Obiekty i referencje

Język Java w dużym stopniu był wzorowany na języku C++. Mimo dużych podobieństw składni odmienny jest model organizacji programu i rozlokowanie jego elementów w pamięci.

W programie w języku Java występują wyłącznie klasy i obiekty.

- Każda klasa, nawet jeśli nie jest to jawnie zadeklarowane dziedziczy po wbudowanej klasie `Object`.
- Również tablice są obiektami
- Dostęp do obiektów możliwy jest wyłącznie za pośrednictwem referencji.

Przykład C++

```
class Complex
{
    public:
    Complex(){x=1;y=0;}
    double x,y;
};
// ……..
Complex c;
```

Deklaracja „`Complex c;`” definiuje zmienną typu `Complex`. Przydzielana jest pamięć dla obiektu. Automatycznie wołany jest konstruktor dokonujący inicjacji pól.

Przykład Java

```
class Complex
{
    Complex(){x=1;y=0;}
    double x,y;
}
// ……..
Complex c; //(1)
c = new Complex(); //(2)
Complex c1 = c; //(3)
```

Deklaracja „`Complex c;`” definiuje zmienną typu referencyjnego. Ma ona wartość początkową zerową (`null`). Nie jest przydzielana pamięć dla obiektu typu `Complex` ani nie jest wołany konstruktor (1).

Aby referencja wskazywała jakiś obiekt:

- musi zostać on jawnie stworzony za pomocą operatora `new`, np.: „`c = new Complex()`”. Następnie wynik wywołania operatora może zostać przypisany referencji. (2)
- referencji może być przypisany obiekt wskazywany przez inną referencję, np.: „`c1 = c`” (3);

Przydział pamięci dla elementów programu

- **Rejestry.** Dostęp i operacje na zmiennych, którym przydzielono pamięć wewnątrz rejestrów jest bardzo szybki.
- **Stos.** Pamięć na stosie przydzielana jest poprzez modyfikację rejestru wskaźnika stosu (SP – *stack pointer*). Kiedy pamięć dla zmiennej jest przydzielana, wartość SP odpowiednio zmniejsza się. Kiedy pamięć jest zwalniana, wartość SP zwiększa się.
  - Stos jest wykorzystywany do przesyłania argumentów wywołań funkcji.
  - Aby efektywnie zarządzać pamięcią stosu konieczna jest znajomość rozmiaru składowanych tam danych.
  - Java przechowuje na stosie wyłącznie referencje (obiektów, tzw. interfejsów oraz tablic) a także dane wbudowanych typów prostych (`int`, `double`, `char`,...).
  - Pamięć dla obiektów nie jest przydzielana na stosie, stąd obiektów nie można np.: przekazywać *przez wartość* do funkcji. Możliwe jest jedynie przekazanie referencji do obiektów.
- **Sterta**. Jest to obszar pamięci ogólnego zastosowania. Wszystkie obiekty języka Java tworzone są na stercie.
  - Przydział i zwalnianie pamięci na stercie jest znacznie wolniejszy niż w przypadku stosu, ale zaletą jest rozmiar dostępnej pamięci i duża swoboda dostępu.
  - Obiekty tworzone są na stercie w wyniku wywołania operatora `new`.
  - W języku Java, w odróżnieniu od C++ nie ma potrzeby usuwania obiektów. Rolę tę pełni wbudowany w maszynę wirtualną mechanizm *garbage collection*.

- **Pamięć statyczna**. Zawiera obiekty (dane) które są dostępne przez cały czas życia programu. Przykładem są zmienne globalne w C/C++. Pamięć dla nich jest przydzielona w segmencie danych pliku wykonywalnego.
  - W języku Java tego typu pamięć nie jest używana (nawet dla pól klas typu `static`).
- **Pamięć stałych.** Wartości stałych są umieszczane bezpośrednio w kodzie.
- **Pamięć zewnętrzna.** Stan obiektów może zostać przetłumaczony do postaci strumienia bajtów. W tej formie może zostać przesłany do innej maszyny przy programowaniu rozproszonym (*streamed objects*) lub zapisany na dysk (*persitent objects*).

Wbudowane typy proste

Język Java posługuje się standardowymi typami wbudowanymi.

Typ	Rozmiar	Wartość minimalna	Wartość maksymalna
boolean	—	—	—
char	16-bit	Unicode 0	Unicode 2^{16}- 1
byte	8-bit	-128	+127
short	16-bit	-2^{15}	$+2^{15}$—1
Int	32-bit	-2^{31}	$+2^{31}$—1
long	64-bit	-2^{63}	$+2^{63}$—1
float	32-bit	$IEEE_{754}$	$IEEE_{754}$
double	64-bit	$IEEE_{754}$	$IEEE_{754}$

-

- W odróżnieniu od języka C/C++ gdzie reprezentacja wewnętrzna typów wbudowanych uzależniona jest od architektury sprzętowej, w języku Java ściśle definiuje się ich rozmiar oraz organizację pamięci (kolejność bajtów, bitów). Pozwala to na pełną niezależność od platformy sprzętowej.
- Dane typów wbudowanych mogą być atrybutami klas, zmiennymi lokalnymi lub parametrami funkcji.
- Pamięć dla zmiennych lokalnych i argumentów funkcji typów wbudowanych przydzielana jest na stosie, stąd:
  - nie trzeba ich tworzyć za pomocą operatora `new`; deklaruje się je jak zwykłe zmienne automatyczne, np.: `double x; int n;`
  - jeśli są parametrami funkcji, wówczas argumenty przekazywane są przez wartość, a nie przez adres (referencję)
- Dla każdej z klas typów wbudowanych zapewniono tzw. klasy opakowujące (`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`).
  - Obiekty tych klas są tworzone za pomocą operatora `new`.
  - Statyczne metody tych klas zazwyczaj zapewniają rozmaite konwersje pomiędzy zapisem tekstowym i reprezentacją liczbową. (odpowiedniki funkcji `atoi`, `sscanf`, `itoa` języka C).

Tablice

Tablice w języku Java są również obiektami. Dostęp do nich realizowany jest za pośrednictwem referencji.

Przykład C/C++

```
int tab[7];
```

Przydział pamięci dla 7-elementowej tablicy `int`

```
int b[];
```

Deklaracja symbolu `b`, tablica musi być zdefiniowana w innym miejscu.

Przykład Java

```
int []tab; // lub int tab[];
```

Deklaracja referencji do tablicy zawierającej elementy `int`;

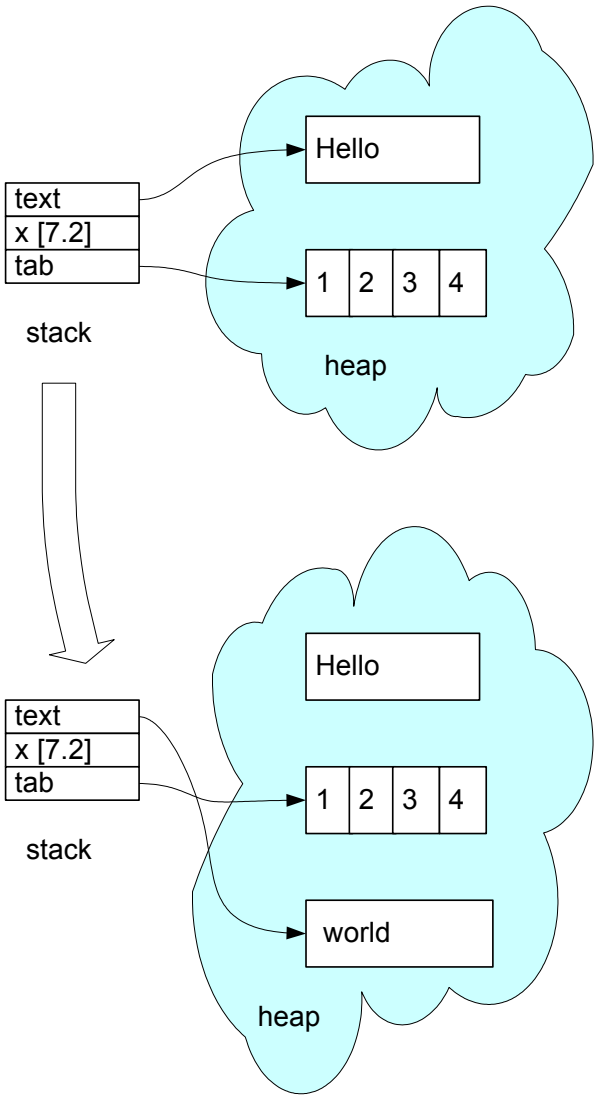
```
tab = new int[7];
```

Utworzenie obiektu typu tablicowego i przypisanie jego adresu referencji `tab`.

Tablice w języku Java mogą zawierać wyłącznie dane typów wbudowanych lub referencje.

Przydział pamięci w języku Java – podsumowanie

```
public class Test
{
   public static void main(String args[])
   {
      String text = "Hello";
      System.out.print(text);
      double x=7.2;
      int[] tab={1,2,3,4};

      text = " world";

      System.out.println(text);
   }
}
```



Elementy składni języka

Javadoc

Komentarz blokowy postaci

/** ... */

ma specjalne znaczenie - zawiera instrukcje dla programu javadoc.

```
public class HalogenLight {

    private double voltage;
    /**
     * Puts the light on
     */
    public void on(){
        voltage=230;
    }
    /**
     * Puts the light off
     */
    public void off(){
        voltage=0;
    }
    /**
     * Dims the light by subtracting v from voltage
     * @param v - value to subtract
     */

    public void dim(double v){
        voltage=voltage-v>=0?voltage-v:0;
    }
    /**
     * Brightens the light by adding v to voltage
     * @param v - value to add
     */
    public void brighten(double v){
        voltage=voltage+v<=230?voltage+v:230;
    }
}
```

```
public class HalogenLight
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description
`HalogenLight()`

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`void`	`brighten(double v)` Brightens the light by adding v to voltage
`void`	`dim(double v)` Dims the light by subtracting v from voltage
`void`	`off()` Puts the light off
`void`	`on()` Puts the light on

Unicode

Programy w języku Java są pisane w zestawie znaków UNICODE. Standard UNICODE reprezentuje znaki jako liczby 16-bitowe bez znaku (0–65535). Pierwszych 128 znaków pokrywa się z zestawem znaków ASCII. Kolejne wartości reprezentują znaki należących do różnych języków.

- Dzięki zastosowaniu standardu UNICODE możliwe jest umieszczenie w jednym dokumencie tekstów pisanych w różnych językach.
- Mimo, że kompilator języka Java przetwarza tekst programów jako ciąg znaków UNICODE, możliwe jest posługiwanie się zestawem znaków ASCII. Znaki UNICODE mogą być wyrażone w stałych znakowych, tekstach oraz identyfikatorach jako *sekwencje escape* postaci `\uxxxx`, gdzie `xxxx` jest zapisanym szesnastkowo kodem znaku.

Przykład 1

```
public static void main (String[] args)
    {

        String a = "tr\u0105ba";
        String b = "trąba";
        if(a.equals(b))
            System.out.println("identyczne");
    }
```

Przykład 2 – Znaki UNICODE w identyfikatorach

```
class Test001
{
    static void wypiszTrąba()
    {
        System.out.println ("trąba");
    }
    public static void main (String[] args)
    {

        wypiszTr\u0105ba();

    }
}
```

Identyfikatory

Składnia identyfikatorów wymaga, aby pierwszym znakiem była litera UNICODE, znak podkreślenia (\u005f) lub znak $ (\u0024). Kolejnymi znakami mogą być dodatkowo cyfry.

Przykład

```
int $x=7;
double _x = 8;
int tr\u0105ba5=7;
trąba5++;
```

Zastosowanie zestawu znaków UNICODE pozwala na stosowanie nazw w językach narodowych (w tym np.: w językach azjatyckich).

Stosowanie identyfikatorów w językach narodowych ma jednak swoje ograniczenia. Większość edytorów tekstowych to edytory ASCII. Translacja plików ASCII, w którym pojawiają się jawnie znaki narodowe (np.: `trąba5`) do UNICODE powiedzie się w środowisku, dla którym wybrano jako *locale* język polski, natomiast może dać błędne rezultaty dla środowiska, w którym wybrano język zachodnioeuropejski.

Styl tworzenia identfikatorów

W języku Java przyjęto następującą konwencję:

- nazwy klas i interfejsów mają postać rzeczownika z ewentualnymi dopełnieniami. Wyrazy są pisane dużymi literami.
  np.: `InputStream`, `DataInputStream`, `StringBuffer`, `ClassLoader`
- nazwy metod rozpoczynają się od malej litery i zazwyczaj zawierają czasownik. Metody odczytujące albo zmieniające wartości powinny nazywać się `getXXX()` oraz `setXXX()`. Metody zwracające rozmiar powinny nazywać się `length()`. Metody dokonujące konwersji typów powinny nazywać się `toXXX()`, np.: `toString()`.
- nazwy pól klas zwierają rzeczownik z dopełnieniami i rozpoczynają się od małej litery. Pozostałe słowa składające się na nazwę pisane są z dużej litery.
- nazwy stałych symbolicznych pisane są dużymi literami. Słowa są oddzielane podkreśleniami (np.: `MIN_VALUE`, `NORTH`, `RED`).

Klasa String

Klasa `String` odgrywa w języku Java szczególną rolę: dla każdego typu danych (wbudowanego lub referencji) możliwa jest automatyczna konwersja do typu `String`.

Dla typów wbudowanych konwersja tworzy tekstową postać wartości. Dla referencji wołana jest metoda `toString()` zdefiniowana w klasie `Object` (lub przedefiniowana w klasach potomnych). Zwraca ona referencję do obiektu klasy `String` o zawartości reprezentującej obiekt wskazywany przez referencję.

Obiekty klasy String są niemodyfikowalne.

- Raz utworzony obiekt nie może zmienić swojej zawartości.
- Metody klasy String nie modyfikują go, ale zwracają nowy obiekt, np.:
  - `String substring(int beginIndex, int endIndex)` – wydziela podciąg
  - `String toLowerCase()` – zamienia litery małe na duże

StringBuilder

Klasą przechowującą teksty i pozwalającą na modyfikowanie ich zawartości jest `StringBuilder`. Implementuje takie przeciążone metody, jak:

- `append()` – dodawanie na końcu
- `insert()` – wstawianie
- `delete()` – usuwanie

Przykład

```
static String toString(double[] array){
    StringBuilder b = new StringBuilder();
    b.append("[");
    for(int i=0;i<array.length;i++){
        if(i!=0)b.append(",");
        b.append(Double.toString(array[i]));
    }
    b.append("]");
    return b.toString();
}
```

```
double[]t = {1,2,3,3.5,4};
String txt = toString(t);
System.out.println(txt);
```

Wynik: `[1.0,2.0,3.0,3.5,4.0]`

Literały łańcuchowe

- Łańcuchy znakowe składają się z dowolnej liczby znaków.
- Specyfikuje się je przez umieszczenie tekstów w znakach cudzysłowu (`"..."`)
- Wewnątrz łańcucha mogą pojawić się dowolne znaki lub sekwencje *escape*.
- Można stosować sekwencje escape unicode , np. `\u03a9`

Przykład

```
""                      /* pusty tekst */
"\n"                    /* tekst zawierający znak nowej linii
                           */
"To jest łańcuch"       /* tekst */
"Ala ma " + "kota"      /* dwa literały łańcuchowe
                           reprezentowane przez jeden obiekt
                           klasy String*/
```

Łańcuchy znakowe umieszczone wewnątrz kodu są traktowane jak referencje do obiektów klasy `String`.

Instancje klasy `String` o tworzone dla literałów łańcuchowych o danej wartości są unikalne.

Przykład

```
public static void main(String[] args) {

❶  String text = "Tekst";
❷  System.out.println("Tekst" == "Tekst");
❸  System.out.println( text == "Tekst");
❹  String txt2 = new String("Tekst");
❺  System.out.println(txt2 == "Tekst");
❻  System.out.println("Tekst".equals(txt2));
❼  System.out.println(
   ("Ala ma " + "kota") == "Ala ma kota");
}
```

❶ Tworzony jest obiekt klasy `String` zawierający napis «`Tekst`»; Deklarowana jest referencja `text` wskazująca ten obiekt.

❷ Operator == dla argumentów będących referencjami sprawdza, czy wskazują ten sam obiekt. Wypisane zostanie `true`.

❸ Referencja `text` wskazuje ten sam obiekt, co referencja `"Tekst"`. Wypisane zostanie `true`.

❹ Stworzony zostanie nowy obiekt klasy `String` i zaincjowany wartością obiektu wskazanego przez referencję `"Tekst"`.

❺ Referencja `txt2` i referencja `"Tekst"` nie wskazują tego samego obiektu. Wypisane zostanie `false`.

❻ Metoda `equals` sprawdza czy obiekty są równoważne, np.: dla obiektów klasy `String` porównywana jest ich zawartość. Metoda jest wołana dla obiektu identyfikowanego przez referencję `"Tekst"`. Porównywana jest zawartość obiektu wskazywanego przez `txt2`. Wypisane zostanie `true`.

❼ Wynik konkatenacji literałów łańcuchowych `"Ala ma " + "kota"` jest referencją do tego samego obiektu, co literał `"Ala ma kota"` Wypisane zostanie `true`.

Operator String +

Operator + ma specyficzne działanie dla argumentów typu `String` (referencji). Dokonuje on konkatenacji tekstów tworząc automatycznie nowy obiekt klasy `String` i zwracając do niego referencję.

Przykład

```
public static void main(String[] args)
{
  String h="Hello ";
  String w="world";
  String text = h+w;
  System.out.println(text);
  // wypisze Hello world
}
```

- Jeżeli jeden z argumentów operatora + jest referencją typu `String`, natomiast drugi typem wbudowanym lub referencją, wówczas zostanie on automatycznie przekształcony do typu `String`.

Przykład

```
class Test
{
static class X
{
  int i=7;
  public String toString(){
    return "i="+i;
  }
}

public static void main(String[] args)
{

  double x=5.2;
  System.out.println("x="+x );
  System.out.println("X[" + new X()+"]" );
}
}
```

Wynik:

```
x=5.2
X[i=7]
```

Przydatne metody klasy String

format

`static String format(String format, Object... args)` – metoda statyczna będąca odpowiednikiem `sprintf()` w C/C++.

Przykład:

```
double angle = 45*Math.PI/360;
String text = String.format(Locale.US,
                    "sin(%f) = %f",angle, Math.sin(angle));
System.out.println(text);
```

Wypisze:

`sin(0.392699) = 0.382683`

`String.format()`, `Math.sin()` – wywołanie metody statycznej -- w C++ byłoby `String::format()`, `Math::sin()`

`Locale.US` – jest to stała (opcjonalna) sterująca sposobem przekształcania liczb do reprezentacji tekstowej.

matches

`boolean matches(String regex)` —testuje, czy tekst odpowiada wzorcowi (wyrażenie regularne)

replace i replaceAll

`String replaceAll(String regex, String rplc)` – zamienia fragment tekstu pasujące do wzorca, zwraca wynik w postaci nowego obiektu

split

`String[] split(String regex)` – traktuje fragmenty pasujące do wzorca jako separatory, umieszcza rozdzielane nimi symbole w tablicy

Przykład

```
String in = "  Ala , 12 , 35";
//String[] tab = in.split("^(\\s*)|(\\s*,\\s*)");
String[] tab = in.split("[^a-zA-Z0-9]+");
for(String s:tab){
   System.out.printf("\'%s\'\n",s);
}
```

Wypisze:

```
''
'Ala'
'12'
'35'
```

Operatory

Operatory w języku Java są niemal identyczne, jak w C++.

Operator rzutowania

Jeżeli zmiennej danego typu przypisujemy wartość innego typu, wówczas możliwa jest automatyczna konwersja, jeśli typ zmiennej stojącej po lewej stronie jest większy (lub bardziej ogólny) niż typ po prawej stronie. Ten rodzaj konwersji nazywany jest konwersją rozszerzającą (ang. *widening conversion*).

Przykład

```
byte a =10;
int b = a;
long c = b;
double d = c;
```

Pragnąc przypisać wartość większego typu do zmiennej mniejszego typu konieczna jest jawna konwersja za pomocą operatora rzutowania. Ten typ konwersji nazywany jest konwersją zawężającą (ang. *narrowning conversion*), ponieważ wartość należąca do większego typu jest jawnie przekształcana do wartości należącej do mniejszego typu.

Przykład

```
double d=2.37;
int b = (int)d;
```

Składnia operatora rzutowania:
(*Type*)*Expression*

- Operator rzutowania konwertuje podczas wykonania programu typ numeryczny w inny typ numeryczny lub konwertuje zmienną typu referencyjnego *Expression* do typu referencyjnego *Type*.
- Jeżeli konwersja typów referencyjnych byłaby niemożliwa, błąd jest sygnalizowany za pomocą wyjątku typu `ClassCastException`.

Operator instanceof

Operator jest dwuargumentowym operatorem relacyjnym (stąd jego rezultatami są wartości typu *boolean*). Służy on do porównania typu zmiennej referencyjnej ze wskazanym typem referencyjnym.

Operator jest zdefiniowany jako

Reference `instanceof` *ReferenceType*

Reference

Wyrażenie obliczane jako zmienna typu referencyjnego lub `null`.

ReferenceType

Typ referencyjny. W języku Java typem referencyjnym jest nazwa klasy, interfejsu lub typ tablicowy. Nie można bezpośrednio zadeklarować (zdefiniować) obiektu klasy.

Operator zwraca wartość `true`, jeżeli referencja *Reference* ma wartość różną od `null` oraz można ją zrzutować na *ReferenceType.*

W pozostałych przypadkach operator zwraca `false`.

Przykład

```
public static void main(String[] args)
{
  System.out.println("Test" instanceof String);

  String s="Test";
  Object o = s;
  System.out.println(o instanceof String);
  if(o instanceof String)
    System.out.println((String)o);

  int[] a=new int[6];
  System.out.println(a instanceof int[]);
}
```

Wynik: `true true Test true`

Instrukcje

Instrukcja for-each

Instrukcja pętli określana jako *for-each* pojawiła się w wersji 1.5 (5.0) języka Java. Pozwala ona na iterację po elementach kolekcji bez jawnego użycia iteratora (klasy Iterator lub zmiennej iteracyjnej).

Składnia:

for(*Type var* : *collection*)*statement*

- *var* jest deklarowaną wewnątrz pętli zmienną typy *Type*
- *collection* jest dowolnym kontenerem zawierającym elementy typu *Type*

Instrukcja może być odczytana jako zapis odpowiadający typowej konstrukcji w pseudokodzie:

foreach *var* **in** *collection* **do** *statement*.

Wprowadzając nową konstrukcję zrezygnowano ze słów kluczowych `foreach` oraz `in`, aby zachować kompatybilność z poprzednimi wersjami języka, które akceptowały te symbole jako identyfikatory.

Przykład 1

```
int []tab ={1,2,3,4};
for(int x : tab)System.out.println(x);
```

Wypisze `1, 2, 3, 4`

Przykład 2

```
int sum=0;
for(int x:tab){
    sum+=x;
    System.out.println(x);
}
System.out.println(sum);
```

Przewaga stosowanie pętli *for-each* jest szczególnie widoczna w przypadku konieczności iteracji po kolekcjach typów generycznych.

```
void cancelAll(Collection<TimerTask> c) {
for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
    i.next().cancel();
}
```

```
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask t : c) t.cancel();
}
```

Przykład 3

```
List<String> l = new ArrayList<String>();
for(int i=0;i<10;i++)l.add(String.format("item#%d", i));
for(String s:l)System.out.println(s);
```

Wydrukuje

```
item#0
item#1
item#2
item#3
...
```

Przykład 4

```
import java.util.*;
//…
Map<String, Integer> dict = new TreeMap<String, Integer>();
dict.put("zero",0);
dict.put("one",1);
dict.put("two",2);
dict.put("three",3);

for(Map.Entry<String, Integer> x : dict.entrySet()){
    System.out.println(x);
}
```

Wydrukuje

```
one=1
three=3
two=2
zero=0
```

`TreeMap<String, Integer>` implementacja odwzorowania typu *map* w postaci drzewa.

`Map.entrySet()` – zwraca zbiór par `Map.Entry<String, Integer>` będących elementami kolekcji

Etykietowane instrukcje break i continue

W języku C/C++ występuje instrukcja `goto`. Pozwala ona na przejście do etykietowanej instrukcji wewnątrz funkcji. Jedynym racjonalnym powodem użycia `goto` jest konieczność opuszczenia, co najmniej dwóch bloków instrukcji (np.: podwójnej pętli lub instrukcji `switch-case` umieszczonej w pętli).

W języku Java tę rolę powierzono etykietowanym instrukcjom `break` i `continue`.

Etykietowane instrukcje

Dowolnej instrukcji może zostać nadana etykieta postaci

label: *Statement*

W odróżnieniu od języka C++ identyfikator *label* nie musi być unikalny wewnątrz metody. Jedynym ograniczeniem jest, aby nie stosować tych samych etykiet do instrukcji zagnieżdżonych (np.: bloku pętli wewnątrz innego bloku).

Etykietowana instrukcja continue

Składnia

`continue` *label*;

Etykieta *label* musi być przypisana zewnętrznej instrukcji blokowej (niekoniecznie najbliższej).

W wyniku wykonania instrukcji sterowanie zostanie przekazane do etykietowanej instrukcji, którą **musi być instrukcja iteracyjna** (`while`, `do` lub `for`). Rozpoczęta zostanie nowa iteracja.

Przykład:

```
mainloop:
for(int i=0;i<20;i++){
  for(int j=0;;j++){
    if(i==j){
      System.out.println("");
      continue mainloop;
    }
    System.out.print(".");
  }
}
```

Etykietowana instrukcja break

Składnia

`break` *label*;

W wyniku wykonania instrukcji sterowanie zostanie przekazane do instrukcji następnej po etykietowanej symbolem `label`. **Instrukcja nie musi być iteracyjna.**

Przykład 1

```
labelled:
  {
    for(int i=0;i<5;i++){
      System.out.print(i+" ");
      if(i==3)break labelled;
    }
    System.out.println("Unreachable");
  }
  System.out.println("After labelled");
```

Wypisze:

```
0 1 2 3 After labelled
```

Przykład 2

```
public static void main(String[] args) {
  mainloop:
  for(;;){
    System.out.println( "Options:\n”+
      ”a - option A\n"+
      "b - option B\nq - quit\n");
    for(;;){
      int c;
      try{
        c = System.in.read();
        if(c<0)break mainloop;
      }
      catch( IOException e){break mainloop;}
      switch(c){
        case 'q':
        case 'Q':     break mainloop;
        case 'a':
        case 'A':
          System.out.println("Selected: a");
          continue mainloop;
        case 'b':
        case 'B':
          System.out.println("Selected: b");
          continue mainloop;
        case '\n':
        case '\r':  break;
        default:
          System.out.println("Unknown option");
          continue mainloop;
      }
    }
  }
  System.out.println("Thank you");
}
```

Tablice

Tablice są ciągami danych typów wbudowanych lub referencji.

- Elementy ciągu są tego samego typu (chociaż w przypadku referencji, klasy rzeczywistych obiektów wskazywanych przez referencje mogą się różnić).
- Ciąg elementów jest oznaczany wewnątrz programu wspólnym identyfikatorem (nazwą tablicy).
- Operator `[ ]` umożliwia dostęp do elementu tablicy.

Aby zadeklarować tablicę posługujemy się następującą składnią:

```
Type[] table;
```

lub w stylu C/C++

```
Type table[];
```

Deklaracja ta definiuje zmienną typu tablicowego, ale nie pociąga za sobą przydziału pamięci dla elementów tablicy. Typem zmiennej `table` jest typ referencyjnego `Type[]`; zmienna ma początkową wartość równą `null`.

- Przydział pamięci odbywa się przez stworzenie na stercie obiektu typu tablicowego. W tym celu należy użyć operatora `new` oraz określić rozmiar tablicy. Rozmiar ten jest zapisany w niemodyfikowalnym polu obiektu `length`.
- Tablice są indeksowane od 0 do `length-1`. Próba sięgnięcia do elementu spoza tego zakresu zakończy się błędem: wygenerowaniem wyjątku `IndexOutOfBoundsException`.

Przykład:

```
int[] table;
table = new int[4];
table[0]=0;
table[1]=4;
table[2]=5;
table[3]=2;
for(int i=0;i<table.length;i++){
  System.out.println(table[i]);
}
```

Jeżeli nie nadamy wartości elementom tablicy, wówczas będą miały one standardowe wartości zerowe.

Język Java dopuszcza deklarację tablicy wraz z inicjalizacją (stworzenie obiektu i wypełnienie wartościami)

Przykład:

```
int[] table={0,4,5,3};
for(int i=0;i<table.length;i++){
  System.out.println(table[i]);
}
```

Równocześnie z deklaracją referencji `table` na podstawie listy inicjalizacyjnej zostanie obliczony rozmiar tablicy, przydzielona pamięć dla obiektu na stercie i przeprowadzona inicjalizacja.

Tablica może zostać w podobny sposób zainicjowana podczas tworzenia jej za pomocą operatora `new`.

Przykład:

```
int[] table;
table = new int[]{0,4,5,3,};
for(int i=0;i<table.length;i++){
  System.out.println(table[i]);
}
```

Tablice obiektów

Jeżeli utworzymy tablicę obiektów, wówczas w rzeczywistości będzie ona zawierała jedynie referencje do obiektów.

Przykład:

```
Integer[] table;
table = new Integer[4];
for(int i=0;i<table.length;i++){
  System.out.print(table[i]+" ");
}
```

Rezultat: `null null null null`

Pełna inicjalizacja tablicy wymaga inicjalizacji referencji będących elementami tablicy.

Przykład

```
Integer[] table;
table = new Integer[4];

table[0]=new Integer(0);
table[1]=new Integer(4);
table[2]=new Integer(5);
table[3]=new Integer(2);

for(int i=0;i<table.length;i++){
  System.out.print(table[i]+" ");
}
```

Obiekty można stworzyć i umieścić na liście inicjalizacyjnej podobnie, jak dla typów prostych.

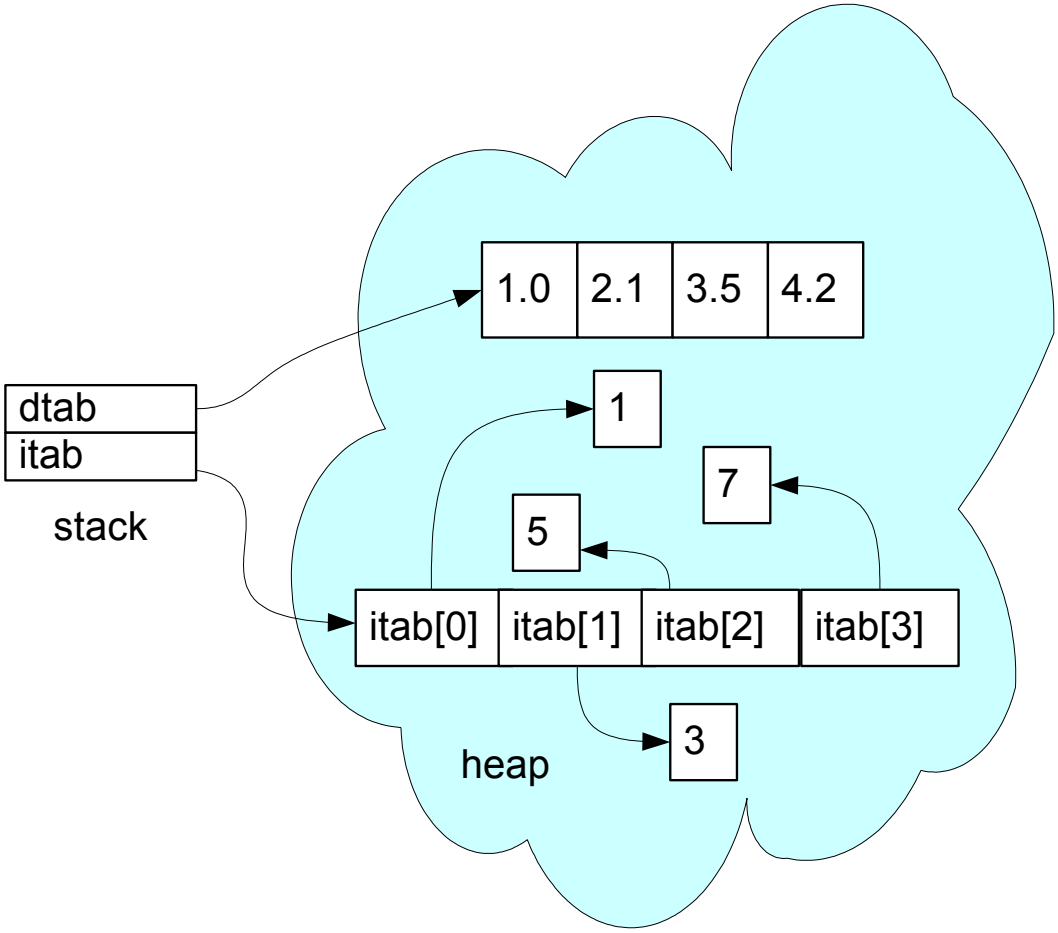
Przykład

```
  Integer[] table={
    new Integer(0),
    new Integer(4),
    new Integer(5),
    new Integer(2),
  };

// lub Integer[] table = new Integer[]{…}

  for(int i=0;i<table.length;i++){
    System.out.print(table[i]+" ");
  }
```

Wypisze: `0 4 5 2`

Przydział pamięci dla elementów tablic

```
public class Test
{
   public static void main(String[] args)
   {
      double[] dtab={1.0,2.1,3.5,4.2};
      Integer[] itab=new Integer[4];
      for(int i=0;i<4;i++){
         itab[i]=new Integer(2*i+1);
      }
   }
}
```



Tablice jako argumenty funkcji

W języku Java argumenty są przekazywane do funkcji (metod) wyłącznie przez wartość. Jeżeli przekażemy do funkcji wartość referencji wskazującej zewnętrzny obiekt (na przykład tablicę), wówczas wewnątrz funkcji możemy odczytać lub zmodyfikować jego zawartość.

Wywołując funkcję z parametrem typu tablicowego, można do niej przekazać grupę obiektów. W zależności od argumentów wywołania, liczba i typ obiektów w grupie może się zmieniać.

Przykład 1

```
public class Test{
public static void main (String[] args)
{
  for(int i=0;i<args.length;i++){
    System.out.print(args[i]+" ");
  }
}
}
```

W wyniku wywołania programu z argumentami „*Hello*” i „*world*”:

```
> java Test Hello world
```

Na ekranie zostanie wypisany tekst „`Hello world`”

Przykład 2

```
class Test{
static void print(Object[] objects)
{
  for(int i=0;i<objects.length;i++){
    System.out.println(objects[i]);
  }
}

public static void main(String[] args)
{
  print(new Object[]
  {
    new Integer(12),
    "Ala ma kota",
    new Double(3.14),
  }
  );
}
}
```

Rezultat:

```
12
Ala ma kota
3.14
```

- W powyższym przykładzie do funkcji `print` przekazana zostaje tablica referencji typu `Object`. (Klasa `Object` jest klasą bazową hierarchii klas w języku Java. Każda klasa dziedziczy po tej klasie.)
- Referencje wskazują jednak obiekty różnych typów (`Integer`, `String`, `Double`).
- Dla każdego z obiektów identyfikowanych przez referencję wołana jest metoda `toString()`; jej rezultat wypisywany jest na standardowym wyjściu.

Tablice wielowymiarowe

Tworzenie tablic wielowymiarowych obejmuje podobne etapy jak dla tablic jednowymiarowych: należy zadeklarować referencję odpowiedniego typu, stworzyć obiekt tablicowy i wypełnić go danymi.

Tablice wielowymiarowe są implementowane przez dynamicznie tworzone struktury danych. Tablica n-wymiarowa jest w rzeczywistości tablicą referencji do tablic n-1 wymiarowych. Pozwala to na implementację tablic zawierających wektory różnej długości.

Przykład

```
int[][] a ={
    {1,2,3},
    {4,5}
};
for(int i=0;i<a.length;i++){
    for(int j=0;j<a[i].length;j++)
        System.out.print(a[i][j]+" ");
    System.out.println();
}
```

Wypisze

```
1 2 3
4 5
```

W omawianym przykładzie:

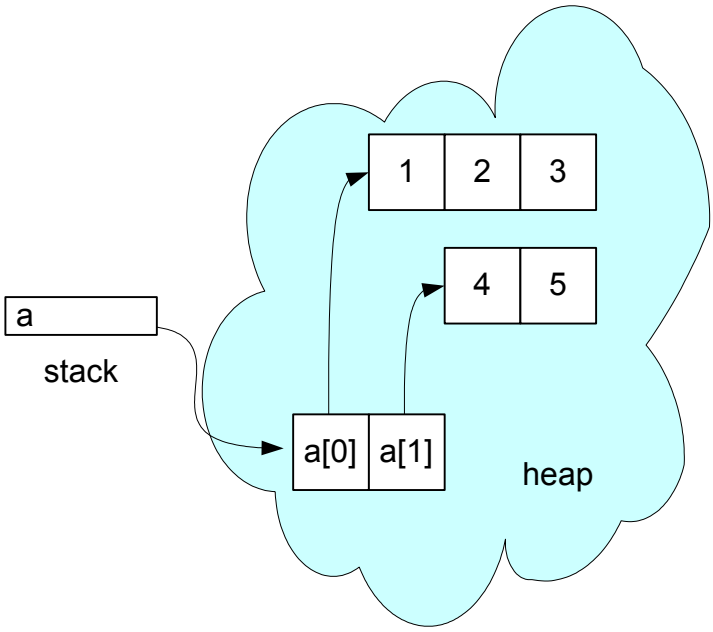
- `a` jest tablicą referencji typu `int[]`. Rozmiar tablicy (`a.length`) wynosi 2.
- `a[0]` jest referencją 3-elementowej tablicy typu `int`
- `a[1]` jest referencją 2-elementowej tablicy typu `int`
- utworzenie obiektów i nadanie wartości początkowych dokonywane jest automatycznie na podstawie listy inicjalizacyjnej.

Kod inicjalizacji może zostać przepisany jako:

```
int[][] a = new int[2][];
a[0]=new int[]{1,2,3};
a[1]=new int[]{4,5};
```

lub

```
int[][] a = new int[2][];
a[0]=new int[3];
a[0][0]=1; a[0][1]=2; a[0][2]=3;
a[1]=new int[2];
a[1][0]=4; a[1][1]=5;
```



W analogiczny sposób tworzone są wielowymiarowe tablice obiektów.

Przykład:

```
Integer[][] a ={
    {new Integer(1), new Integer(2), new Integer(3)},
    { new Integer(4), new Integer(5)}
};
```

Zmiana rozmiarów tablicy

Tablice mają stały rozmiar ustalany w momencie tworzenia obiektu. Nie jest możliwe poszerzenie lub zmniejszenie tablicy.

Jednakże możliwe jest utworzenie nowej tablicy, skopiowanie zawartości i przypisanie referencji.

Przykład

```
int[] a = {1,2,3};
int[]tmp = new int[5];
System.arraycopy(a,0,tmp,0,a.length);
tmp[3]=4;
tmp[4]=5;
a=tmp;
for(int i=0;i<a.length;i++)
   System.out.print(a[i]+" ");
System.out.println();
```

Rezultat

```
1 2 3 4 5
```

Powyższy przykład wykorzystuje statyczną metodę `arraycopy` klasy `System` realizująca kopiowanie ciągu elementów tablicy.

```
public static native void arraycopy(
   Object src,
   int src_position,
   Object dst,
   int dst_position,
   int length )
```

Metoda ta jest zaimplementowana jako `native` – czyli zrealizowana bezpośrednio w kodzie maszynowym platformy sprzętowej, na której uruchamiana jest maszyna wirtualna Java.

Kontenery: podstawowe informacje

Obecnie (2017) tablice najczęściej są stosowane dla **typów wbudowanych**. Mogą one na przykład być częścią interfejsu bibliotek, które używają wewnętrznie kodu napisanego w C/C++ lub wykonywanego na GPU (jak OpenCL lub JCuda).

Tablice obiektów są rzadko stosowane, ponieważ zostały całkowicie wyparte przez kontenery mające postać typów generycznych.

Składnia typów generycznych przypomina szablony C++:

- `List<Double>`
- `ArrayList<String>`
- `TreeSet<Integer>`
- `Map<String, Person>`.

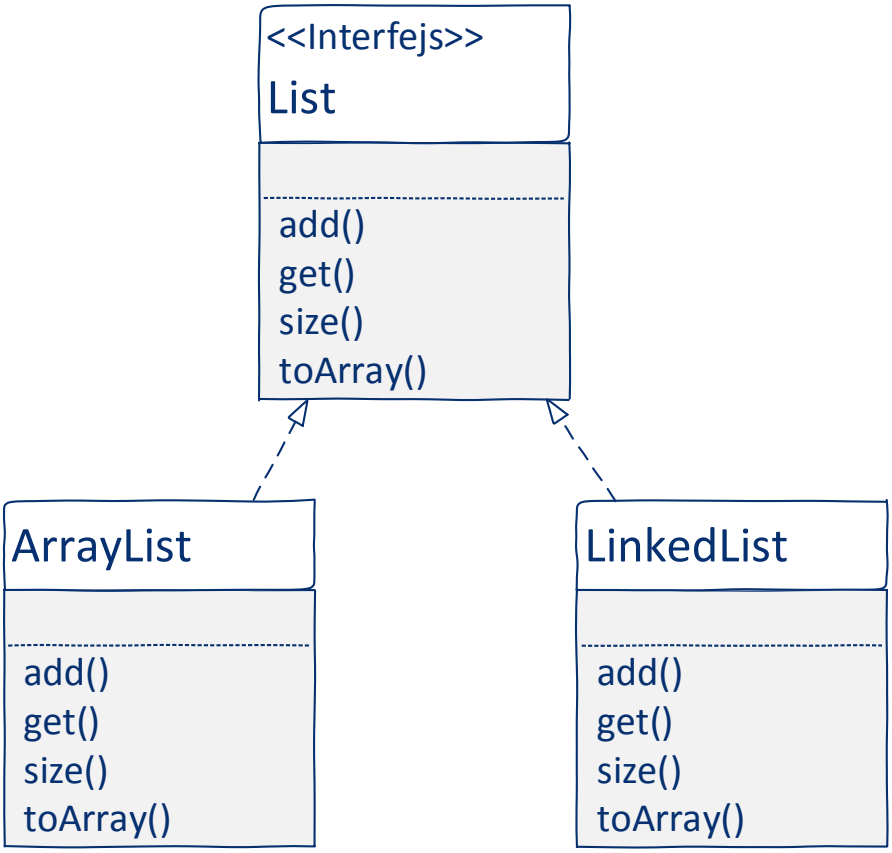
Kontenery zdefiniowane jako typy generyczne są kontenerami obiektowymi, czyli przechowują referencje do obiektów, dla których pamięć jest przydzielana na stercie.

Nie jest możliwe posługiwanie się kontenerami parametryzowanymi typem wbudowanym, jak `ArrayList<int>` czy `ArrayList<double>`, ale można stosować `ArrayList<Integer>` lub `ArrayList<Double>`.

Listy

Najbliższe tablicom są listy. W API zdefiniowany jest interfejs `List<E>` oraz kilka klas realizujących ten interfejs. Najczęściej używane klasy to:

- `ArrayList<E>` -- wektor
- `LinkedList<E>` -- lista



Podstawowe metody:

- `add(E e)` – dodaje element e na końcu listy
- `E get(int i)` – zwraca i-ty element (niezalecane dla `LinkedList<E>`, lepiej użyć iteratora)
- `int size()` – zwraca liczbę elementów
- `Iterator<E> iterator()` – zwraca iterator

Dla klas będących opakowaniami (ang. *wrapper*) typów wbudowanych konwersje: `int` ↔ `Integer`, `double` ↔ `Double` są wykonywane automatycznie (ang. *unboxing*).

Przykład:

```
ArrayList<Double> l = new ArrayList<>();
for(double x=0;x<=Math.PI;x+=Math.PI/10)l.add(x);

for(double x:l)
     System.out.println(x);
for(int i=0;i<l.size();i++)
     System.out.println(l.get(i));
```

Dostęp do elementów zapewniony jest także przez iteratory:

```
Iterator<Double> it = l.iterator();
while(it.hasNext()){
    Double v = it.next();
    System.out.println(v);
}
```

```
for(Iterator<Double> it2 = l.iterator();
it2.hasNext();) {
    double v = it2.next();
    System.out.println(v);
}
```

Tablice dwuwymiarowe w postaci list

Odpowiednikiem tablic dwuwmiarowych jest lista list, czyli na przykład `ArrayList<ArrayList<Integer>>`.

```
ArrayList<ArrayList<Integer>> l2d =
new ArrayList<ArrayList<Integer>>();
//List<List<Integer>> l2d =
new ArrayList<List<Integer>>();
l2d.add(new ArrayList<Integer>());
l2d.add(new ArrayList<Integer>());
l2d.get(0).add(1);
l2d.get(0).add(2);
l2d.get(1).add(3);
l2d.get(1).add(4);
l2d.get(1).add(5);

for(int i=0;i<l2d.size();i++){
    for(int j=0;j<l2d.get(i).size();j++) {
        System.out.print(l2d.get(i).get(j) + " ");
    }
    System.out.println();
}
```

Wynik:

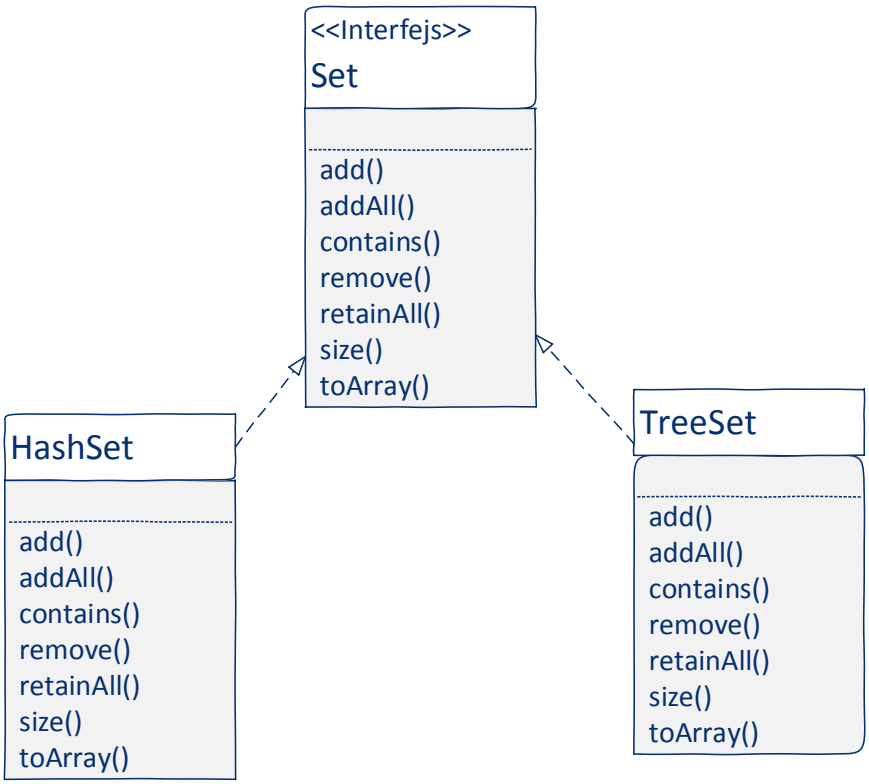
```
1 2
3 4 5
```

- Wyrażenie `l2d.get(i).get(j)` to odpowiednik `l2d[i][j]`.
- Doświadczony programista raczej zadeklaruje l2d jako: `List<List<Integer>> l2d = new ArrayList<List<Integer>>();`

Zbiory

Podobnie jak dla list (spośród których `ArrayList` nie jest w rzeczywistości listą) w API zdefiniowano kontenery obiektowe realizujące funkcjonalność zbiorów:

- Przechowują referencje do obiektów
- Zapewniają unikalność elementów na podstawie:
  - `HashSet<E>`: równości elementów -- `equals()`
  - `TreeSet<E>`: porównania wartości – `compare()` lub `compareTo()`



Podstawowe metody:

- `add(E e)` – dodaje element
- `addAll(Collection<? extends E> c)` – dodaje wszystkie element z kolekcji `c`
- `contains(Object o)` – sprawdza, czy o należy do zbioru
- `remove(Object o)` – usuwa `o` ze zbioru
- `retainAll(Collection<?> c)` – usuwa ze zbioru wszystkie element, które nie należą do `c`

Przykład

```
Set<String> words = new HashSet<>();
// Set<String> words = new TreeSet<>();
 for(String s :
"wyklęty powstań ludu ziemi ziemi ludu powstań
wyklęty".split(" ")){
    words.add(s);
}
System.out.print(words.contains("wyklęty")+ " ");
for(String s: words)System.out.print(s+ " ");
```

Wynik:

```
HashSet: true wyklęty ludu powstań ziemi
TreeSet: true ludu powstań wyklęty ziemi
```

Różnice:

- `HashSet` przechowuje referencje w pojemnikach (ang. *buckets*) i wybiera pojemnik na podstawie wartości funkcji mieszającej `hashCode()`, następnie przeszukuje pojemnik.
- `TreeSet` przechowuje referencje w drzewie i wymaga, aby elementy były **porównywalne**
- Iteracja po `TreeSet` zwraca elementy posortowane
- Prawdopodobnie w większości przypadków `HashSet` działa szybciej

Przykład – pomiar czasu wykonania

Generujemy losowo dwie kolekcje elementów

```
List<Double> inCollection = new ArrayList<>();
List<Double> outOfCollection = new ArrayList<>();

Random rand = new java.util.Random(123);
for(int i=0;i<1000;i++){
    inCollection.add(rand.nextDouble());
}

for(int i=0;i<1000;i++){
    outOfCollection.add(rand.nextDouble());
}
```

Wyznaczamy czasy:

- Dodawania elementów z `inCollection` do zbioru
- Wyszukiwania elementów z `inCollection`
- Wyszukiwania nieobecnych elementów z `outOfCollection`

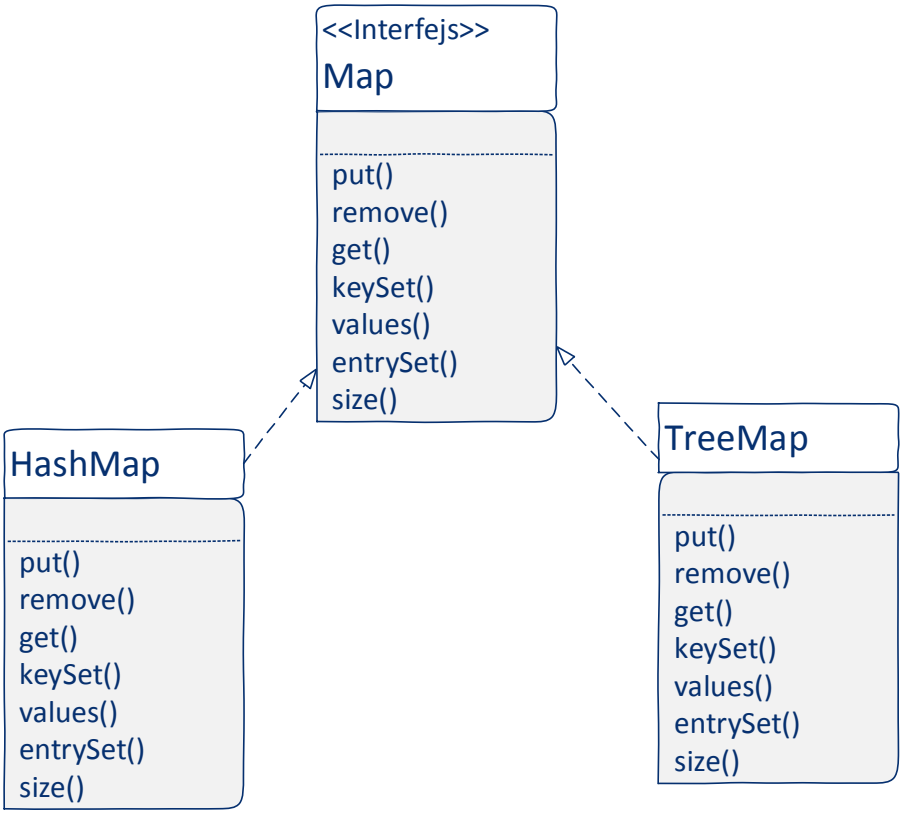
```
Set<Double> set = new HashSet<>();
// Set<Double> set = new TreeSet<>();
double t1 = System.nanoTime()/1e6;
for(Double d:inCollection){
    set.add(d);
}
double t2 = System.nanoTime()/1e6;
for(Double d:inCollection){
    boolean isinSet = set.contains(d);
}
double t3 = System.nanoTime()/1e6;
for(Double d:outOfCollection){
    boolean isinSet = set.contains(d);
}
double t4 = System.nanoTime()/1e6;
System.out.printf(Locale.US,
        "Dodawanie: %f ms\n" +
        "Wyszukiwanie obecnych: %f ms\n" +
        "Wyszukiwanie nieobecnych %f ms\n",
        t2-t1,t3-t2,t4-t3);
```

Przykładowe wyniki

	HashSet	TreeSet
Dodawanie	3.508065 ms	4.149499 ms
Wyszukiwanie obecnych	0.849493 ms	3.475877 ms
Wyszukiwanie nieobecnych	0.515947 ms	0.729136 ms

Mapy

Mapy przechowują odwzorowanie kluczy w wartości. Znając klucz – możemy szybko znaleźć wartość. Klucze przechowywanych w mapie są unikalne.



Główne implementacje:

- `HashMap<K,V>` – klucze są przechowywane i wyszukiwane na podstawie funkcji mieszającej
- `TreeMap<K,V>` – klucze są przechowywane w drzewie, muszą być porównywalne

Podstawowe metody:

- `put(K key, V value)` -- dodaje parę `(key,value)` do mapy
- `remove(Object key)` – usuwa parę odpowiadającą kluczowi `key`
- `get(Object key)` – zwraca wartość `V` odpowiadającą kluczowi `key` lub `null`, jeżeli klucza nie znaleziono
- `keySet()` – zwraca zbiór kluczy (klucze są unikalne)
- `values()` – zwraca `Collection<V>` – kolekcję wartości
- `entrySet()` – zwraca zbiór par (klucz, wartość), typu `Set<Map.Entry<K,V>>`

Przykład 1

```
Map<Double,Double> sin = new TreeMap<>();
for(double a=0;a<Math.PI+1e-5;a+=Math.PI/16){
    sin.put(a,Math.sin(a));
}
System.out.println(sin.get(Math.PI/4));

for(Map.Entry<Double,Double> e: sin.entrySet()){
    System.out.printf(Locale.US,"sin(%f)=%f\n",
        e.getKey(),e.getValue());
}
}
```

Przykład 2

Chcemy przechowywać w kontenerze obiekty klasy `Person`

```
class Person {
    String pesel;
    String imię;
    String nazwisko;
    Person(String pesel, String imię, String nazwisko) {
        this.pesel = pesel;
        this.imię = imię;
        this.nazwisko = nazwisko;
    }
}
```

Kontener możemy zdefiniować w postaci listy:

```
List<Person> plist = new ArrayList<>();
plist.add(new Person("2001010112345","Jan","Kowalski"));
// ... i 100000 podobnych obiektów
```

Następnie chcemy odnaleźć jeden z obiektów

```
static Person find(String pesel, List<Person> personList){
    for(Person p:personList){
        if(p.pesel.equals(pesel))return p;
    }
    return null;
}
```

Wywołanie

```
Person p = find("2001010112345");
```

ma liniową złożoność obliczeniową O(n), gdzie n jest liczbą elementów listy.

Jeżeli zastosujemy mapę

```
Map<String,Person> pesel2person =
    new TreeMap<String, Person>();
pesel2person.put("2001010112345",
    new Person("2001010112345","Jan","Kowalski"));
// ... i 100000 podobnych obiektów
```

Wywołanie:

```
Person p = pesel2person.get("2001010112345");
```

będzie miało złożoność nie gorszą niż O(log(n)) – dla 100000 będzie to 12 porównań (8685 razy szybciej).