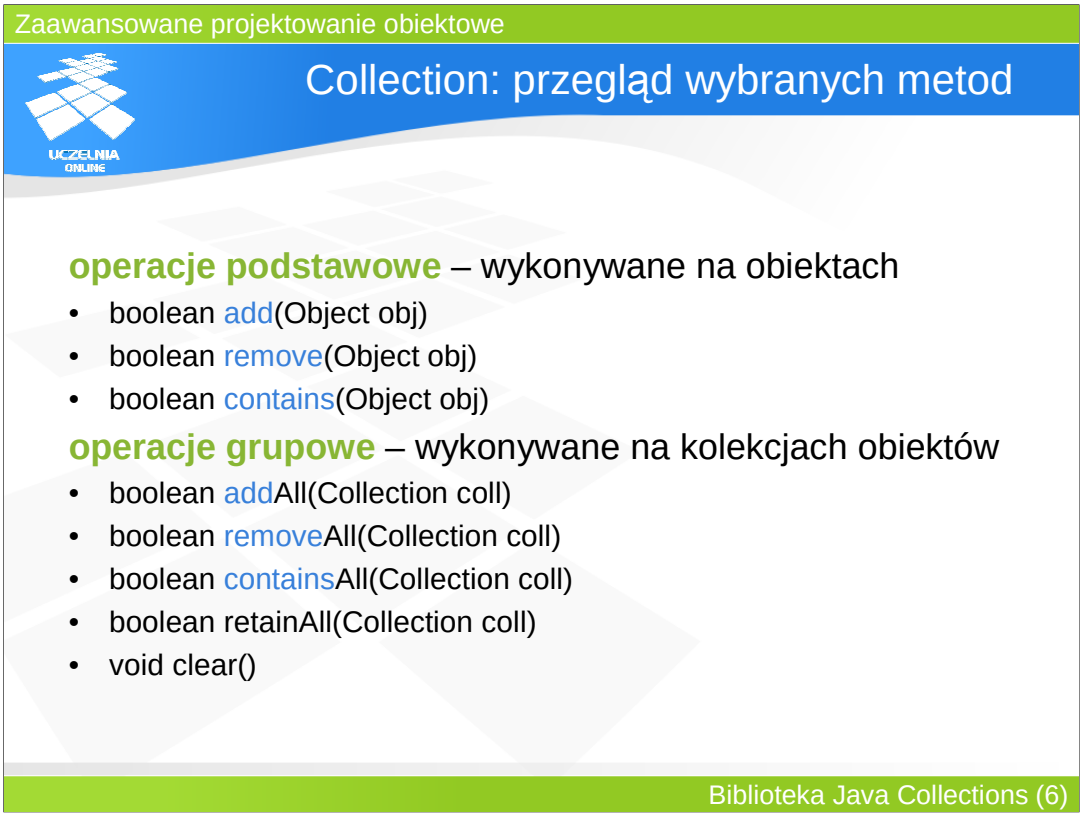
Collection: specyfikacja

- Wspólny interfejs dla większości kolekcji
- Reprezentuje obiekt zarządzający *grupą elementów*
- Brak ograniczeń dotyczących elementów
- Dwa obligatoryjne konstruktory:
  - bezparametrowy: *Collection()*
  - kopiujący: *Collection(Collection src)*
- Konstruktor zapewnia utworzenie obiektu zgodnego z semantyką danej kolekcji
- Metody nieobowiązkowe mogą zgłaszać wyjątek *java.lang.UnsupportedOperationException*
- Brak bezpośredniej implementacji w JDK

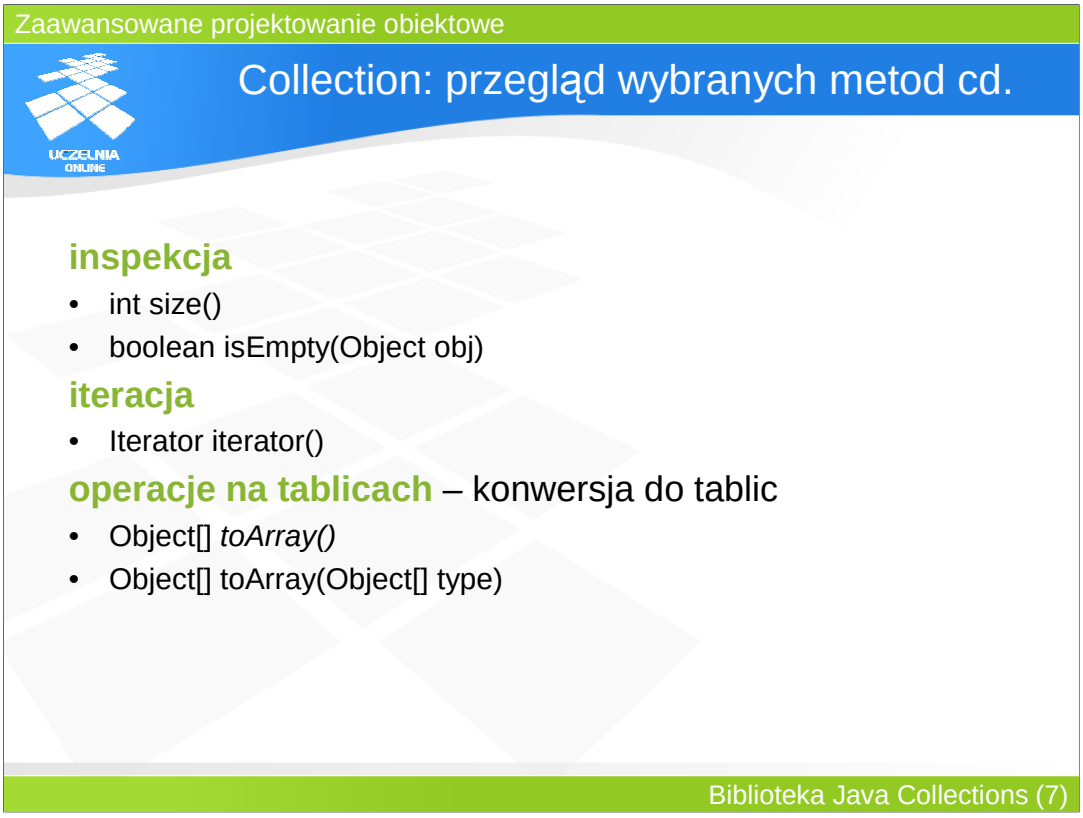
Z uwagi na rolę interfejsu Collection, konieczne było przyjęcie założeń dotyczących jego semantyki, konwencji nazewniczych i projektowych. Collection, jako najogólniejszy typ kolekcji, nakłada najmniejsze ograniczenia na przechowywane w nim elementy: specyfikacja mówi, że przechowuje on po prostu grupę elementów; z tego względu szczegółowe decyzje, dotyczące uporządkowania elementów, duplikatów, dostępu do elementów etc. są podejmowane w dziedziczących interfejsach lub bezpośrednio w implementacjach. Na szczególną uwagę zasługuje niesprawdzany wyjątek java.lang.UnsupportedOperationException, który jest zgłaszany, gdy implementacja interfejsu, mimo syntaktycznej konieczności zdefiniowania metody, nie chce jej obsługiwać. W ten sposób, unikając zubażania interfejsu Collection, obsługiwane są szczególne wypadki, w których konkretna kolekcja celowo jest pozbawiona pewnej funkcjonalności

Interfejs ten nie posiada bezpośredniej implementacji, tzn. nie istnieje w JDK klasa, która implementowałaby tylko ten interfejs. Jednak są struktury danych, które można stworzyć właśnie w ten sposób, np. multizbiór. Jednak dzięki temu wszystkie kolekcje są do pewnego stopnia ze sobą zgodne.

Przyjęto też konwencję, że każda kolekcja posiada dwa konstruktory: bezparametrowy, tworzący pustą kolekcję oraz kopiujący, który zapamiętuje w nowej kolekcji wszystkie elementy należące do starej. Konwencja ta jest stosowana dla wszystkich klas potomnych tego interfejsu: list, zbiorów i kolejek.

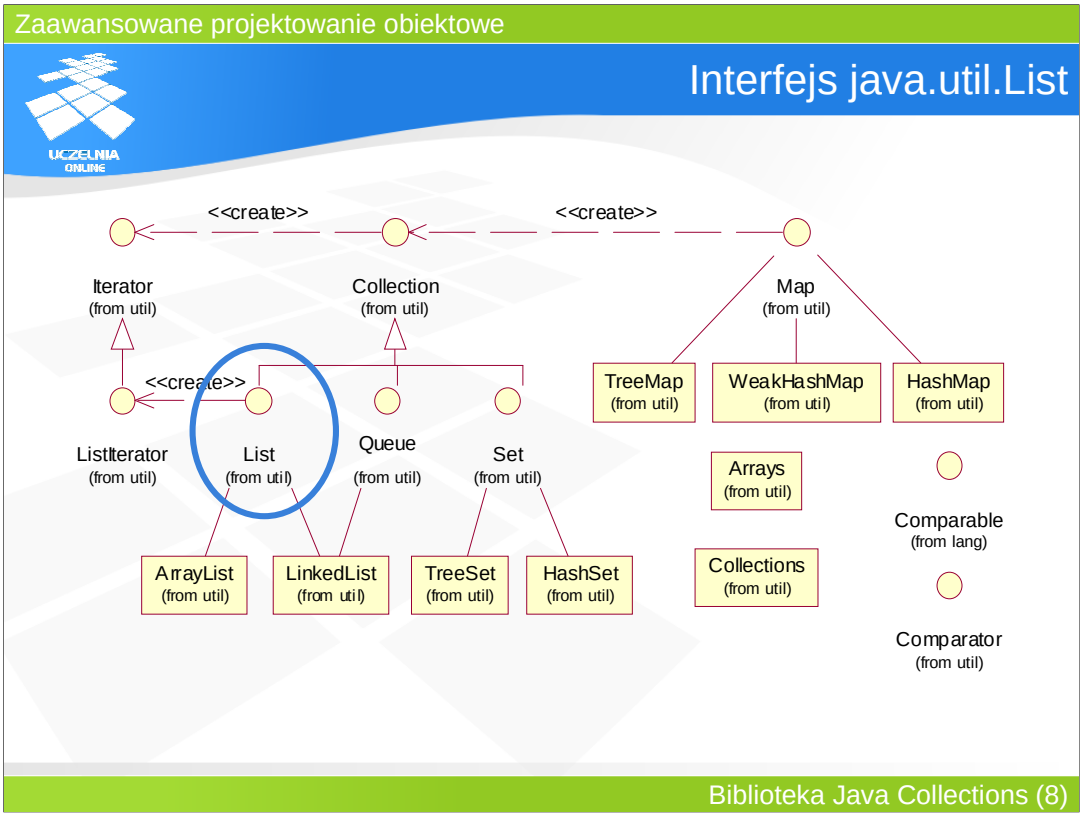


Interfejs Collection definiuje tylko te metody, które mogą być zaimplementowane we wszystkich kolekcjach, bez względu na ich semantykę. Posiada zatem zestaw metod służących do operacji na pojedynczych obiektach (*add()*, *remove()* i *contains()*) oraz odpowiadające im metody operujące na kolekcjach obiektów (*addAll()*, *removeAll()* i *containsAll()*). Wartość zwracana przez te metody – *true* lub *false* – oznacza, czy metoda się powiodła, czy nie. Ponieważ interfejs Collection jest ogólny, nie przesądza on o tym, jak poszczególne metody się zachowują. Np. w niektórych implementacjach niedopuszczających duplikatów elementów, metoda dodająca element zwraca wartość *false*, jeżeli następuje próba wstawienia duplikatu. W ten sposób w interfejsie przewidziano możliwość zastosowania w implementujących go klasach rozwiązań różniących się semantyką.



Warto zauważyć, że interfejs Collection nie definiuje sposobu dostępu do poszczególnych elementów kolekcji. Dzieje się tak, ponieważ kolekcja nie decyduje o sposobie organizacji elementów, a zatem nie można stwierdzić, czy np. elementy można indeksować, jak w przypadku tablicy. Z drugiej strony, interfejs ten deklaruje metodę *iterator()*, która zwraca obiekty typu Iterator. Pozwala on na sekwencyjny dostęp do wszystkich elementów kolekcji, dzięki czemu jest realizowany dostęp do nich. Istnienie takiej metody nakłada na wszystkie kolekcje obowiązek zaimplementowania jej, tzn. że wszystkie pozwalają na taki właśnie dostęp do elementów.

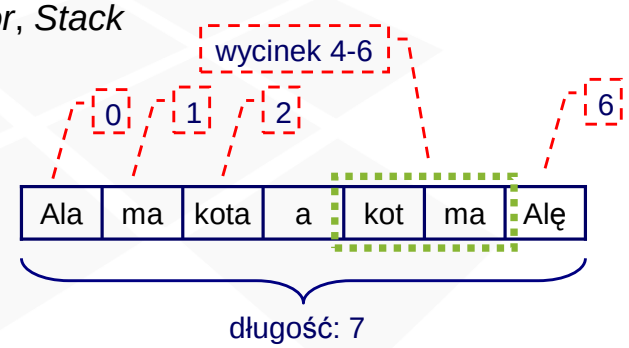
Collection posiada także dwie metody służące jak interfejs umożliwiający utworzenie tablicy na podstawie kolekcji. Wersja metody *toArray()* z parametrem typu Object[] pozwala utworzyć tablicę określonego typu – parametr służy wówczas jako prototyp zwracanej tablicy.



Najprostszym rodzajem kolekcji jest lista. W przeciwieństwie do interfejsu Collection, narzuca ona pewne ograniczenia i definiuje znacznie bardziej szczegółową semantykę.

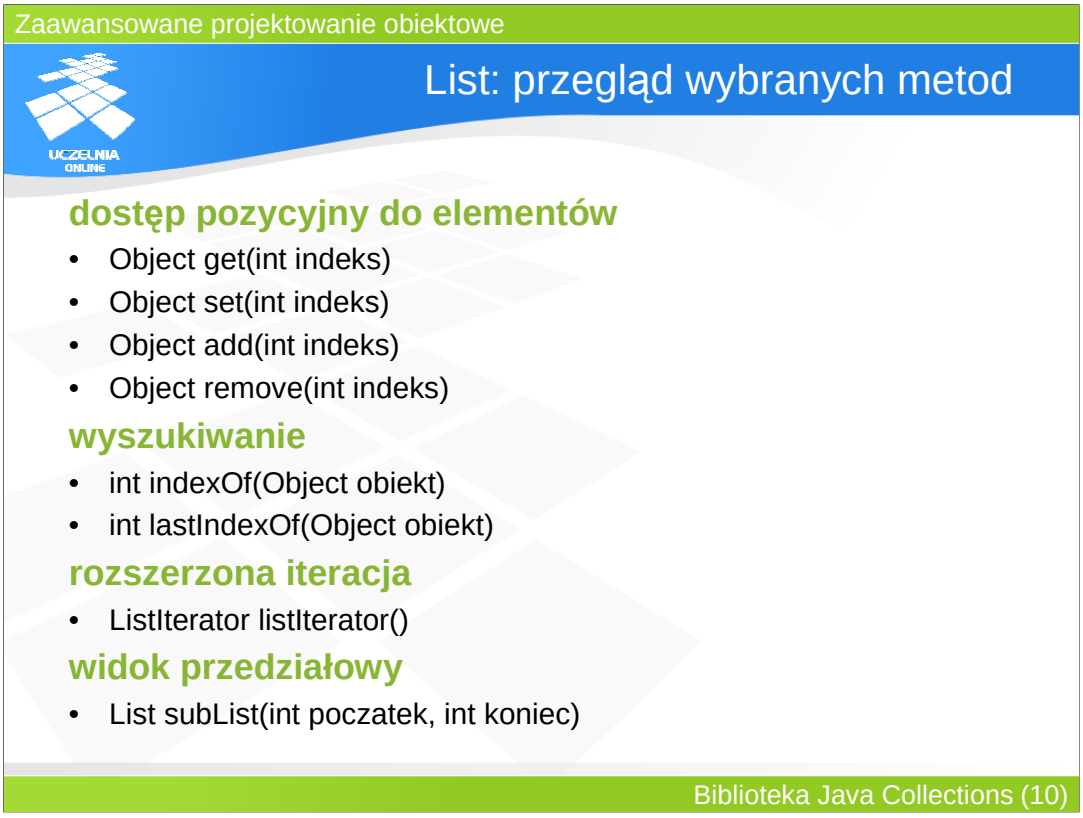
List: specyfikacja

- Reprezentacja *uporządkowanej sekwencji elementów*
- Większość implementacji dopuszcza istnienie duplikatów elementów
- Najważniejsze implementacje: *ArrayList*, *LinkedList*, *Vector*, *Stack*



Lista reprezentuje uporządkowaną kolekcję, tzn. taką, w której elementom można przypisać kolejne liczby. Interfejs ten nie decyduje o możliwości przechowywania duplikatów, ale wszystkie implementacje znajdujące się w JDK pozwalają na ich istnienie.

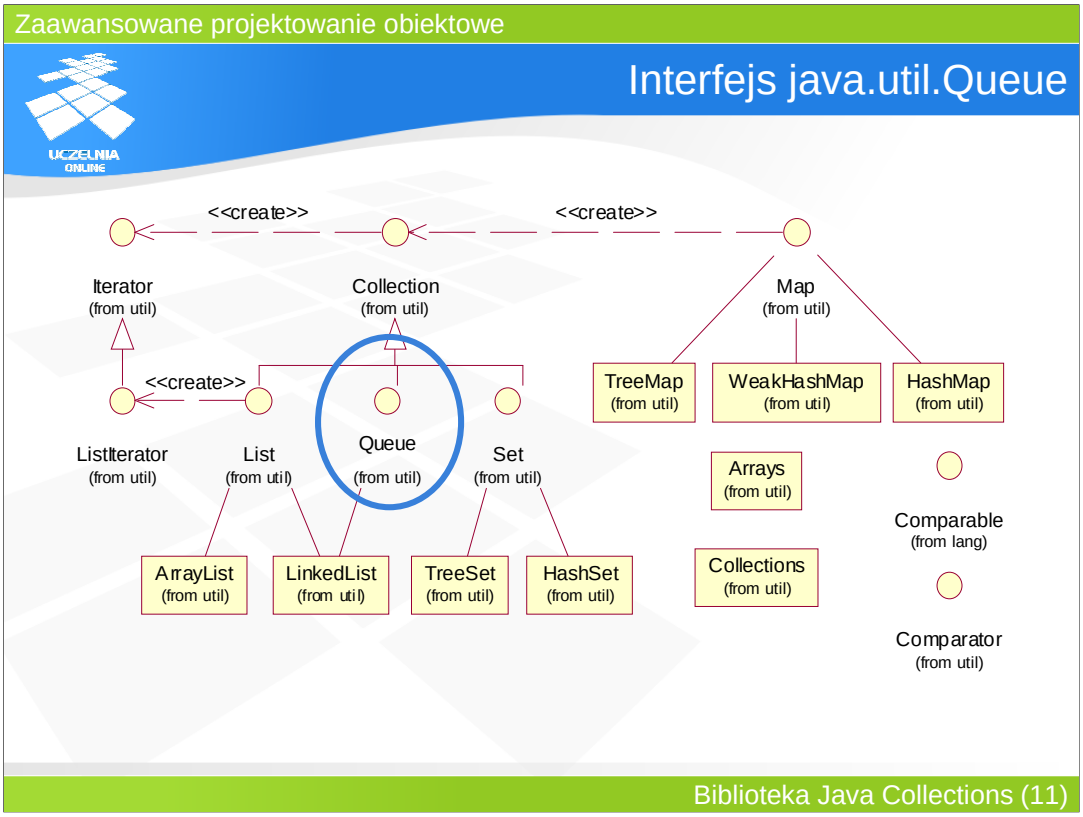
JDK posiada kilka implementacji tego interfejsu, które całkowicie różnią się sposobem przechowywania elementów. Klasa ArrayList stosuje w tym celu tablicę, która jest zwiększana w momencie przekroczenia jej maksymalnego rozmiaru. Klasa LinkedList jest klasyczną listą łączoną, w której każdy element posiada referencję do następnika. Te dwie implementacje charakteryzują się różną wydajnością, ale można stosować je zamiennie. Z kolei klasa Vector jest przepisaną na nowo wersją znaną z JDK 1.0. Jest obecna w bibliotece Java Collections tylko ze względu na potrzebę wstecznej zgodności.



Z semantyki interfejsu wynika funkcjonalność, jakiej powinien on dostarczać. Lista posiada więc metodę *get()*, służącą do pozycyjnego dostępu do elementów, metody *set()* i *add()*, które dodają element do kolekcji i *remove()*, który go usuwa. Za każdym razem element jest identyfikowany poprzez pozycję na liście, a metody te zwracają referencję do obiektu, do którego następuje odwołanie (w przypadku metod *set()* i *add()* jest to element, który znajdował się na liście poprzednio; ponadto *set()* zmienia element na podanej pozycji na inny, a *add()* dodaje go, przesuwając następne elementy).

Ponadto istnieją metody wyszukujące elementy, które odwracają sposób działania metody *get()* – znajdują pozycję elementu na podstawie referencji do niego.

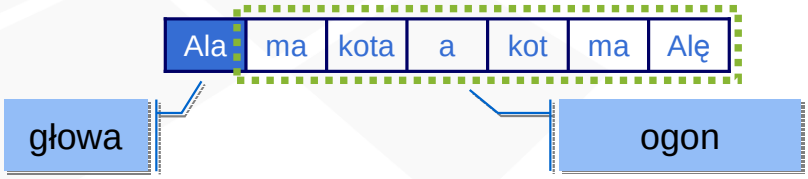
Lista, korzystając ze swoich możliwości, definiuje rozszerzoną wersję iteratora. Warto zauważyć, że lista w ten sposób posiada dwa iteratory: odziedziczony z Collection oraz własny, typu ListIterator.



Interfejs Queue jest stosunkowo nowym elementem biblioteki.
Wprowadzono go, aby doprecyzować różnice, jakie dzielą listy od kolejek.

Queue: specyfikacja

- Bezpośredni potomek interfejsu *Collection*
- Reprezentacja kolejki (LIFO, FIFO, priorytetowa)
- Podział na pierwszy element (*głowę*) i pozostałą część kolejki (*ogon*)
- Najważniejsze implementacje: *LinkedList*, *PriorityQueue*



Interfejs ten reprezentuje kolejkę – klasyczną strukturę danych, w której wyróżnione są dwa elementy: głowa (ang. *head*) – pierwszy element, do którego można się w każdej chwili odwołać, i ogon (ang. *tail*) – kolejkę będącą pozostałością po odcięciu głowy. Zmiana polega tutaj na innym sposobie dostępu: odwołania dotyczą głowy, którą można odczytać i usunąć, natomiast dostęp do ogona jest możliwy dopiero po usunięciu dotychczasowej głowy (głową staje się pierwszy element ogona). Posługiwanie się w kolejkach metodami zdefiniowanymi dla Collection wydaje się nieuzasadnione, jeżeli metody mają swoje odpowiedniki w interfejsie Queue. Widać w ten sposób, że jest to nowy interfejs w JDK, który nie do końca odpowiada pierwotnym założeniom dotyczącym rozkładu odpowiedzialności poszczególnych interfejsów i klas.

W większości przypadków kolejki są skonstruowane w ten sam sposób do listy. Wskazują na to istniejące w JDK implementacje, np. LinkedList, które jednocześnie implementują interfejsy List i Queue. Dzięki temu ten sam obiekt może pełnić dwie różne role, w zależności od kontekstu.

Queue: przegląd metod

Metody niezależne od interfejsu Collection:

dodawanie elementu

- *boolean offer(Object obiekt)*

usuwanie elementu

- *Object remove()*
- *Object poll()*

inspekcja

- *Object element()*
- *Object peek()*

Metody dodane w interfejsie Queue (w porównaniu do interfejsu Collection) w zasadzie dublują ich funkcjonalność, jednak pozwalają posługiwać się tym obiektem jak kolejką.

Metoda *offer()* wstawia obiekt do kolejki w miejscu zależnym od implementacji (FIFO, LIFO czy priorytetowa). Wartość zwracana przez tę metodę wskazuje, czy operacja się powiodła.

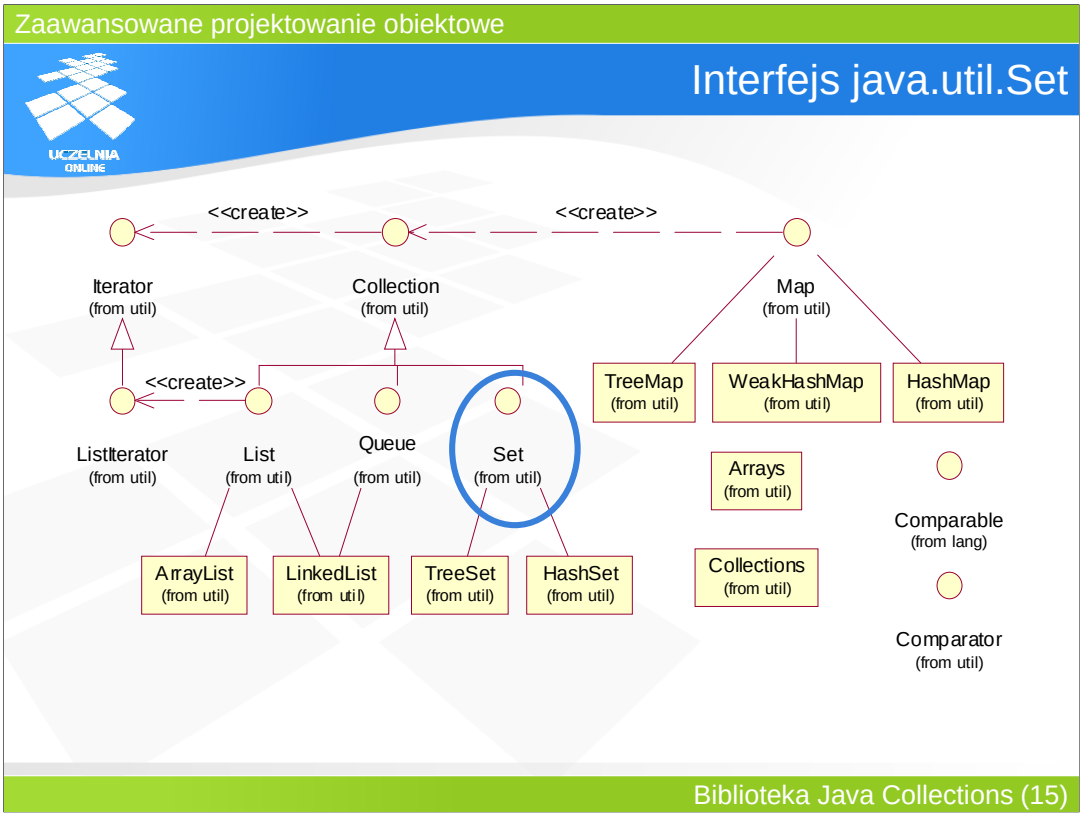
Metody *remove()* i *poll()* usuwają głowę kolejki i zwracają ją. Gdy kolejka jest pusta, metoda *remove()* zgłasza wyjątek, a *poll()* zwraca wartość *null*.

Podobnie zachowują się metody *element()* i *peek()*, które udostępniają głowę kolejki, ale bez jej usuwania: w przypadku pustej kolejki pierwsza zgłasza wyjątek, druga – zwraca wartość *null*.

Przykład

```
public class Odliczanie {
   public static void main(String[] args) {
      int liczba = Integer.parseInt(args[0]);
      Queue kolejka = new LinkedList();
      for (int i = liczba; i >= 0; i--) {
         kolejka.add(i);
      }
      while (!kolejka.isEmpty()) {
         System.out.println(kolejka.remove());
      }
   }
}
```

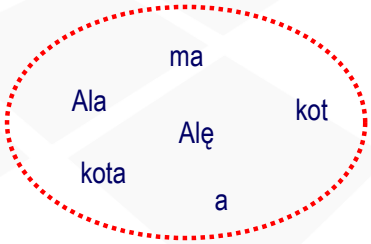

Oto prosty przykład wykorzystania kolejek. Elementy przekazane z linii poleceń są wstawiane do kolejki, a następnie kolejka jest przetwarzana w pętli *while*: dopóki kolejka nie jest pusta, jest usuwany z niej i wyświetlany element stanowiący jej głowę.



Kolejnym interfejsem dziedziczącym po Collection jest zbiór (*java.util.Set*).

Set: specyfikacja

- Reprezentacja zbioru matematycznego
  - brak relacji porządku wewnątrz zbioru
  - brak duplikatów
- Brak nowych metod w stosunku do *Collection*
- najważniejsze implementacje: *HashSet*, *TreeSet*



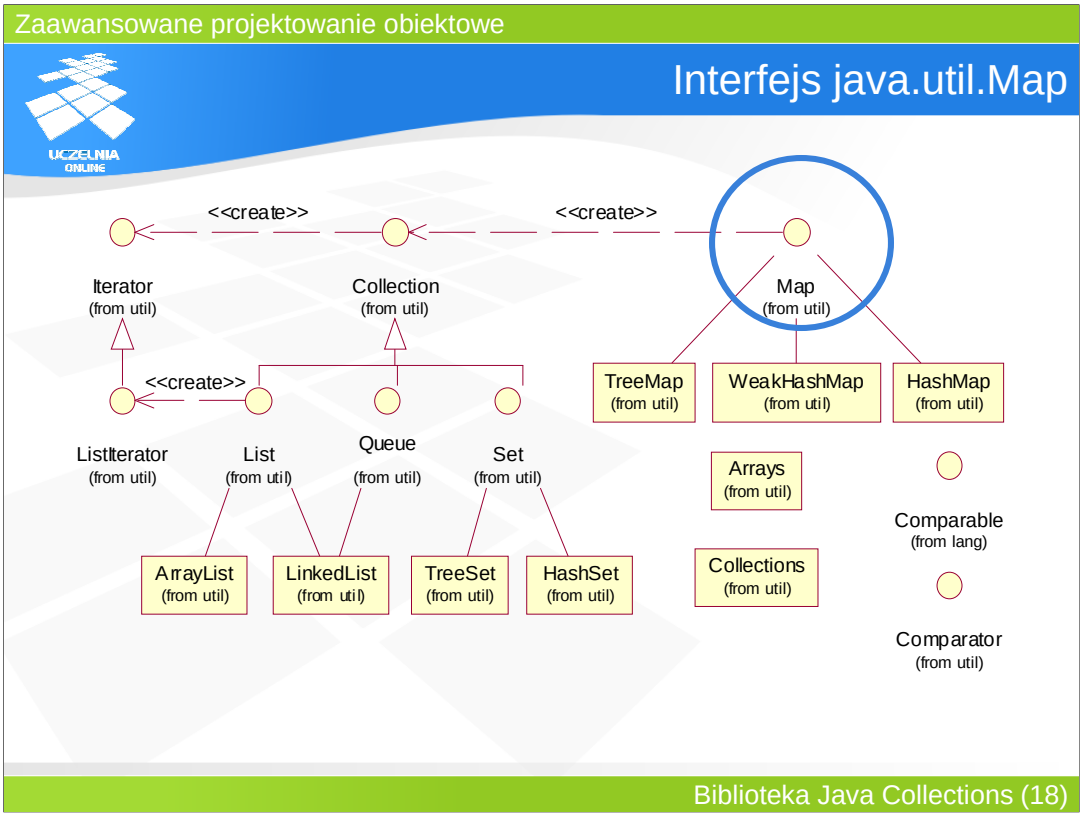
Zbiór posiada, odmienną niż lista, semantykę: nie zachowuje kolejności elementów, natomiast wyklucza istnienie duplikatów. O elemencie można zatem powiedzieć jedynie, czy należy do zbioru (w jednym egzemplarzu), czy nie. Co ciekawe, interfejs ten nie definiuje żadnych nowych metod w porównaniu do interfejsu Collection. Wynika to z faktu, że trudno wskazać funkcjonalność, która wyróżniałaby zbiór od kolekcji. Warto zadać pytanie, dlaczego zatem do jego reprezentacji powołano specjalny interfejs? Otóż wskazanie, że określony obiekt jest typu Set niesie informację bardziej szczegółową niż w przypadku zwykłej kolekcji, dlatego zostało to podkreślone przez specjalny typ obiektu.

Podobnie, jak w przypadku listy, zbiór posiada w JDK kilka gotowych implementacji. Jedną z nich jest HashSet, w którym unikatowość elementów jest zapewniona przez zastosowanie tablicy asocjacyjnej; w przypadku klasy TreeSet rolę tablicy pełni drzewo dwukolorowe.

Przykład

```
public class Duplikaty {
   public static void main(String args[]) {
      Set s = new HashSet();
      for (int i = 0; i < args.length; i++) {
         if (! s.add(args[i]))
             System.out.println("Duplikat!");
      }
   }
}
```

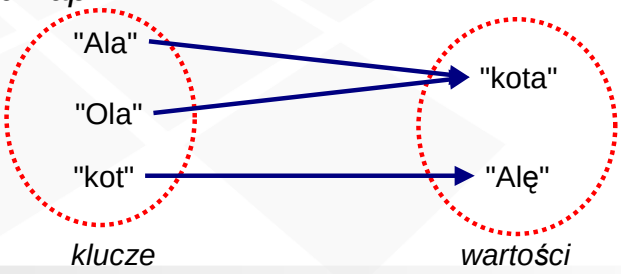

Na tym slajdzie przedstawiono przykładowe wykorzystanie zbiorów do eliminacji duplikatów. Dodawanie do zbioru elementu kończy się sukcesem tylko w przypadku, gdy element ten dotychczas w zbiorze nie istniał. W przeciwnym przypadku metoda *add()* zbioru zwraca wartość *false*.



Kolejnym ważnym elementem biblioteki Java Collections jest interfejs Map. Nie jest on związany z interfejsem Collection przez dziedziczenie, ponieważ zarówno jego kontrakt, jak i struktura są różne, natomiast jest z nim powiązany innymi relacjami.

Map: specyfikacja

- Kolekcja jednoznacznych odwzorowań *klucz-wartość*
- Klucze, wartości i pary *klucz-wartość* dostępne jako obiekty *Collection*
- Dwa obowiązkowe konstruktory (analogicznie do kolekcji)
- Najważniejsze implementacje: *HashMap*, *TreeMap*, *SortedMap*

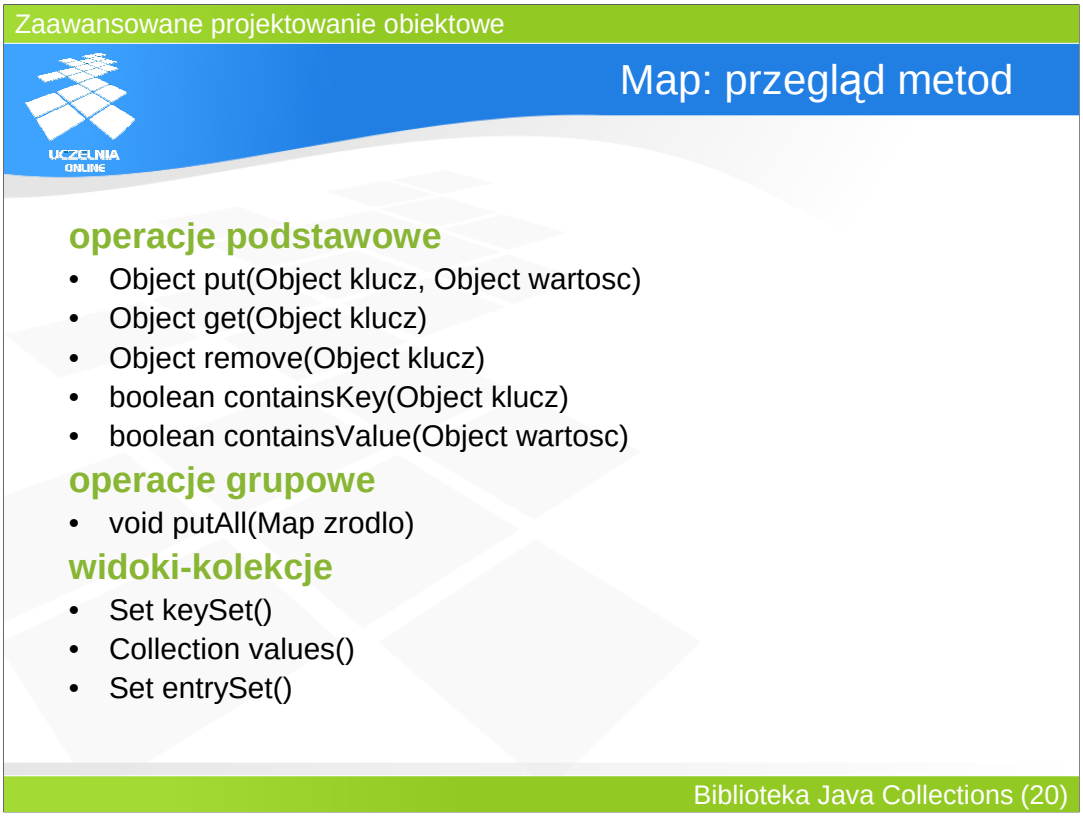


Mapa przechowuje nie pojedyncze elementy, ale pary klucz-wartość. Pary te pozwalają na jednoznaczne odwzorowanie klucza na wartość, tzn. dla każdego klucza istnieje co najwyżej jedna odpowiadająca mu wartość. W ten sposób mapa nie jest bezpośrednio związana z interfejsem Collection, jednak pozwala na dostęp do swoich składowych traktowanych właśnie jako instancje tego interfejsu. Istnieją trzy widoki, udostępniające zbiór kluczy (*keySet()*), kolekcję wartości (*values()*) i zbiór par klucz-wartość (*entrySet()*). Zatem każdą mapę można – w razie potrzeby – traktować jak zwykłą kolekcję.

Mapy z założenia nie gwarantują określonej kolejności elementów w żadnym z tych widoków, choć mogą zapewnić to implementacje.

Analogicznie, jak w przypadku kolekcji, mapy posiadają dwa konwencjonalne konstruktory: bezparametrowy i kopiujący; w przypadku tego drugiego parametrem jest inna mapa.

JDK zawiera trzy implementacje map: HashMap, TreeMap i SortedMap. Ta ostatnia mapa dodatkowo właśnie zapewnia porządek wśród przechowywanych w mapie par klucz-wartość.

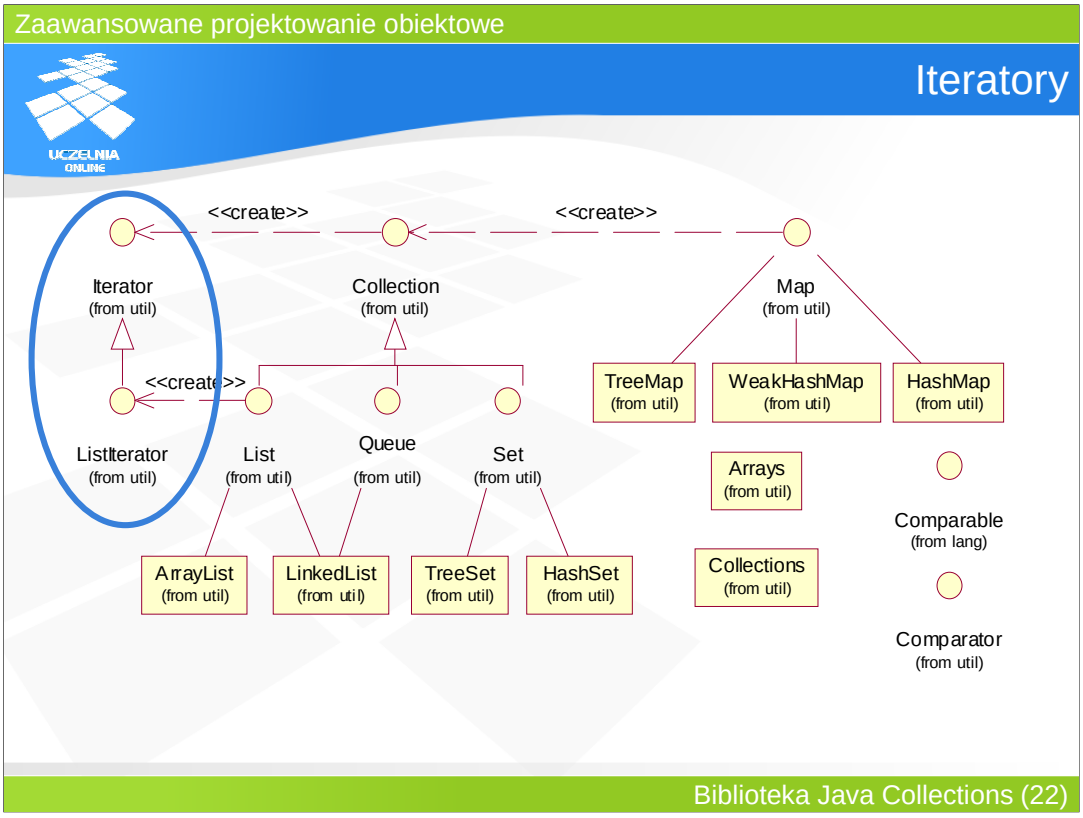


Z uwagi na odmienną funkcjonalność oferowaną przez mapę, wszystkie metody są zdefiniowane od nowa. Wśród operacji podstawowych: metoda *put()* służy do wstawienia do mapy nowej pary klucz-wartość, *get()* – odczytanie wartości związanej z danym kluczem, a *remove()* – usunięcie pary z podanym kluczem. Istnieją osobne metody weryfikujące obecność danego klucza i wartości.

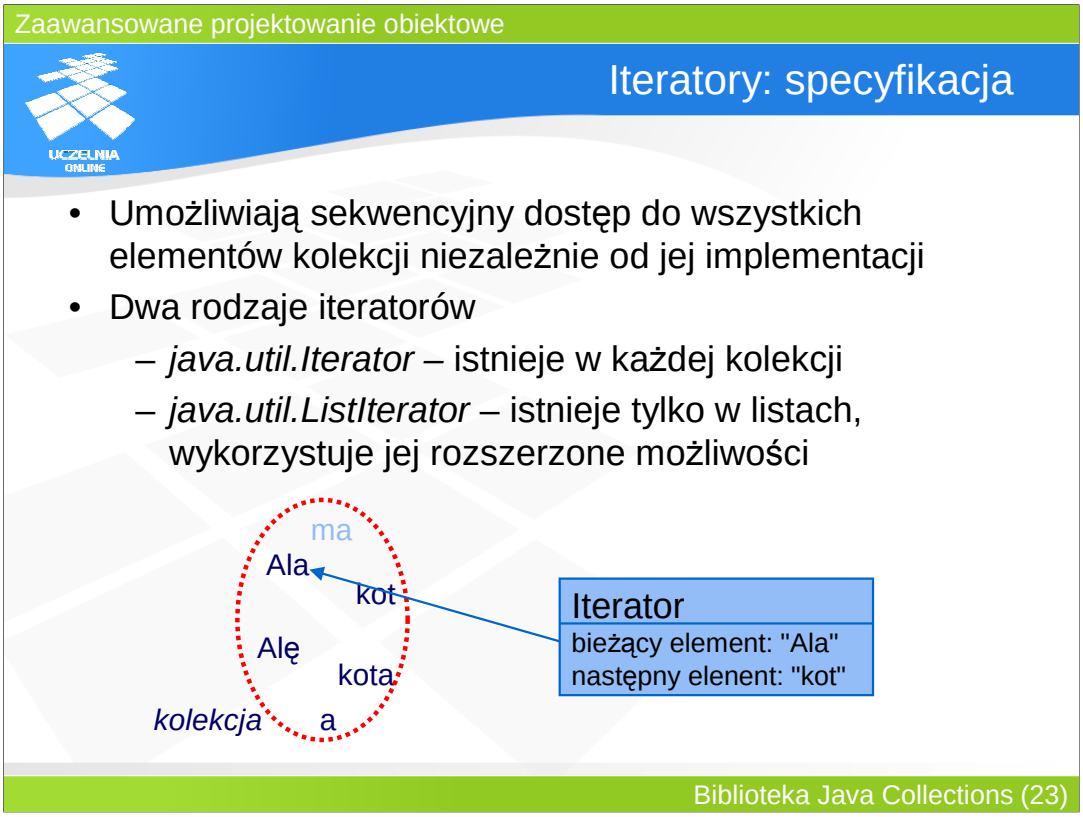
Przykład

```
public class Czestotliwosc {
   private static final Integer JEDEN = new Integer(1);
   public static void main(String args[]) {
      Map mapa = new HashMap();
      for (int i=0; i<args.length; i++) {
         Integer czest = (Integer) mapa.get(args[i]);
         mapa.put(args[i], (czest == null ? JEDEN :
            new Integer(czest.intValue() + 1)));
      }
      System.out.println(mapa.size() + " różnych słów");
      System.out.println(mapa);
   }
}
```


W tym przykładzie mapa służy do przechowywania informacji o częstotliwości wystąpień obiektów przekazanych jako parametry wywołania programu. W mapie kluczami są wyrazy, natomiast wartościami liczby wskazujące na częstotliwość. Każdorazowy odczyt wartości związanej z wyrazem-kluczem może dać dwa wyniki: jeżeli wartość taka nie istnieje, to znaczy, że wcześniej takie odwołanie nie miało miejsca i obecna częstotliwość wynosi jeden. W przeciwnym przypadku, wartość pochodząca z mapy jest traktowana jako liczba, która jest zwiększana o jeden. W obu przypadkach wartość ta jest ponownie wprowadzana do mapy.



Kilkukrotnie, podczas omawiania kolekcji i map, była mowa o iteratorach. Są one blisko związane z kolekcjami: każda kolekcja tworzy swoją własną implementację Iteratora.



Celem iteratora jest udostępnienie wszystkich elementów kolekcji w sposób niezależny od samej kolekcji i jej implementacji. Dzięki temu w bibliotece Java Collections wszystkie kolekcje można przejrzeć w identyczny sposób. Iterator jest obiektem stanowym: zapamiętuje bieżący element w kolekcji, a wywołanie określonej metody (w Javie jest to *next()*) powoduje przesunięcie tego wskaźnika na kolejny element. Koniec kolekcji jest określany za pomocą metody *hasNext()*, która zwraca true tylko wówczas, gdy istnieje jeszcze nieodwiedzony element.

Biblioteka zawiera dwa rodzaje iteratorów: zwykłe, implementujące interfejs *java.util.Iterator*, które pozwalają jedynie przeglądać wszystkie elementy oraz usuwać bieżący element, oraz listowe, implementujące interfejs *java.util.ListIterator* dziedziczący po interfejsie Iterator. Ten ostatni posiada znacznie rozszerzone możliwości, wynikające z semantyki listy: stałą kolejność dostępu do elementów, możliwość zmiany kierunku, a także dodawania elementów. Warto zwrócić uwagę, że o ile każdy iterator ma możliwość usuwania bieżącego elementu kolekcji, o tyle tylko iteratory listowe mogą dodawać elementy. Jest to świadoma decyzja autorów: nie można przewidzieć konsekwencji dodawania elementów do każdej kolekcji poprzez iterator.

W przypadku listy najprawdopodobniej obie metody (*iterator()* i *listIterator()*) zwracają iteratory tego samego typu – dzięki korzystaniu z nich przez interfejsy ich funkcjonalność jest ograniczona tylko do specyfikacji interfejsu. Jest to ciekawy przykład hermetyzacji implementacji klasy przy użyciu interfejsu.

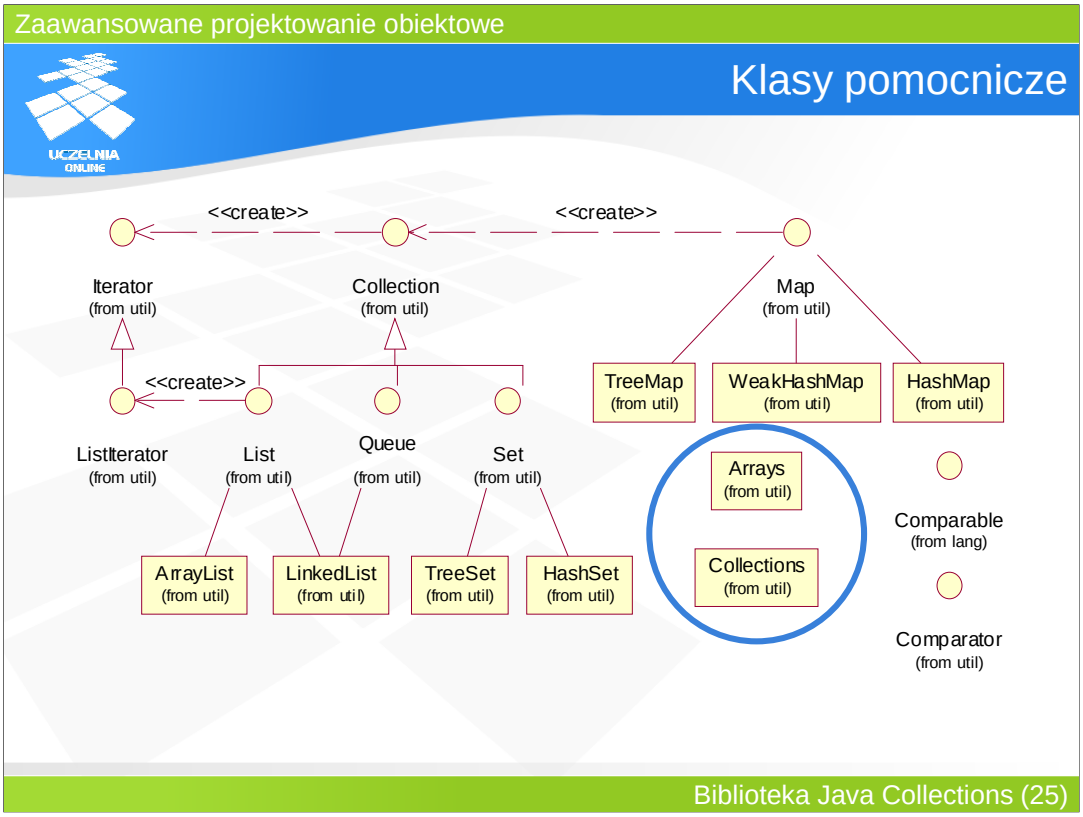
Szczegółowy opis funkcjonalności Iteratora zostanie przedstawiony podczas wykładu o wzorcach projektowych.

Przykład

```
public class Iteracja {
   public static void main(String args[]) {
      Collection coll = new ArrayList();
      for (int i = 0; i < args.length; i++) {
         coll.add(args[i]);
      }
      for (Iterator iter = coll.iterator(); iter.hasNext();)
         String element = (String) iter.next();
         System.out.println("element = " + element);
      }
   }
}
```

```
> java Iteracja Ala ma kota
element = Ala
element = ma
element = kota
```


Przykład pokazuje sposób wykorzystania iteratora w pętli *for*. Jest to pewnego rodzaju idiom, który warto zapamiętać.



Nie cała funkcjonalność związana z kolekcjami jest zawarta w klasach je implementujących. Część znajduje się w klasach pomocniczych (w szczególności w klasie *java.util.Collections*), które oferują metody statyczne realizujące typowe funkcje. W ten sposób uniknięto duplikacji kodu w implementacjach poszczególnych interfejsów kolekcji

Algorytmy

Metody statyczne klasy *Collections* implementują typowe algorytmy wykonywane na kolekcjach:

- wyszukiwanie binarne
- sortowanie
- operacje algebraiczne na zbiorach
- odwracanie list
- permutacje list
- wyszukiwanie elementów max/min

Za ich pomocą można m.in.

•wyszukać metodą połowienia element wewnątrz listy

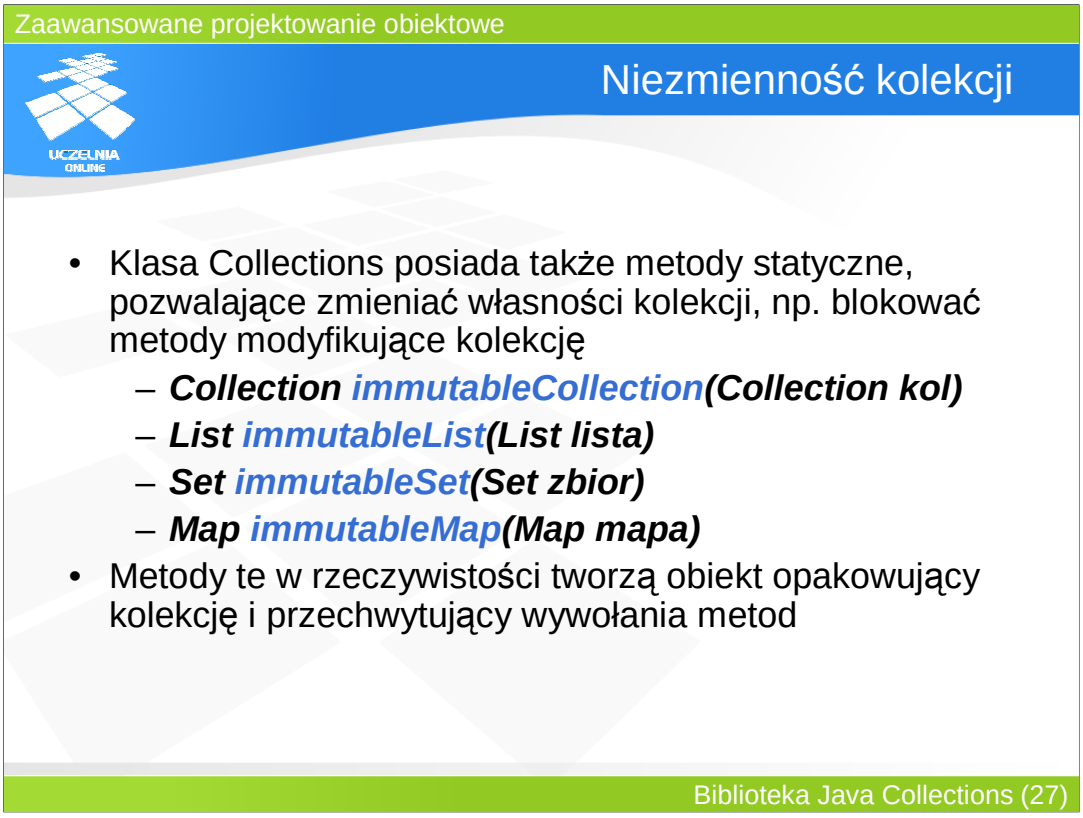
•posortować listę

•obliczyć sumę, część wspólną, rozłączną i różnicę dwóch zbiorów

•obliczyć dowolną permutację listy

•wyszukać największy i najmniejszy element listy

Funkcjonalność ta jest zawarta poza kolekcjami, ponieważ w bibliotece nie istnieje jedna klasa, po której inne dziedziczą. Wybrano implementację interfejsów, a nie dziedziczenie klas jako metodę wiązania kolekcji ze sobą, dlatego, aby uniknąć konieczności dodawania tej samej funkcjonalności w każdej z nich, wyłączono ją do statycznych metod w osobnej klasie.

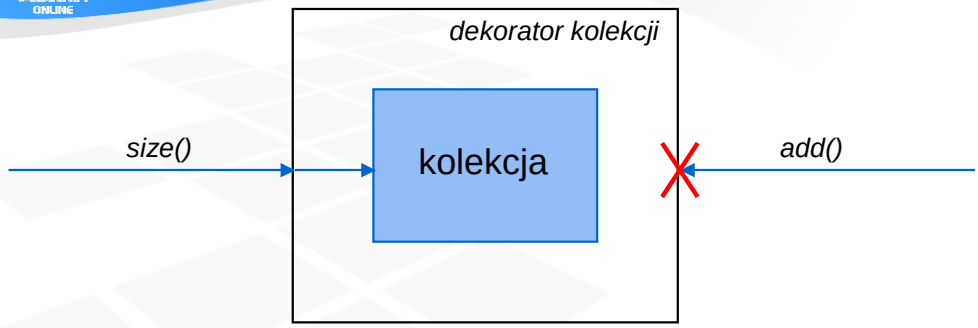


Jednym z ciekawszych rozwiązań zastosowanych w klasie Collections jest możliwość uczynienia kolekcji niemodyfikowalną. Typowe rozwiązanie polega na stworzeniu dedykowanej podklasy, która posiada zmodyfikowane właściwości. Jednak w praktyce jest to rozwiązanie nieakceptowalne: powoduje podwojenie liczby klas, w zasadzie nie wprowadzając istotnych różnic. Ponadto, często zachodzi konieczność uniemożliwienia modyfikacji w czasie, gdy kolekcja już istnieje. Zmiana klasy istniejącego obiektu jest jednak niemożliwa.

Dlatego w bibliotece Java Collections zastosowano wzorzec projektowy Decorator: opakowanie istniejącego obiektu specjalnym obiektem tego samego typu, który zmieni zachowanie wybranych metod. W tym przypadku będzie to zablokowanie wszystkich metod, które mogą modyfikować kolekcję. Pozostałe metody są bezpośrednio delegowane do opakowanej kolekcji. Z punktu widzenia klienta zmiana nie jest widoczna: zarówno oryginalna kolekcja, jak i opakowanie są tego samego typu, inna jest tylko referencja do tego obiektu. Opakowanie, naturalnie, nie przechowuje kopii elementów kolekcji, a jedynie odwołuje się do jej metod. Zastosowanie obiektów opakowujących ma też tę zaletę, że można jednocześnie użyć kilku kolejnych opakowań jednocześnie.

Warto zwrócić uwagę na sposób realizacji opakowania: metody opakowujące w klasie Collections przyjmują, jako argument, kolekcję (listę, zbiór, mapę), a zwracają taką samą kolekcję, ale w postaci opakowanej. Dzięki temu wywołanie metody pozornie zmienia klasę, do której należy przekazany obiekt.

Przykład



Kolekcja *Collection* jest "opakowana" w obiekt blokujący niektóre metody. Wywołania metody *size()* są przez obiekt opakowujący delegowane do wewnętrznej kolekcji, natomiast metody *add()* – blokowane.

Schemat działania tego rozwiązania jest następujący: kolekcja jest opakowana w specjalny obiekt, który blokuje metody modyfikujące kolekcję (np. metodę *add()*), natomiast inne metody są delegowane do kolekcji.

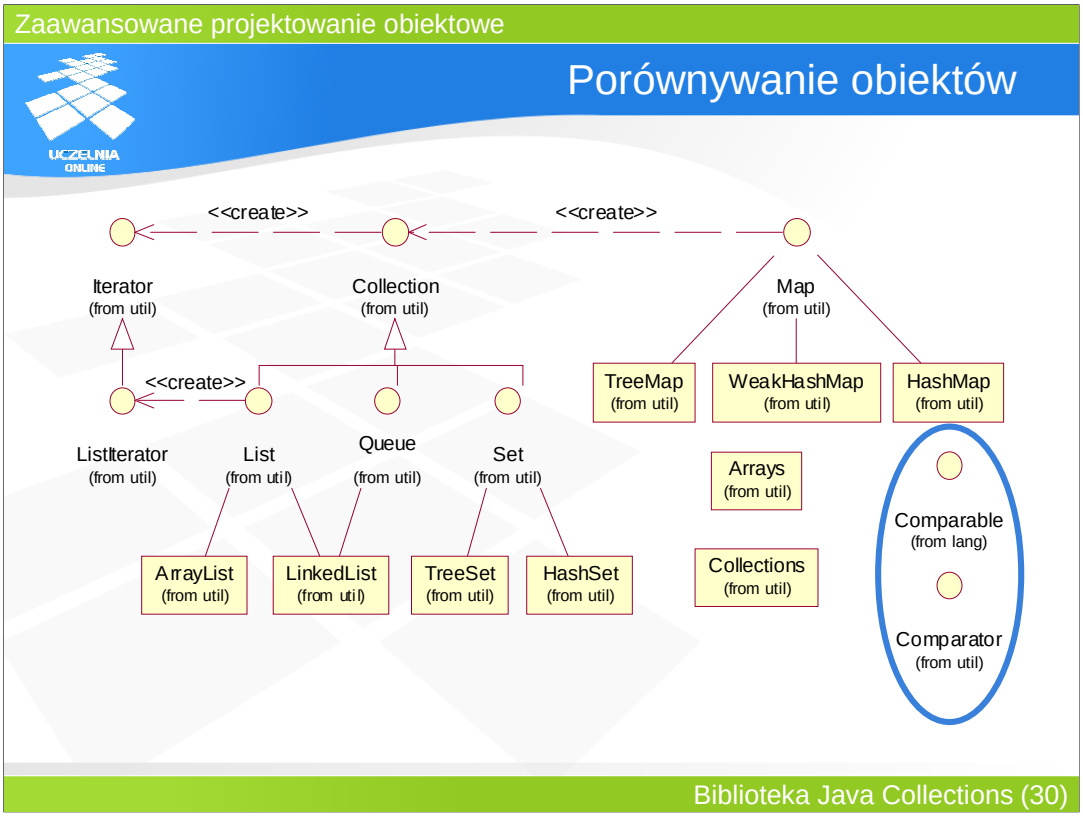
Inne klasy opakowujące

- Analogicznie, w klasie *Collections* istnieją inne metody tworzące specjalizowane obiekty opakowujące; nazwy metod są zgodne z konwencją i zależą od interfejsu
  - ***Interfejs* synchronized*Interfejs*(*Interfejs* i)** zapewnia synchronizację dostępu do interfejsu; wielowątkowy dostęp powoduje zgłoszenie wyjątku
  - ***Interfejs* checked*Interfejs*(*Interfejs* i)** zapewnia weryfikację typu wstawianego elementu do kolekcji

Podobne rozwiązanie jest stosowane w celu implementacji także innych właściwości:

•synchronizacji kolekcji (metody *synchronizedCollection()*, *synchronizedList()*, *synchronizedSet()* i *synchronizedMap()*), aby wewnątrz niej mógł operować tylko jeden wątek

•weryfikacji typu elementów w momencie wstawiania ich do kolekcji (metody *checkedCollection()*, *checkedList()*, *checkedSet()* i *checkedMap()*), które uniemożliwia wstawienie elementu o typie innym od uprzednio zadeklarowanego



Interfejsy te nie są bezpośrednio częścią biblioteki Collections, jednak są wykorzystywane przez wiele jej elementów, dlatego są przedstawiane właśnie w tym miejscu.

Porównywanie obiektów

- Wiele algorytmów wykonywanych na kolekcjach (np. sortowanie, wyszukiwanie) wymaga zdefiniowania relacji częściowego porządku
- Relacja porównania jest intuicyjnie zdefiniowana dla liczb, ale nie dla obiektów
  - Czy można porównać obiekty różnych klas?
  - Jak porównać dwie osoby?
- Porównywalność jest własnością obiektu, którą można z niego wyłączyć
- Java Collections korzysta z dwóch interfejsów służących do porównywania obiektów
  - *java.lang.Comparable*
  - *java.util.Comparator*

Porównywanie obiektów jest ściśle związane z przetwarzaniem kolekcji. Jest ono szczególnie istotne m.in. w przypadku sortowania, wyszukiwania min/max elementu etc. Dla liczb oraz napisów relacja porządku jest intuicyjna, choć nie jest ona jedyną możliwą. Natomiast porównywanie dowolnych obiektów nie jest już proste: porównania dwóch obiektów klasy Osoba można dokonać na wiele sposobów: alfabetycznie wg nazwiska, wg daty urodzenia, wg płci, wg adresu zamieszkania, lub dowolnej kombinacji wymienionych kryteriów.

Dlatego wprowadzono porównywalność jako własność obiektu, którą można zdefiniować. W tym celu wprowadzono dwa interfejsy: *java.lang.Comparable* i *java.util.Comparator*, których metody służą do porównywania obiektów. Implementując te interfejsy, programista podaje kryterium porównania.

Interfejs Comparable

- Jak posortować listę osób wg daty urodzenia?
  - Collections.sort(people)

```
public class Person implements Comparable {
   private Date birthday;

   public Date getBirthday() {
      return birthday;
   }
   public int compareTo (Object obj) {
      String birthday = ((Person) obj).getBirthday();
      return (this.birthday.compareTo(birthday));
   }
}
```


Spośród nich, interfejs Comparable jest w pewnym sensie bardziej intuicyjny: klasa, której obiekty mają być porównywalne, musi implementować ten interfejs, tzn. zdefiniować metodę *compareTo()*. Jej argumentem jest obiekt, z którym bieżący obiekt zostanie porównany. W przypadku porównania dat urodzenia z bieżącego obiektu i parametru są one odczytywane i zwracany jest wynik ich porównania. Należy zwrócić uwagę, że metoda *compareTo()* jest przeciwsymetryczna, tzn. zamiana obiektu z podmiotem może wpłynąć na jej wynik.

Interfejs Comparable cd.

- Jak posortować listę osób wg daty urodzenia?
  - Collections.sort(people)
- Sortowanie jest możliwe tylko dla list zawierających obiekty implementujące interfejs *java.lang.Comparable*
- Comparable definiuje jedną metodę, *compareTo(Object o)*, która zwraca
  - *mniej niż 0* – gdy jej parametr jest większy niż własny obiekt
  - *0* – gdy jej parametr jest równy własnemu obiektowi
  - *więcej niż 0* – gdy jej parametr jest mniejszy niż własny obiekt

Korzystanie z możliwości porównania obiektów klasy implementującej interfejs *java.util.Comparable* nie wymaga wielu zabiegów. W szczególności metoda sortująca zakłada, że na liście znajdują się obiekty implementujące ten interfejs, dlatego nie jest wymagana zmiana wywołania metody.

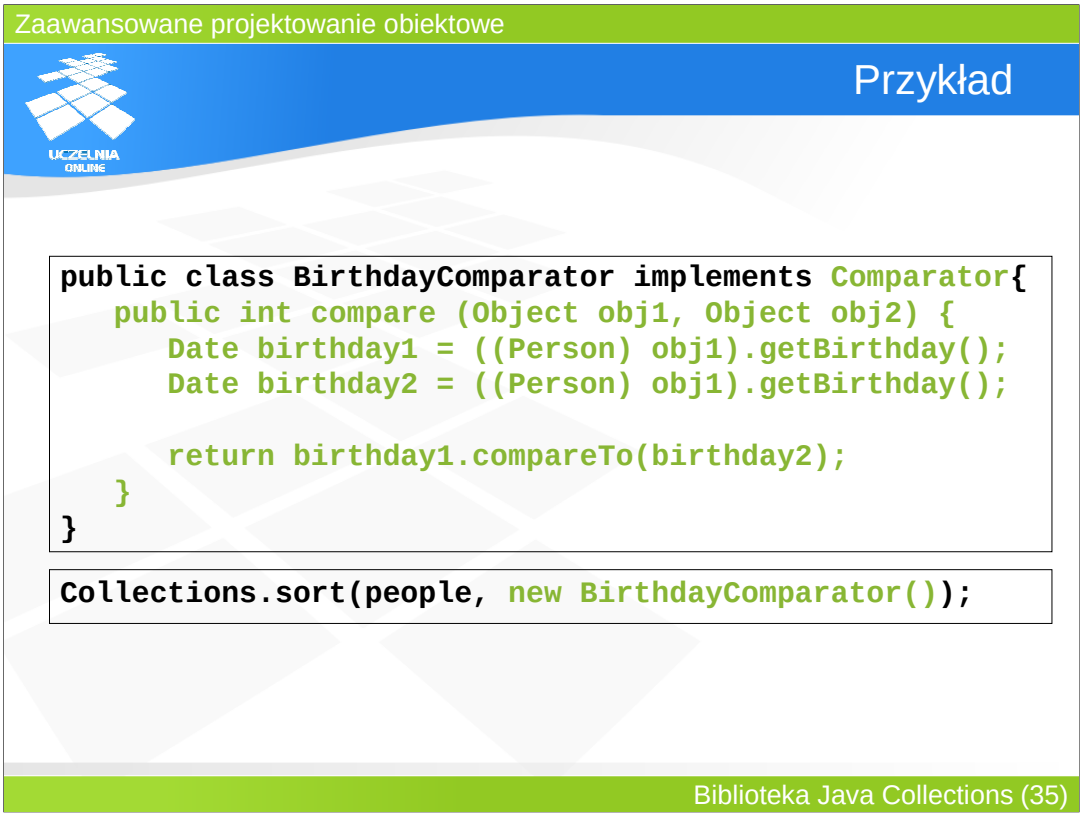
Metoda *compareTo()* określa, czy bieżący obiekt jest "mniejszy" od parametru metody (wynik > 0), "równy" (wynik = 0) czy też "mniejszy" (wynik < 0)

Interfejs Comparator

- Interfejs *Comparator* oferuje podobną funkcjonalność jak *Comparable*, ale porównuje ze sobą dwa obiekty przekazane jako parametry
- *Comparator* definiuje jedną metodę, *compare (Object o1, Object o2)*, analogiczną do metody *Comparable.compareTo(Object o)*
- *Comparator* przekazywany jest jako parametr metod sortujących, wyszukujących etc.
  - *Collections.sort(people, comparator)*
- Sortowanie z wykorzystaniem interfejsu *Comparator* jest możliwe dla dowolnych obiektów

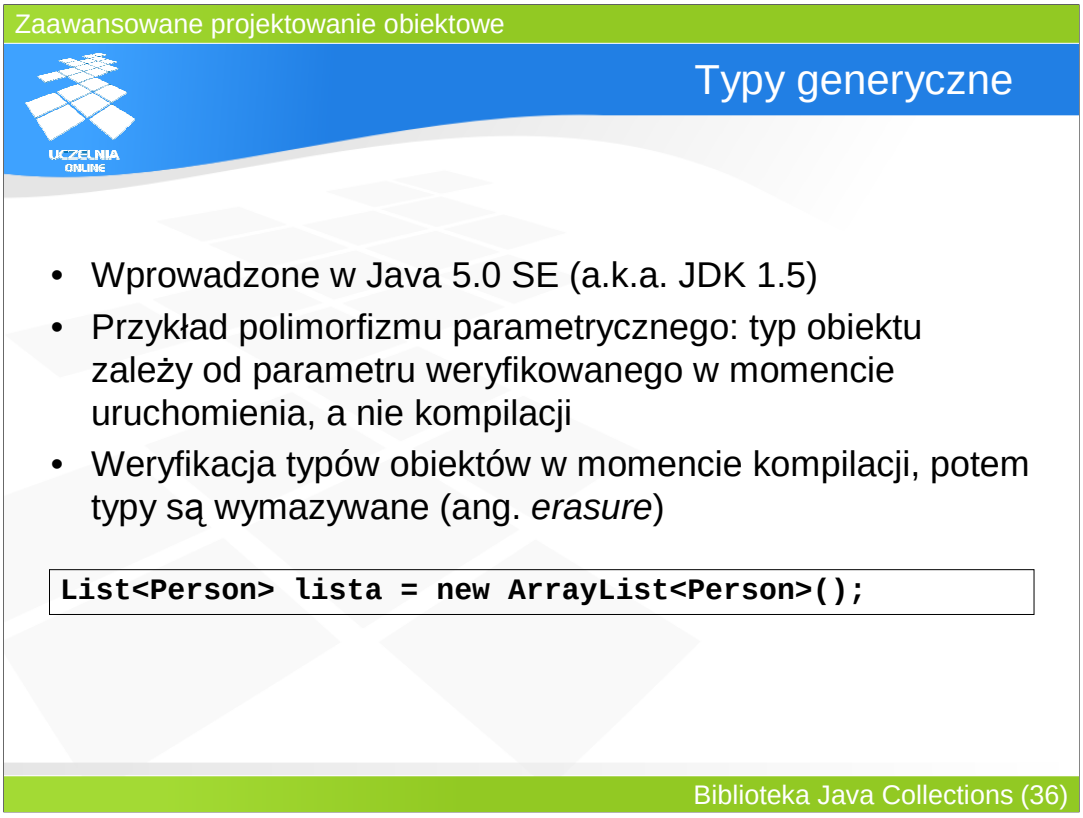
Drugim rozwiązaniem jest zastosowanie interfejsu Comparator. Posiada on tę samą semantykę co Comparable, jednak nie jest implementowany bezpośrednio w obiekcie, lecz jako niezależna klasa. Pozwala on porównywać dwa obiekty przekazane jako parametry zdefiniowanej w nim metody *compare()*, dzięki czemu m.in. nie jest konieczna rekompilacja klasy, która ma być porównywalna.

Sortowanie list obiektów z wykorzystaniem interfejsu Comparator wymaga przekazania obiektu klasy go implementującej jako drugiego parametru metody *sort()*.



Comparator zastosowany w podanym wcześniej przykładzie odczytuje daty urodzenia osób przekazanych jako parametry metody *compare()*. Zwraca ona takie same wyniki, co metoda *compareTo()* z interfejsu Comparable.

Sortowanie polega na przekazaniu utworzonej instancji obiektu implementującego interfejs Comparator – BirthdayComparator.



Najnowsze zmiany wprowadzone w bibliotece Java Collections dotyczyły wykorzystania typów generycznych. Cecha ta pojawiła się w języku w wersji 5.0 (dawniej 1.5) i była jedną z najdłużej oczekiwanych zmian. Częściowo implementuje ona polimorfizm parametryczny: typ obiektu nie musi być ustalany w trakcie kompilacji; jest on parametrem ewaluowanym w momencie uruchamiania programu. Ze względu na konieczność zapewnienia zgodności wstecz z poprzednimi wersjami języka, w Javie zastosowano weryfikację tylko do momentu kompilacji; następnie informacja o typie jest wymazywana, i w trakcie uruchamiania programu jest niedostępna.

Typy generyczne mają szczególnie duże znaczenie dla kolekcji, ponieważ pozwalają definiować typy elementów kolekcji. Od wersji 5.0 języka kolekcje w pełni wykorzystują typy generyczne.

Przykład

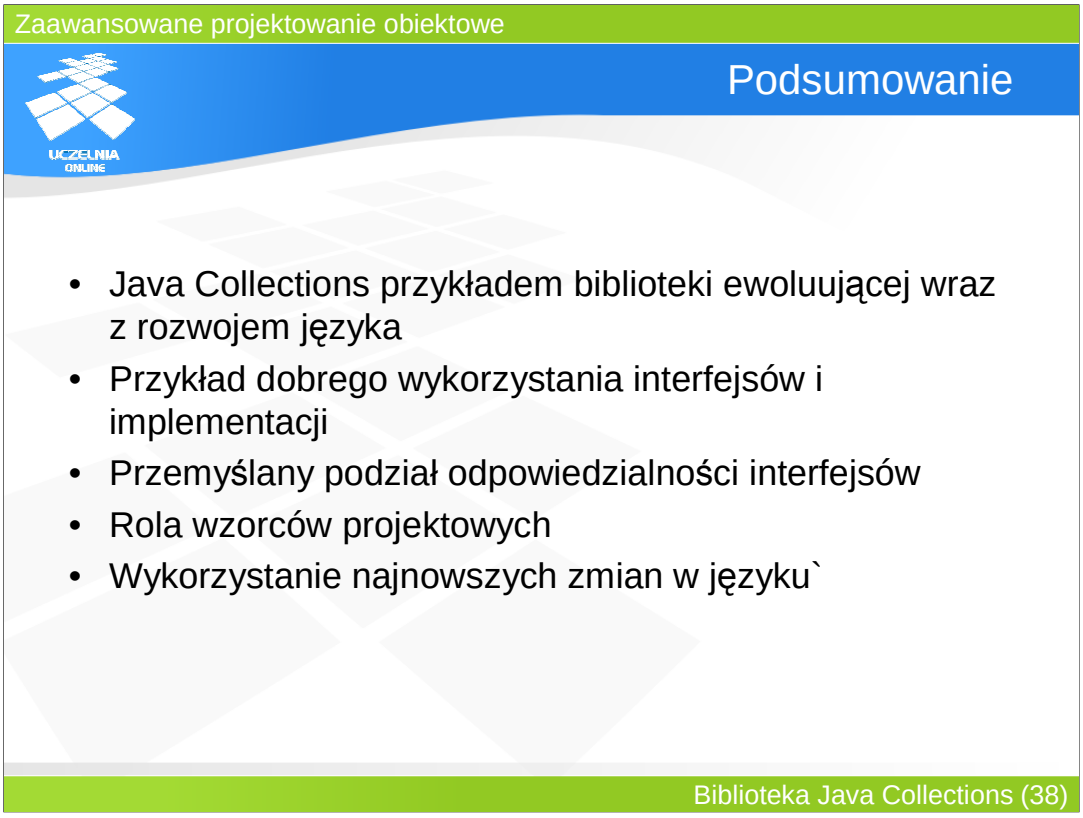
Tradycyjne odwołania do kolekcji

```
List coll = new ArrayList();
for (Iterator iter = coll.iterator(); iter.hasNext(); )
   Person osoba = (Person) iter.next();
   System.out.println("osoba = " + person);
}
```

Wykorzystanie typów generycznych

```
List<Person> coll = new ArrayList<Person>();
for (Iterator<Person> iter = coll.iterator();
     iter.hasNext(); )
   Person osoba = iter.next();
   System.out.println("osoba = " + person);
}
```


Przykład pokazuje, czym różni się kod zapisany tradycyjnie od kodu wykorzystującego typy generyczne. W drugim przypadku, dzięki deklaracji typu Person jako typu elementów listy, nie ma potrzeby rzutowania wyniku metody *next()* iteratora. Kompilator wie, że obiekt zwracany przez tę metodę jest właśnie oczekiwanego typu.



Podczas wykładu prześledzona została ewolucja biblioteki Java Collections i zmiany w jej projekcie. Biblioteka ta ewoluowała wraz ze zmianami w języku, pozostając jednak zgodna wstecz z poprzednimi wersjami. Jest ona przykładem prawidłowego i wyważonego podziału pomiędzy abstrakcję (interfejsy) i implementacje, dzięki czemu istnieją dalsze możliwości jej rozwoju. Projekt biblioteki Java Collections zawartej w JDK 1.2 zawiera kilka wzorców projektowych pozwalających łatwo osiągnąć pożądane właściwości, co wskazuje na ich użyteczność. Najnowsza wersja pozwala na użycie typów generycznych, dzięki którym możliwa jest weryfikacja typów elementów w momencie kompilacji.

Biblioteka ta stanowi dobry przykład podążania przez projektantów za potrzebami programistów z jednej strony i zmianami zachodzącymi w języku – z drugiej.