

Obiektowe Języki Programowania

Kolekcje

Łukasz Rybka · Gdańsk 2015

Co to są tablice?

“ (...) są tworem, który się
konstruuje i wypenia elementami
(...)

— Bruce Eckel

Tablice - podstawowe informacje

Tablice - podstawowe informacje

Tablice odróżniają dwie kwestie: wydajność i typ

Tablice - podstawowe informacje

Tablice odróżniają dwie kwestie: wydajność i typ

Tablica jest prostą sekwencją liniową, która pozwala na szybki dostęp do elementów

Tablice - podstawowe informacje

Tablice odróżniają dwie kwestie: wydajność i typ

Tablica jest prostą sekwencją liniową, która pozwala na szybki dostęp do elementów

Kiedy tworzymy tablicę, jej rozmiar jest ustalony i nie może zostać zmieniony

Tablice - podstawowe informacje

Tablice - podstawowe informacje

Poszczególne elementy wybieramy za pomocą indeksu typu **int**

Tablice - podstawowe informacje

Poszczególne elementy wybieramy za pomocą indeksu typu **int**

Tablica przechowuje referencje do obiektów, nie same obiekty (!)

Tablice - podstawowe informacje

Poszczególne elementy wybieramy za pomocą indeksu typu **int**

Tablica przechowuje referencje do obiektów, nie same obiekty (!)

Dostęp do elementów tablicy jest kontrolowany w runtime - **RuntimeException**

Co to są tablice?

“ Tablica jest (...) najbardziej wydajnym sposobem zapisu i (...) dostępu do sekwencji obiektów (...)

— Bruce Eckel

Polimorficzne struktury danych

“ (...) struktury danych, zawierające obiekty różnych typów są nazywane **polimorficznymi strukturami danych.**

Tablice - podstawowe informacje

Tablice - podstawowe informacje

**W tablicach przechowujemy obiekty danego
typu**

Tablice - podstawowe informacje

W tablicach przechowujemy obiekty danego
typu

Tablica jest pełnoprawnym obiektem -
identyfikator tablicy jest rzeczywistym
odwołaniem do prawdziwego obiektu, który
został stworzony na stercie

Przykład tworzenia tablicy

```
public class Main {  
    public static void main(String[] args) {  
        String[] arr = new String[5];  
  
        arr[0] = "Element 0";  
        arr[1] = "Element 1";  
        arr[2] = "Element 2";  
        arr[3] = "Element 3";  
        arr[4] = "Element 4";  
  
        System.out.println(arr); // ???  
    }  
}
```


Wyświetlanie składników tablicy

```
public class Main {  
    public static void main(String[] args) {  
        String[] arr = new String[5];  
  
        arr[0] = "Element 0";  
        arr[1] = "Element 1";  
        arr[2] = "Element 2";  
        arr[3] = "Element 3";  
        arr[4] = "Element 4";  
  
        System.out.println(Arrays.deepToString(arr));  
    }  
}
```

Uzupełnianie tablicy elementami

```
public class Main {  
    public static void main(String[] args) {  
        String[] arr = new String[10];  
        Random rand = new Random();  
  
        for (int i = 0; i < 5; i++) {  
            arr[rand.nextInt(10)] = String.valueOf(rand.nextInt());  
        }  
  
        System.out.println(Arrays.deepToString(arr));  
        System.out.println("Ilość elementów: " + arr.length); // ???  
    }  
}
```

Pole .length tablicy

Pole `.length` tablicy

**Zawiera informację o ilości elementów w
tablicy**

Pole `.length` tablicy

Zawiera informację o ilości elementów w tablicy

Przy tworzeniu tablicy wszystkie jej elementy są uzupełniane null'ami

Pole `.length` tablicy

Zawiera informację o ilości elementów w tablicy

Przy tworzeniu tablicy wszystkie jej elementy są uzupełniane null'ami

Nie ma prostej (i szybkiej) metody sprawdzenia ilości wypełnionych elementów tablicy

Statyczna inicjalizacja tablicy

```
public class Main {  
    public static void main(String[] args) {  
        String[] arr = {  
            "Element 0",  
            "Element 1",  
            "Element 2",  
            "Element 3",  
            "Element 4"  
        }  
  
        System.out.println(Arrays.deepToString(arr));  
    }  
}
```

Tablice typów prostych

Tablice typów prostych

Tablica nie musi składać się tylko ze złożonych typów

Tablice typów prostych

Tablica nie musi składać się tylko ze złożonych typów

Istnieje możliwość tworzenia tablicy obiektów prostych

Tablice typów prostych

Tablica nie musi składać się tylko ze złożonych typów

Istnieje możliwość tworzenia tablicy obiektów prostych

Kiedy typem tablicy jest obiekt, a dodajemy do niego odpowiadający obiekt typu prostego - następuje zjawisko **autoboxingu**

Statyczna inicjalizacja tablicy

```
public class Main {  
    public static void main(String[] args) {  
        int[] ints = {1, 2, 3, 4, 5, 6, 7, 8, 9};  
        double[] doubles = {1.1, 2.2, 3.3, 4.4, 5.5};  
  
        Integer autoboxed = new Integer[10];  
        for (int i = 0; i < 10; i++) {  
            autoboxed[i] = i * i;  
        }  
  
        System.out.println(Arrays.deepToString(ints));  
        System.out.println(Arrays.deepToString(doubles));  
        System.out.println(Arrays.deepToString(autoboxed));  
    }  
}
```

Tablice wielowymiarowe

Tablice wielowymiarowe

Istnieje możliwość tworzenia tablic o więcej niż jednym wymiarze

Tablice wielowymiarowe

Istnieje możliwość tworzenia tablic o więcej niż jednym wymiarze

Tworzenie tablic wielowymiarowych może być statyczne lub dynamiczne (słowo kluczowe **new**)

Tablice wielowymiarowe

Istnieje możliwość tworzenia tablic o więcej niż jednym wymiarze

Tworzenie tablic wielowymiarowych może być statyczne lub dynamiczne (słowo kluczowe **new**)

Każda para zagnieżdżonych nawiasów klamrowych to osobny wymiar tablicy

Przykład prostej tablicy dwuwymiarowej

```
public class Main {  
    public static void main(String[] args) {  
        int[][] ints = {  
            {1, 2, 3, 4, 5},  
            {6, 7, 8, 9, 10}  
        };  
  
        System.out.println(Arrays.deepToString(ints));  
  
        for (int i = 0; i < ints.length; i++) {  
            for (int j = 0; j < ints[0].length; j++) {  
                System.out.println("Element: " + ints[i][j]);  
            }  
        }  
    }  
}
```

Ragged arrays

Ragged arrays

**Każdy wymiar tablicy może być
inicjalizowany dynamicznie z dowolnym
rozmiarem**

Ragged arrays

Każdy wymiar tablicy może być inicjalizowany dynamicznie z dowolnym rozmiarem

Tablica o różnej ilości elementów w tzw. wektorze (wierszu) nazywana jest **regged** (poszarpana)

Przykład tablicy poszarpanej

```
public class Main {  
    public static void main(String[] args) {  
        Random rand = new Random(47);  
        int[][][] arr = new int[rand.nextInt(7)][][];  
  
        for(int i = 0; i < arr.length; i++) {  
            arr[i] = new int[rand.nextInt(5)][];  
  
            for (int j = 0; j < arr[i].length; j++) {  
                arr[i][j] = new int[rand.nextInt(5)];  
            }  
        }  
  
        System.out.println(Arrays.deepToString(arr));  
    }  
}
```

Po co nam kontenery danych?

“ Program obejmujący wyłącznie
ustaloną liczbę obiektów,
których czas życia jest znany,
jest to program dosyć prosty.

— Bertrand Meyer

Zbiory klas kontenerowych

Zbiory klas kontenerowych

**Wprowadzone (dopracowane) w Java SE 5 do
java.util.***

Zbiory klas kontenerowych

Wprowadzone (dopracowane) w Java SE 5 do
java.util.*

Najważniejsze zbiory:

List

Set

Queue

Map

Kolekcja Collection

Kolekcja Collection

**Grupa odrębnych elementów, podlegających
jakimś regułom**

Kolekcja Collection

Grupa odrębnych elementów, podlegających
jakimś regułom

W jej skład wchodzi typy List, Set czy też
Queue

Kolekcja Collection

Grupa odrębnych elementów, podlegających jakimś regułom

W jej skład wchodzi typy List, Set czy też Queue

Wszystkie implementacje (jak ArrayList) podlegają jednemu interfejsowi

Kolekcja Map

Kolekcja Map

Grupa par obiektów typu klucz-wartość

Grupa par obiektów typu klucz-wartość
Pozwala na wydobywanie wartości dla
znanego klucza

Kolekcja Map

Grupa par obiektów typu klucz-wartość

**Pozwala na wydobywanie wartości dla
znanego klucza**

**Kluczem jest obiekt (analogicznie jak w liście
indeks numeryczny)**

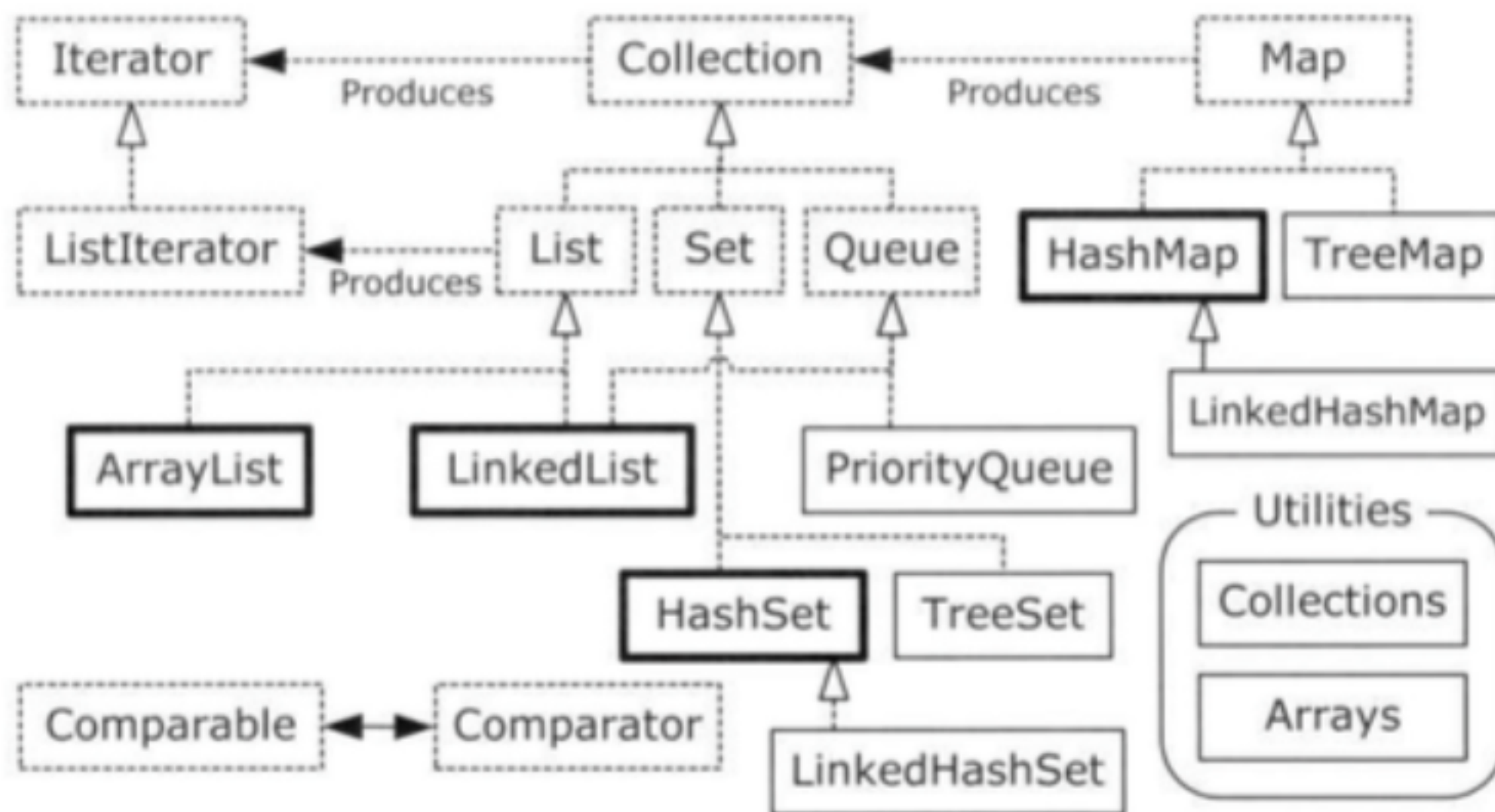
Grupa par obiektów typu klucz-wartość

Pozwala na wydobywanie wartości dla
znanego klucza

Kluczem jest obiekt (analogicznie jak w liście
indeks numeryczny)

Inaczej nazywany **tablicą asocjacyjną** lub
słownikiem

Taksonomia kontenerów



Interfejs List

Obiecuje zachowanie kolejności elementów

Interfejs List

Obiecuje zachowanie kolejności elementów
Uzupełnia interfejs Collections o zestaw metod
pozwalający na wstawianie i usuwanie
elementów do i ze Środka kolekcji

Interfejs List

Obiecuje zachowanie kolejności elementów

Uzupełnia interfejs Collections o zestaw metod pozwalający na wstawianie i usuwanie elementów do i ze Środka kolekcji

Posiada dwa podtypy: **ArrayList** oraz **LinkedList**

Interfejs ArrayList

Interfejs ArrayList

Podstawowy typ kolekcji w Javie

Interfejs ArrayList

Podstawowy typ kolekcji w Javie

Daje swobodny (i "tani") dostęp do elementów w dowolnym miejscu kolekcji

Interfejs ArrayList

Podstawowy typ kolekcji w Javie

Daje swobodny (i "tani") dostęp do elementów w dowolnym miejscu kolekcji

Niższa wydajność przy wstawianiu i usuwaniu elementów z listy

Dawne podejście do kolekcji ArrayList

```
public class Main {  
    public static void main(String[] args) {  
        List strings = new ArrayList();  
  
        strings.add("A");  
        strings.add("B");  
        strings.add("C");  
  
        strings.add(0, "F");  
  
        System.out.println("Size: " + strings.size());  
        System.out.println(strings);  
  
        strings.add(0, new Date()); // ???  
        System.out.println((String) strings.get(0)); // ???  
    }  
}
```

Listy typizowane

```
public class Main {  
    public static void main(String[] args) {  
        List<String> strings = new ArrayList<String>();  
  
        strings.add("A");  
        strings.add("B");  
        strings.add("C");  
  
        strings.add(0, "F");  
  
        System.out.println("Size: " + strings.size());  
        System.out.println(strings);  
  
        strings.add(0, new Date()); // Compilation Error!!  
        System.out.println((String) strings.get(0)); // Zbędne!  
    }  
}
```

Interfejs LinkedList

Interfejs LinkedList

**Optymalny typ kolekcji z dostępem
sekwencyjnym**

Interfejs LinkedList

Optymalny typ kolekcji z dostępem
sekwencyjnym

Zoptymalizowany pod kątem operacji
wstawiania i usuwania elementów ze Środka

Interfejs LinkedList

Optymalny typ kolekcji z dostępem sekwencyjnym

Zoptymalizowany pod kątem operacji wstawiania i usuwania elementów ze Środka

Wolne operacje swobodnego dostępu (np. pobrania elementu o wskazanym indeksie)

Przykład kolekcji LinkedList

```
public class Main {  
    public static void main(String[] args) {  
        LinkedList<String> strings = new LinkedList<>(); // !  
  
        strings.add("A");  
        strings.add("B");  
        strings.add("C");  
  
        strings.addFirst("F");  
        strings.addLast("E");  
  
        strings.add("G");  
        strings.removeFirst();  
        strings.removeLast();  
  
        System.out.println(strings);  
    }  
}
```

Iteratory

Iteratory

**Wprowadzają dodatkowy poziom abstrakcji
dla podstawowych operacji na kolekcjach**

Iteratory

Wprowadzają dodatkowy poziom abstrakcji
dla podstawowych operacji na kolekcjach
**Jednakowe API dla dowolnej kolekcji (czy to
Collection czy też Map!)**

Iteratory

**Wprowadzają dodatkowy poziom abstrakcji
dla podstawowych operacji na kolekcjach
Jednakowe API dla dowolnej kolekcji (czy to
Collection czy też Map!)
Bardzo tanie w stworzeniu**

Iteratory

**Wprowadzają dodatkowy poziom abstrakcji
dla podstawowych operacji na kolekcjach
Jednakowe API dla dowolnej kolekcji (czy to
Collection czy też Map!)
Bardzo tanie w stworzeniu
Ograniczony zestaw operacji**

Iteratory

Wprowadzają dodatkowy poziom abstrakcji
dla podstawowych operacji na kolekcjach
Jednakowe API dla dowolnej kolekcji (czy to
Collection czy też Map!)

Bardzo tanie w stworzeniu
Ograniczony zestaw operacji
Iteracja wyłącznie w jednym kierunku - od
początku do końca

Możliwości iteratorów

Możliwości iteratorów

Wywołując metodę **iterator()** obiektu Collection otrzymujemy iterator z gotowym pierwszym elementem

Możliwości iteratorów

Wywołując metodę **iterator()** obiektu Collection otrzymujemy iterator z gotowym pierwszym elementem

Uzyskanie dostępu do kolejnego obiektu dzięki metodzie **next()**

Możliwości iteratorów

Możliwości iteratorów

Sprawdzenie, czy kolekcja posiada kolejny element - metoda `hasNext()`

Możliwości iteratorów

Sprawdzenie, czy kolekcja posiada kolejny element - metoda **hasNext()**

Usunięcie ostatnio zwróconego elementu metodą **remove()**

Przykład wykorzystania iteratora

```
public class Main {  
    public static void main(String[] args) {  
        List<String> strings = new ArrayList<String>() {{  
            add("A");  
            add("B");  
            add("C");  
        }};  
  
        Iterator<String> iterator = strings.iterator();  
        while(iterator.hasNext()) {  
            System.out.println("Element: " + iterator.next());  
        }  
  
        for (String str : strings) {  
            System.out.println("For each: " + str);  
        }  
    }  
}
```

Interfejs Queue

Kontener FIFO - first-in, first-out

Interfejs Queue

Kontener FIFO - first-in, first-out

Typowa implementacja Queue - LinkedList

Interfejs Queue

Kontener FIFO - first-in, first-out

Typowa implementacja Queue - LinkedList

**Interfejs ten zawiera dodatkowe metody
dostępne za pomocą rzutowania w górę**

Przykład zastosowania kolejki Queue

```
public class Main {  
    public static void main(String[] args) {  
        Queue<Integer> queue = new LinkedList<Integer>();  
        Random rand = new Random(47);  
  
        for (int i = 0; i < 10; i++) {  
            queue.offer(rand.nextInt(i + 10));  
        }  
  
        System.out.print(queue);  
  
        while (queue.peek() != null) {  
            System.out.print(queue.remove() + " - ");  
        }  
  
        System.out.print(queue);  
    }  
}
```

Metody interfejsu Queue

Metody interfejsu Queue

offer() - wstawia element na koniec kolejki
(jeśli to możliwe)

Metody interfejsu Queue

offer() - wstawia element na koniec kolejki
(jeśli to możliwe)

peek() - zwraca element z przodu bez
usuwania lub **false**

Metody interfejsu Queue

offer() - wstawia element na koniec kolejki
(jeśli to możliwe)

peek() - zwraca element z przodu bez
usuwania lub **false**

element() - zwraca element z przodu bez
usuwania lub rzuca **NoSuchElementException**

Metody interfejsu Queue

offer() - wstawia element na koniec kolejki
(jeśli to możliwe)

peek() - zwraca element z przodu bez
usuwania lub **false**

element() - zwraca element z przodu bez
usuwania lub rzuca **NoSuchElementException**

poll() - usuwa i zwraca element z czoła kolejki
lub **false**

Metody interfejsu Queue

offer() - wstawia element na koniec kolejki
(jeśli to możliwe)

peek() - zwraca element z przodu bez
usuwania lub **false**

element() - zwraca element z przodu bez
usuwania lub rzuca **NoSuchElementException**

poll() - usuwa i zwraca element z czoła kolejki
lub **false**

remove() - usuwa i zwraca element z czoła

Interfejs Set

Interfejs Set

Kolekcja, która nie może zawierać więcej niż jednego egzemplarza danej wartości

Interfejs Set

Kolekcja, która nie może zawierać więcej niż jednego egzemplarza danej wartości
Kolekcje typu Set zostały zoptymalizowane pod kątem szybkości wyszukiwania elementu

Interfejs Set

Kolekcja, która nie może zawierać więcej niż jednego egzemplarza danej wartości
Kolekcje typu Set zostały zoptymalizowane pod kątem szybkości wyszukiwania elementu
Posiada dokładnie ten sam interfejs co Collection

Interfejs Set

Kolekcja, która nie może zawierać więcej niż jednego egzemplarza danej wartości

Kolekcje typu Set zostały zoptymalizowane pod kątem szybkości wyszukiwania elementu

Posiada dokładnie ten sam interfejs co

Collection

Dodanie kolejnego identycznego elementu jest ignorowane

Przykład zastosowania kolekcji Set

```
public class Main {  
    public static void main(String[] args) {  
        Random rand = new Random(47);  
        Set<Integer> intset = new HashSet<Integer>();  
  
        for (int i = 0; i < 10000; i++) {  
            intset.add(rand.nextInt(30));  
        }  
  
        System.out.println(intset);  
    }  
}
```


Przykład posortowanej kolekcji TreeSet

```
public class Main {  
    public static void main(String[] args) {  
        Random rand = new Random(47);  
        Set<Integer> intset = new TreeSet<Integer>();  
  
        for (int i = 0; i < 10000; i++) {  
            intset.add(rand.nextInt(30));  
        }  
  
        System.out.println(intset);  
    }  
}
```

Interfejs Map

Daje możliwość odwzorowania obiektów na inne obiekty

Interfejs Map

Daje możliwość odwzorowania obiektów na inne obiekty

Pozwala na przeszukiwanie zadanego klucza (metoda **containsKey()**) oraz wartości (metoda **containsValue()**)

Interfejs Map

Daje możliwość odwzorowania obiektów na inne obiekty

Pozwala na przeszukiwanie zadanego klucza (metoda `containsKey()`) oraz wartości (metoda `containsValue()`)

Istnieje wiele implementacji - do różnych zastosowań i optymalizacji

Przykład wykorzystania interfejsu Map

```
public class Main {  
    public static void main(String[] args) {  
        Random rand = new Random(47);  
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();  
  
        for (int i = 0; i < 10000; i++) {  
            int r = rand.nextInt(20);  
            Integer freq = map.get(r);  
  
            map.put(r, freq == null ? 1 : freq + 1);  
        }  
  
        System.out.println(map);  
  
        for (int i = 0; i < 20; i++) {  
            System.out.println(i + " -> " + map.containsKey(i));  
        }  
    }  
}
```

Interfejs Map - dodatkowe informacje

Interfejs Map - dodatkowe informacje

Kluczami mogą być tylko obiekty, nie typy proste

Interfejs Map - dodatkowe informacje

Kluczami mogą być tylko obiekty, nie typy proste

Wszystkie klucze pobieramy za pomocą metody **keySet()** (Set)

Interfejs Map - dodatkowe informacje

Kluczami mogą być tylko obiekty, nie typy proste

Wszystkie klucze pobieramy za pomocą metody **keySet()** (Set)

Wszystkie wartości pobieramy za pomocą metody **values()** (Collection)

Własna klasa klucza mapy

```
public class CustomKey {  
    private int num;  
  
    public CustomKey(int num) {  
        this.num = num;  
    }  
  
    public void setNum(int num) {  
        this.num = num;  
    }  
  
    public int getNum() {  
        return num;  
    }  
  
    @Override  
    public String toString() {  
        return "CustomKey{" + "num=" + num + '}';  
    }  
}
```

Wykorzystanie klasy CustomKey

```
public class Main {  
    public static void main(String[] args) {  
        Map<CustomKey, Integer> map = new HashMap<>();  
  
        CustomKey key1 = new CustomKey(1);  
        map.put(key1, 1);  
  
        CustomKey key2 = new CustomKey(2);  
        map.put(key2, 2);  
  
        key1 = new CustomKey(1);  
        map.put(key1, -1);  
  
        System.out.println(map); // ???  
    }  
}
```

Metody hashCode() i equals() klasy CustomKey

```
public class CustomKey {  
    // ...  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
  
        CustomKey customKey = (CustomKey) o;  
  
        return num == customKey.num;  
    }  
  
    @Override  
    public int hashCode() {  
        return num;  
    }  
}
```

Metoda equals

Metoda equals

**Musi być zwrotna: dla każdego x , $x.equals(x)$
ma zwracać wartość `true`**

Metoda equals

Musi być zwrotna: dla każdego x , $x.equals(x)$ ma zwracać wartość `true`

Musi być symetryczna: dla dowolnego x i y , $x.equals(y)$ ma zwracać wartość `true` wtedy i tylko wtedy, gdy $y.equals(x)$ zwraca `true`

Metoda equals

Metoda equals

**Musi być przechodnia: dla dowolnych x , y i z ,
jeśli $x.equals(y)$ zwraca `true` oraz $y.equals(z)$
zwraca `true` to także $x.equals(z)$ powinna
zwracać `true`**

Metoda equals

Musi być przechodnia: dla dowolnych x , y i z ,
jeśli $x.equals(y)$ zwraca true oraz $y.equals(z)$
zwraca true to także $x.equals(z)$ powinna
zwracać true

Dla dowolnego x różnego od null wywołanie
 $x.equals(null)$ powinno zwracać false

Metoda equals

Musi być spójna: dla dowolnych x i y wielokrotne wywołania $x.equals(y)$ spójnie zwracają wartość `true` lub `false`, zakładając że żadne informacje używane przy porównywaniu obiektów nie zostały zmienione

Metoda hashCode

Metoda hashCode

**Przetrzymujemy jej wartość w zmiennej
wstępnie uzupełnianej stałą numeryczną
(np. 31)**

Metoda hashCode

Przetrzymujemy jej wartość w zmiennej wstępnie uzupełnianej stałą numeryczną (np. 31)

Dla każdego znaczącego pola wyliczamy osobny hashCode i dodajemy (według wzoru, np. $\text{result} = 31 * \text{result} + c$) do wyniku

Metoda hashCode

Przetrzymujemy jej wartość w zmiennej wstępnie uzupełnianej stałą numeryczną (np. 31)

Dla każdego znaczącego pola wyliczamy osobny hashCode i dodajemy (według wzoru, np. $\text{result} = 31 * \text{result} + c$) do wyniku

Pola nieznaczące to takie, których wartość możemy uzyskać z kombinacji innych pól

Metoda hashCode

Metoda hashCode

Pole boolean: f? 1: 0

Metoda hashCode

Pole boolean: $f ? 1 : 0$

Pole byte, char, short lub int: $(int) f$

Metoda hashCode

Pole boolean: $f ? 1 : 0$

Pole byte, char, short lub int: $(int) f$

Pole long: $(int) (f \wedge (f >>> 32))$

Metoda hashCode

Pole boolean: $f ? 1 : 0$

Pole byte, char, short lub int: $(int) f$

Pole long: $(int) (f \wedge (f >>> 32))$

Pole float: `Float.floatToIntBits(f)`

Metoda hashCode

Metoda hashCode

**Pole double: Double.doubleToLongBits(f) +
zasada dla long**

Metoda hashCode

**Pole double: Double.doubleToLongBits(f) +
zasada dla long**

**Pole obiektowe: dla null: 0, w przeciwnym
wypadku wartość f.hashCode()**

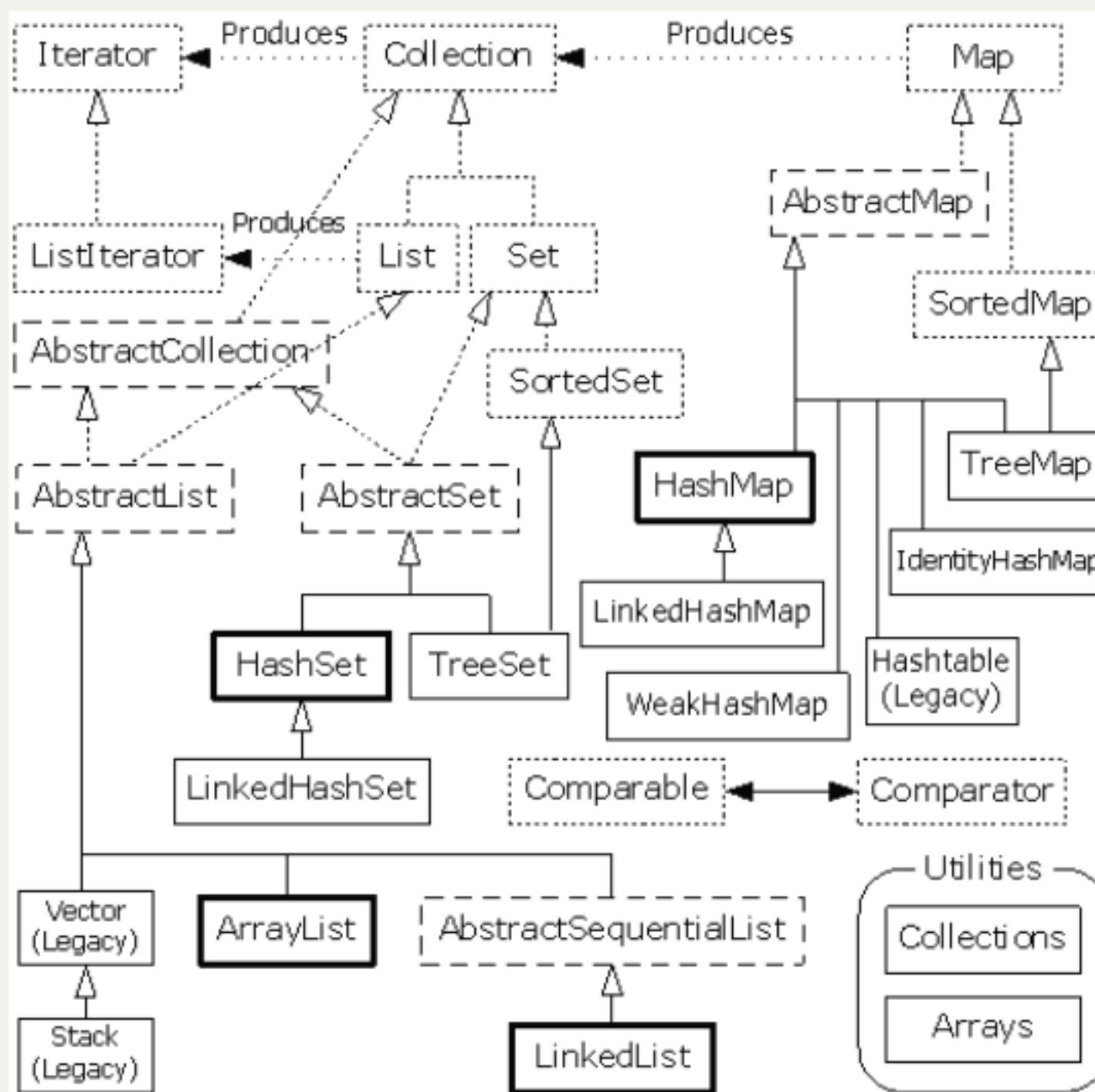
Metoda hashCode

**Pole double: Double.doubleToLongBits(f) +
zasada dla long**

**Pole obiektowe: dla null: 0, w przeciwnym
wypadku wartość f.hashCode()**

**Pole tablicowe: każdy element traktujemy jak
osobne pole obiektu**

Pełna taksonomia kontenerów



ANY
QUESTIONS
?

Pytania?