JAVA

M. Albrycht P. Laskowski K. Stefański Ł. Zubkowicz

15 grudnia 2008

Green Project

- Początkowo mały, zamknięty projekt, zapoczątkowany w 1991 roku przez Patricka Naughtona, Mike'a Sheridana i Jamesa Goslinga z firmy Sun.
- Celem nie było zaprojektowanie języka programowania.
- Tajny "Green Team" pracował przez 18 miesięcy.
- Efektem pracy był prototyp urządzenia do sterowania różnego rodzaju sprzętem w domu, nazwany *7.

StarSeven

Oak

- Możliwość współpracy *7 z szeroką gamą urządzeń została osiągnięta, dzięki zastosowaniu nowego języka programowania, niezależnego od maszyny.
- Autor języka, James Gosling, nazwał go Oak (inspiracją był dąb znajdujący się za oknem).
- Grupa szukała popytu na urządzenie u telewizji kablowych.
- Rynek nie wykazał jednak zainteresowania.

A może Internet?

- Internet był dziedziną, która w tamtym czasie stawała się powoli popularna (1993 rok).
- Różnego rodzaju urządzenia łączyły się z Internetem.
- Przez przypadek język, nazwany później Javą (bo nazwy Oak nie można było zastrzec), doskonale realizował potrzeby związane z pisaniem aplikacji internetowych.

Sukces

- W 1994 roku powstała przeglądarka internetowa WebRunner (nazwana później HotJava™) wykorzystująca technologię Javy, pozwalająca wyświetlać ruchome obrazy, grafikę 3D i inne niespotykane dotąd rzeczy
- Podczas konferencji ludzie, przyzwyczajeni do statycznego wyglądu stron WWW, byli zachwyceni nowymi możliwościami
- W 1995 roku kod źródłowy Javy został udostępniony w Internecie
- Po kilku miesiącach liczba pobrań wynosiła kilka tysięcy

Write Once, Run Anywhere

- Java 1.0 była dostępna na popularne platformy
- Przeglądarki internetowe zaczęły obsługiwać aplety Javy
- Od wersji 2 (zwanej początkowo 1.2 – 1998 rok) pojawiły się nowe konfiguracje Javy na różne rodzaje platform – J2SE, J2EE, J2ME
- Aktualna wersja 6 została wydana w 2006 roku. Zrezygnowano z nazw J2SE, J2EE, J2ME na rzecz Java SE, Java EE, Java ME

Prosta, obiektowa i znajoma

- Przeciętny programista powinien sobie z nią poradzić bez specjalnego przeszkolenia
- Zapewnia mechanizmy obiektowości
- Składnia języka powinna być znajoma i intuicyjna – w tym celu większość składni została oparta na języku C++, usuwając z niego złożone i problemotwórcze elementy

Solidna i bezpieczna

- Dokładna kontrola typów, przechwytywania wyjątków itp.
- Mechanizm zarządzania pamięcią
  - operator *new*
  - brak operatora *delete*
  - brak wskaźników
  - brak wycieków pamięci
- Mechanizmy bezpieczeństwa uniemożliwiające modyfikację kodu programu z zewnątrz

Przenośna i niezależna od architektury

- Kompilowanie programu do bytecode'u
- Wirtualna maszyna uruchamiana na danym komputerze, wykonywująca tak skompilowany kod

Wydajna

- Mechanizm zarządzania pamięcią przydziela na żądanie za każdym razem kolejną porcję pamięci, więc działa to szybko
- Odśmiecanie jest uruchamiane, gdy brakuje pamięci. Wtedy program jest zatrzymywany, a aktywne obiekty są kopiowane na nową stertę.
- Gdy program nie pobiera już dużej ilości pamięci, odśmiecanie przechodzi w tryb niskopriorytetowego wątku, który usuwa pojedyncze nieużywane obiekty i "żongluje" pozostałymi, aby zajmowały w miarę spójny obszar

Interpretowana, wielowątkowa i dynamiczna

- Bytecode jest prosty do interpretacji, po prostu wykonywane są kolejne instrukcje. Nie jest wymagana żadna analiza leksykalna
- Mechanizmy wielowątkowości pozwalają poprawić responsywność programów, a mechanizmy synchronizacji (monitory, muteksy) zapewnić ich poprawność
- Klasy są ładowane dynamicznie, dopiero wtedy, gdy zajdzie potrzeba ich użycia

Java SE (Standard Edition)

- Podstawowa wersja
- Przeznaczona do ogólnych zastosowań
- Zarówno dla serwerów jak i komputerów stacjonarnych

Java EE (Enterprise Edition)

- Wspiera architekturę wielowarstwową
- Komponenty umieszczane na serwerze aplikacji (np. JBoss, Apache Tomcat)

Java ME (Micro Edition)

- Wersja uproszczona
- Przeznaczona dla urządzeń o mniejszych zasobach niż komputery klasy PC, np. telefonów komórkowych, palmtopów
- Okrojony zbiór klas
- Różne podkonfiguracje
  - CLDC (Connected Limited Device Information) – najczęściej w telefonach komórkowych, znacznie ograniczone środowisko
  - CDC (Connected Device Information) – stosowane w urządzeniach o większej mocy obliczeniowej, zawiera prawie wszystkie klasy z Java SE, nie licząc tych zajmujących się obsługą GUI

Wszystko jest obiektem

- Java to język obiektowy posiadający hierarchię klas z korzeniem
- Klasą bazową, z której dziedziczą wszystkie inne (pośrednio lub bezpośrednio) jest klasa Object
- Kontrola typów jest statyczna - na etapie kompilacji musimy wiedzieć, jakiego typu są dane zmienne

Deklarowanie nowych zmiennych

- Deklaracja zmiennej wygląda, tak jak w języku C++:

```
Pizza hawajska;
```

- Jednak w przeciwieństwie do C++, taki zapis nie tworzy nowego obiektu typu Pizza.

Dostęp do obiektów

- Dostęp do obiektów jest realizowany przez referencje.
- Zadeklarowane zmienne są więc jedynie referencjami, które mogą wskazywać na jakiś obiekt.
- Przykład z poprzedniego slajdu tworzy jedynie referencję do jakiegokolwiek obiektu typu Pizza. Jest inicjowany na specjalną referencję *null*, która nie wskazuje na żaden obiekt
- Jeśli chcemy utworzyć obiekt danego typu, musimy jawnie wywołać konstruktor jego klasy, używając operatora **new**:

```
Pizza hawajska = new Pizza();
```

Przypisywanie

```
Pizza hawajska = new Pizza();
Pizza pepperoni = hawajska;
```

- W języku C++ taka konstrukcja spowodowałaby skopiowanie pizzy hawajskiej i przypisanie kopii na pizzę pepperoni.
- Skoro jednak w Javie zmienne są referencjami, to ta konstrukcja skopiuje referencję do pizzy hawajskiej.
- W efekcie mamy dwie referencje, które wskazują na jedną i tę samą pizzę hawajską.

Kontakt z obiektami

- Do porozumiewania się z obiektem służy operator kropki (.)
- Operator kropki użyty na referencji, odwołuje się do obiektu, na który dana referencja wskazuje.
- Za jego pomocą możemy wywoływać metody obiektu.

```
Pizza pepperoni = new Pizza();
pepperoni.zjedz();
```

Przekazywanie parametrów

- Parametry funkcji przekazywane są przez **wartość**.
- Skoro zmienne są referencjami, jest to wartość referencji!
- Wewnątrz funkcji operujemy więc na obiekcie, którego referencję przekazaliśmy, a nie na jego kopii:

```
...
public Boolean jedz(Jedzenie posilek) {
    this.glodny -= posilek.zjedz();
    return this.glodny <= 0;
}
...
```

Czy na pewno wszystko jest obiektem?

- No... prawie wszystko.
- Pewna grupa typów jest traktowana w sposób szczególny
- Utworzenie obiektu za pomocą operatora **new** jest kosztowne – wymaga przydzielenia pamięci na stercie, a następnie uwzględnianie go przez odśmiecarkę przy oczyszczaniu pamięci.
- Z pomocą przychodzą tzw. typy podstawowe, których zmienne nie są referencjami, a stanowią swoją wartość
- Oznacza to, że interesująca nas wartość zmiennej jest trzymana na stosie

Typy podstawowe

- Typy podstawowe to boolean, char, byte, short, int, long, float, double i void.
- Typy podstawowe **nie** są podklasami klasy Object.
- Ich zmienne tworzymy nie używając operatora **new**:

```
int zmienna_int;
boolean zmienna_boolean = true;
```

- Istotną rzeczą jest fakt, że w przeciwieństwie do C++, typy podstawowe mają zawsze ten sam, ustalony rozmiar, niezależnie od architektury maszyny, na jakiej uruchamiamy program.

A przekazywanie zmiennych typów podstawowych?

- Tak samo jak w przypadku referencji, przekazywanie następuje przez wartość.
- Jednak w tym przypadku przekazywana przez wartość zmienna nie jest referencją! To jej wartość nas interesuje!
- Wniosek: Przekazując zmienną typu podstawowego jako parametr, nie możemy wartości przez nią niesionej zmodyfikować tak, by było to widziane na zewnątrz funkcji

```
void daj_kase(int i) {
    i.ustaw(0); // ?!?! i nie jest referencja!
}
```

Opakowywanie typów podstawowych

- Czasem chcielibyśmy operować na prostych danych jak liczby typu integer, jednak korzystać z nich jak z obiektów
- Java udostępnia system opakowywania typów podstawowych w obiekty
- Dla każdego typu podstawowego mamy klasę opakowującą, np.:

```
int zm_podst = 3;
Integer zm_ob = new Integer(zm_podst);
```

Komentarze

- Komentarz w Javie robi się tak samo jak w C++

```
// To jest komentarz jednolinijkowy

/*
    To jest komentarz
    wielolinijkowy
*/
```

Operatory

- Operatory w Javie w zasadzie nie różnią się od tych z C++
- W większości operują na typach podstawowych
- Wyjątkami są = (przypisanie), == (identyczność) i != (sprawdzenie, czy zmienne są różne), które operują również na obiektach (czyt. ich referencjach)
- Trzeba pamiętać, że w przypadku obiektów operujemy na referencjach, a w przypadku typów podstawowych - na wartościach. Jeśli chcemy porównywać obiekty inaczej niż po referencjach, musimy zdefiniować dla nich metodę *equals(Object obj)* i porównywać za jej pomocą.

Przeciążanie

- Operatorów nie można przeciążać. Dla obiektów definiujemy odpowiednie metody, które mają spełniać rolę operatorów np.

```
Liczba.mniejsza(inna_liczba);
```

- Czy też wcześniej wspomniane *equals*
- Wyjątek stanowi klasa String, reprezentująca napis, dla której mamy przedefiniowane operatory + i += (konkatenacja)

Operatory arytmetyczne

- Operatory arytmetyczne

```
c = a + b // dodawanie liczb
d = a - b // odejmowanie liczb
e = a * b // mnozenie liczb
f = a / b // dzielenie liczb
g = a % b // dzielenie modulo
h = -a    // zmiana znaku
i = +a    // w sumie nic nie robi, dualny do
          // jednoargumentowego (-)
```

Skrócone operatory arytmetyczne

- Skrócone operatory arytmetyczne

```
a *= b // dodawanie liczb
a += b // odejmowanie liczb
a -= b // mnozenie liczb
a /= b // dzielenie liczb
a %= b // dzielenie modulo
```

Operatory inkrementacji i dekrementacji

- Operatory inkrementacji i dekrementacji

```
i++ // zwieksz o 1 i zwroc stara wartosc
++i // zwieksz o 1 i zwroc nowa wartosc
i-- // zmniejsz o 1 i zwroc stara wartosc
--i // zmniejsz o 1 i zwroc nowa wartosc
```

Operatory relacji

- Operatory relacji

```
a >= b   // tylko liczby
a > b    // tylko liczby
a <= b   // tylko liczby
a < b    // tylko liczby
a == b   // wszystkie typy
a != b   // wszystkie typy
```

Operatory logiczne

- Operatory logiczne

```
a && b   // zachodzi a i zachodzi b
a || b   // zachodzi a lub zachodzi b
!a       // nie zachodzi a
```

- Operatory logiczne operują na typie podstawowym boolean i zwracają wartość typu boolean
- Stosowane jest leniwe wyliczanie

Operatory bitowe

- Operatory bitowe

```
c = a & b    // koniunkcja bitowa (and)
d = a | b    // alternatywa bitowa (or)
e = a ^ b    // alternatywa wykluczajaca (xor)
f = ~a       // negacja bitowa (not)
```

- Operatory bitowe operują na zmiennych typów podstawowych, reprezentujących liczby całkowite

Skrócone operatory bitowe

- Skrócone operatory bitowe

```
a &= b // koniunkcja bitowa (and)
a |= b // alternatywa bitowa (or)
a ^= b // alternatywa wykluczajaca (xor)
```

Przesunięcia bitowe

- Przesunięcia bitowe

```
d = a << b   // przesuniecie bitowe w lewo
c = a >> b   // przesuniecie bitowe w prawo
             // z powieleniem pierwszego bitu
e = a >>> b  // przesuniecie bitowe w prawo
             // z uzupelnieniem zerami
```

Skrócone przesunięcia bitowe

- Przesunięcia bitowe

```
a <<= b      // przesuniecie bitowe w lewo
a >>= b      // przesuniecie bitowe w prawo
             // z powieleniem pierwszego bitu
a >>>= b     // przesuniecie bitowe w prawo
             // z uzupelnieniem zerami
```

Operator trójargumentowy if-else

- Operator trójargumentowy if-else

```
wyrazenie-logiczne ? wartosc1 : wartosc2
```

- Jeśli wyrażenie-logiczne zwraca *true*, to wartością całego wyrażenia jest wartość1, w przeciwnym przypadku wartość2

Skoki warunkowe

```
if (wyrazenie_logiczne)
    instrukcja1
else
    instrukcja2
```

- przykład

```
if (a >= 0)
    b += a;
else
    a--;
```

Skoki warunkowe

- Klauzula **else** może być pominięta

```
if (wyrazenie_logiczne)
    instrukcja
```

- Przykład:

```
if (ja.glodny())
    ja.jedz(kanapka);
```

Uwaga!

- W przeciwieństwie do C++, w Javie predykat MUSI być wyrażeniem logicznym!

```
if (1) // zle! 1 nie jest wyrazeniem logicznym!
  ja.jedz(kanapka);
```

- Ma to na celu zapobiegnięcie jakże częstym pomyłkom zdarzającym się programistom C++:

```
if (zmienna1 = zmienna2)
  ja.jedz(kanapka);
```

Switch

- Instrukcja switch działa tak samo jak w C++

```
switch (wyrazenie) {
    case wartosc-calkowita-1: instrukcja1
    case wartosc-calkowita-2: instrukcja2
    ...
    default: instrukcja
}
```

- Wyrażenie powinno zwracać wartość całkowitą

Pętle

- Pętle również działają tak samo jak w C++
- Pętla while

```
while (wyrazenie-logiczne)
    instrukcja
```

- Pętla do-while

```
do
    instrukcja
while (wyrazenie-logiczne)
```

- Pętla for

```
for (inicjalizacja; wyrazenie-logiczne; krok)
    instrukcja
```

Nowość - foreach

- W Javie wprowadzono nowość w stosunku do C++ - pętlę foreach
- Służy ona do przeiterowania się po wszystkich elementach tablicy bądź kolekcji (tablice i kolekcje będą omówione później, na razie odwołuję się do intuicji)

```
for (zmienna : kolekcja)
    instrukcja
```

- Przykład:

```
for (Jedzenie posilek : lodowka) {
    ja.jedz(posilek);
}
```

final - zamiast const

- Aby zadeklarować stałą, używamy słowa kluczowego *final*
- Wartość stałej nie musi być znana w czasie kompilacji
- Możemy tworzyć puste stałe, które mogą być przypisywane tylko raz

```
private final int stala_czasu_wykonania = (new Random()).nextInt(20);
private final int stala_czasu_kompilacji = 15;
private final boolean stala_pusta;

public void test() {

    stala_czasu_kompilacji = 10;    // zle! nie mozemy modyfikowac stalej!

    stala_pusta = true;             // ok, robimy pierwsze i jedyne przypisanie
    stala_pusta = false;            // zle! kolejne modyfikacje sa niedozwolone!
}
```

final - a co z obiektami?

- Obiekty są reprezentowane za pomocą referencji
- Deklarując zmienną reprezentującą obiekt jako finalną zaznaczamy, że **referencja** ma być stałą
- Nie ma mechanizmu, który zapewniłby niezmienność obiektu!

```
private final Pizza hawajska = new Pizza();
private Pizza pepperoni = new Pizza();
private Sos pomidorowy = new Sos();

public void costam() {
    hawajska = pepperoni; // zle! referencja zadeklarowana jako finalna
                          // nie moze zmienic obiektu, na ktory wskazuje!
    hawajska.polejSosem(pomidorowy); // ok, obiekt mozemy modyfikowac
}
```

Inne zastosowania słowa kluczowego final

- Słowo kluczowe final ma różne znaczenia, zależnie od kontekstu
- Gdy metoda jest zadeklarowana jako finalna, oznacza to, że nie może być przesłonięta w podklasach

```
class A {
    public final int metoda_finalna() { return 0; }
}

class B extends A {
    public final void metoda_finalna() { return 1; } // zle!
}
```

- Zadeklarowanie klasy jako finalnej zabrania dziedziczenia z niej

```
final class A { }
class B extends A { } // zle! A jest finalna!
```

Modyfikatory dostępu do klas

- Szczegóły deklaracji klas, metod itp. będą omówione za chwilę
- Teraz skupimy się na modyfikatorach
- Każda klasa, metoda bądź atrybut ma przypisany modyfikator dostępu. Jeśli go nie podamy, przyjmowany jest domyślny.
- Wyróżniamy 2 modyfikatory dostępu dla klas
  1. Pakietowy (domyślny) - dostęp do danej klasy jest jedynie wewnątrz pakietu
  2. Publiczny - dostęp do danej klasy jest z każdego miejsca

```
class DostepnaWPakiecie {
}
public class DostepnaWszedzie {
}
```

Modyfikatory dostępu do metod

- Wyróżniamy 4 modyfikatory dostępu dla metod
  1. Prywatny - dostęp do danej metody ma jedynie klasa, do której metoda należy
  2. Pakietowy (domyślny) - dostęp do danej metody jest jedynie wewnątrz pakietu
  3. Chroniony - dostęp do danej metody ma klasa, do której należy metoda, jej podklasy oraz wszystkie klasy znajdujące się w tym samym pakiecie co klasa zawierająca tę metodę
  4. Publiczny - dostęp do danej metody jest z każdej klasy

Modyfikatory dostępu do metod

```
package pakiet_a

public class A {
    private void metoda_prywatna() {}
    void metoda_pakietowa() {}
    protected void metoda_chroniona() {}
    public void metoda_publiczna() {}
}
class WTymSamymPakiecie {
    A a = new A();
    public void test() {
        a.metoda_prywatna(); // zle!
        a.metoda_pakietowa() // ok
        a.metoda_chroniona() // ok
        a.metoda_publiczna() // ok
    }
}
```

Modyfikatory dostępu do metod

```
package pakiet_b
class Podklasa extends A {
    A a = new A();
    public void test() {
        this.metoda_prywatna(); // zle!
        a.metoda_prywatna();    // zle!
        this.metoda_pakietowa();// zle!
        a.metoda_pakietowa();   // zle!
        this.metoda_chroniona();// ok
        a.metoda_chroniona();   // ok
        this.metoda_publiczna();// ok
        a.metoda_publiczna();   // ok
    }
}
class ObcaWInnymPakiecie {
    A a = new A();
    public void test () {
        a.metoda_prywatna(); // zle!
        a.metoda_pakietowa() // zle!
        a.metoda_chroniona() // zle!
        a.metoda_publiczna() // ok
    }
}
```

Modyfikatory dostępu do atrybutów

- W przypadku atrybutów, sprawa wygląda identycznie jak w przypadku metod
- Mamy 4 takie same modyfikatory i ich działanie jest jednakowe

Omówienie przez przykład

```
class Komputer {
  private Procesor procesor;
  Komputer() {
    procesor = new Pentium(1000);
  }
  int wlacz() { ... }
  private void wczytajBIOS() { ... }
}
```

- Wygląda znajomo.
- Brak średnika.
- Tylko definicje.
- Brak zbiorowych modyfikatorów widoczności.

"Ty to masz klasę!"

```
[abstract | final] [public]  class Komputer extends PozeraczPradu
 implements PozeraczCzasu {
  ...
}
```

- domyślna widoczność - w obrębie pakietu
- klasa abstrakcyjna
- klasa finalna

"Nie ma to jak solidna konstrukcja!"

```
class Komputer {
  [private | public | protected] Komputer() {
    this("powerpc");
  }
  [private | public | protected] Komputer(String architektura) {
    this.architektura = architektura;
 }
}
```

- nazewnictwo
- widoczność
- przeciążanie
- nie podlegają dziedziczeniu
- konstruktor pusty

"Nie ma to jak solidna konstrukcja!" c.d.

```
class Komputer extends PozeraczPradu {
  Komputer() {
    this("powerpc");
    wlacz();
  }
  Komputer(String architektura) {
    super(220);
  }
}
```

- wywołanie innego konstruktora (this, super)
- konstruktor nadklasy
- łańcuch wywołań konstruktorów
- inicjalizacja pól instancyjnych klasy

"Mamy swoje metody."

```
class Komputer {
  [private|protected|public] [static] [abstract | final] bool wlacz()
    throws BrakPradu {
    ...
  }
}
```

- widoczność
- abstract, final
- przeciążanie
- statyczne wiązanie metod statycznych
- poprawność zwracanego wyniku

"Hulali po polach (...)"

```
class Komputer {
  [private|protected|public] [static] [final] Typ nazwaPola;
  Procesor procesor = new C2D("2GHz");
  public Przycisk power = new Przycisk("zielony"), reset, turbo;
}
```

- widoczność
- static, final, transient
- wiele pól tego samego typu
- statyczne wiązanie pól
- inicjalizacja pól statycznych

"A Hasta la vista, baby..."

```
class Komputer {
  public void finalize() {...}
}
```

- brak destruktorów
- metoda finalize przy usuwaniu
- kiedy?
- finalizacja z nadklasy
- ostatnia deska ratunku

"To moje dziedzictwo."

```
class Komputer extends PozeraczPradu {
  ...
}
```

- brak wielodziedziczenia
- przysłanianie pól

Rzutowanie

- w górę

```
Komputer mojPC = new Komputer();
Urzadzenie u = mojPC;
```

  - pola statycznie
  - metody instancyjne dynamicznie

- w dół

```
Komputer domowyPC = (Komputer) u;
```

  - sprawdzane w trakcie kompilacji (ClassCastException)
  - instanceof

Interfejsy

```
[public] interface Transport {
   ...
}
interface TransportLadowy extends Transport {
    static final int MAX_SPEED = 100;
    ...
}
```

- podobieństwo do klasy abstrakcyjnej
- "potrafię" zamiast "jestem"
- nowy typ referencyjny
- extends
- dostęp do pól static final
- widoczność

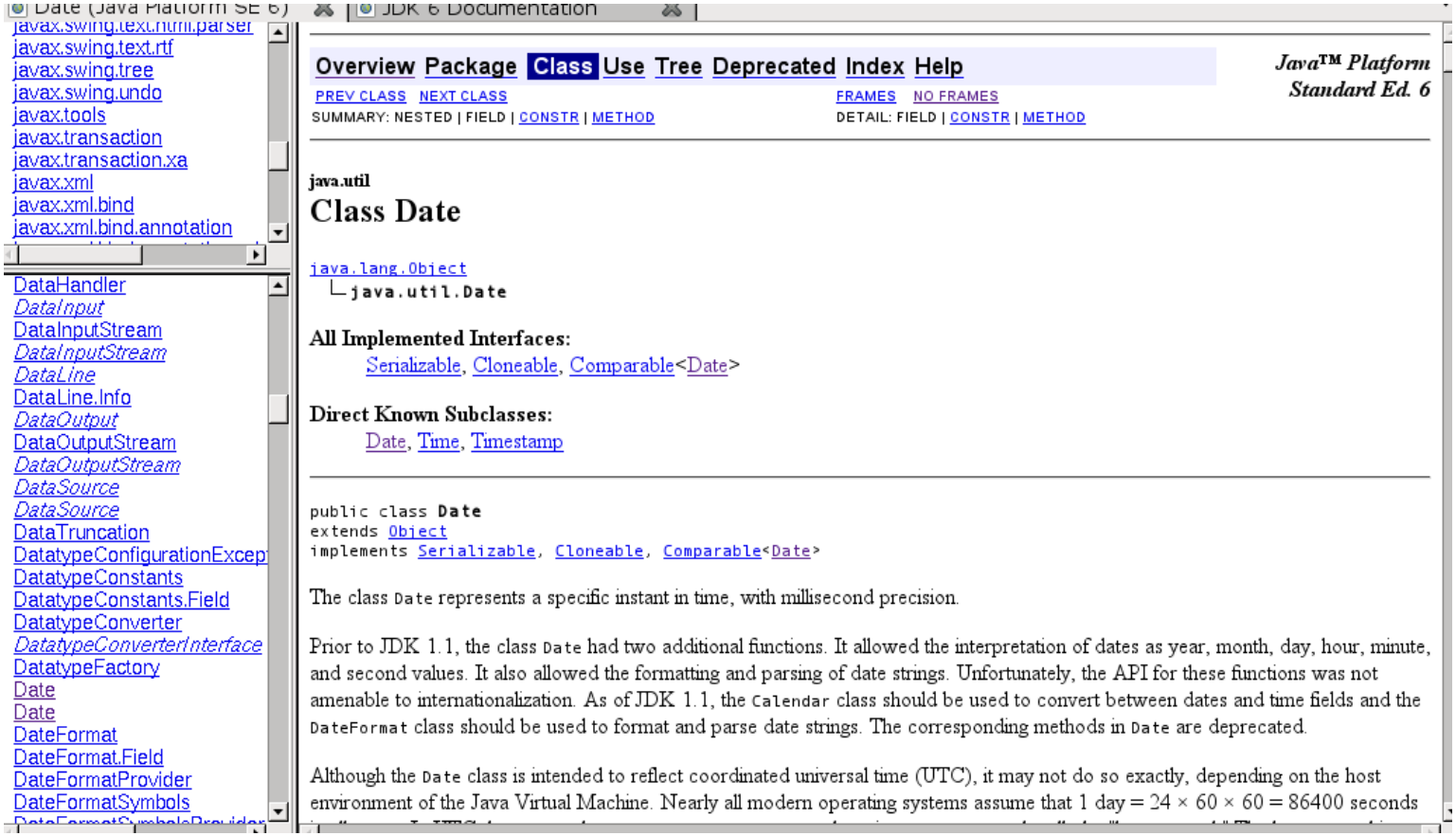
”Moja klasa jest najmojsza.“

- blok inicjalizacji
- statyczny blok inicjalizacji
- klasy wewnętrzne, ANONIMOWE klasy wewnętrzne???
- ...

Idea

- Najlepiej tworzyć dokumentację na bieżąco, kiedy jeszcze wiadomo jaką semantykę ma nasz kod.
- Wygodne jest powiązanie dokumentacji z kodem - nie trzeba wtedy jej szukać, bo jest na miejscu.
- Przy okazji dostajemy generowanie czytelnej i jednolicie wyglądającej dokumentacji wynikowej.

Idea


Date (Java Platform SE 6)
JDK 6 Documentation
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
javax.tools
javax.transaction
javax.transaction.xa
javax.xml
javax.xml.bind
javax.xml.bind.annotation
DataHandler
DataInput
DataInputStream
DataInputStream
DataLine
DataLine.Info
DataOutput
DataOutputStream
DataOutputStream
DataSource
DataSource
DataTruncation
DatatypeConstants
DatatypeConstants.Field
DatatypeConverter
DatatypeConverterInterface
DatatypeFactory
Date
Date
DateFormat
DateFormat.Field
DateFormatProvider
DateFormatSymbols
Overview Package Class Use Tree Deprecated Index Help
Java™ Platform Standard Ed. 6
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
java.util
Class Date
java.lang.Object
java.util.Date
All Implemented Interfaces:
Serializable, Cloneable, Comparable<Date>
Direct Known Subclasses:
Date, Time, Timestamp
public class Date
extends Object
implements Serializable, Cloneable, Comparable<Date>
The class Date represents a specific instant in time, with millisecond precision.
Prior to JDK 1.1, the class Date had two additional functions. It allowed the interpretation of dates as year, month, day, hour, minute, and second values. It also allowed the formatting and parsing of date strings. Unfortunately, the API for these functions was not amenable to internationalization. As of JDK 1.1, the Calendar class should be used to convert between dates and time fields and the DateFormat class should be used to format and parse date strings. The corresponding methods in Date are deprecated.
Although the Date class is intended to reflect coordinated universal time (UTC), it may not do so exactly, depending on the host environment of the Java Virtual Machine. Nearly all modern operating systems assume that 1 day = 24 × 60 × 60 = 86400 seconds

Składnia

- Przez kompilator *Javadoc* jest traktowany po prostu jako komentarz.
- Dozwolone jest używanie *HTML*a, który zostanie włączony bezpośrednio do generowanej dokumentacji.
- Dodatkowo mamy do dyspozycji najróżniejsze rodzaje znaczników dokumentacyjnych.

```
/** Komentarz odnosnie klasy. */
public class DokumentowanaKlasa {

        /** Komentarz odnosnie pola. */
        int dokumentowanePole;

        /** Komentarz odnosnie metody. */
        void dokumentowanaMetoda() {}

}
```

Przykład

```
/**
 * Przyklad pokazujacy zastosowanie Javadoc.
 * Metoda nie robi <strong>absolutnie nic</strong>.
 *
 * @param liczba Liczba calkowita na nic tak naprawde sie nie przydajaca.
 * @param bajt Bajt, z ktorym metoda nic nie robi.
 * @return Referencja do obiektu String, ktorego tak naprawde nie ma.
 *
 * @see Testowa
 */
String method(int liczba, byte bajt) {
        return null;
}
```

Przykład

method

```
java.lang.String method(int liczba,
                        byte bajt)
```

Przyklad pokazujacy zastosowanie Javadoc. Metoda nie robi **absolutnie nic**.

Parameters:

`liczba` - Liczba calkowita na nic tak naprawde sie nie przydajaca.
`bajt` - Bajt, z ktorym metoda nic nie robi.

Returns:

Referencja do obiektu String, ktorego tak naprawde nie ma.

See Also:

`Testowa`

Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Uruchamianie kodu

- Kod wykonywanego programu znajduje się w statycznej metodzie ***main*** klasy publicznej.
- Nazwa pliku ze źródłami musi być taka sama jak nazwa klasy publicznej w nim zawartej (stąd możemy mieć w pliku tylko jedną klasę publiczną).
- Jeśli w pliku są tylko klasy o dostępie pakietowym, to nazwa pliku może być dowolna.

```
// Plik z ponizszym kodem powinien sie nazywac "Program.java"

public class Program {
        public static void main(String[] args) {
                /* ... nasze instrukcje ... */
        }
}
```

Uruchamianie kodu (przykład)

- W wierszu poleceń wpisujemy **javac Program.java**, co wywołuje kompilator *Javy*. Otrzymujemy Plik **Program.class** z bajtkodem.
- Dla każdej klasy niepublicznej również otrzymamy po jednym pliku **.class** (takie same rządzą kompilacją pliku ze źródłami nie zawierającego klasy publicznej).
- Aby uruchomić nasz program wpisujemy **java Program** (uwaga: celowo pominięte zostało rozszerzenie **.class**, chodzi o nazwę klasy ze statyczną metodą ***main***, a nie pliku ją zawierającego).

Modularyzacja kodu

- W *Javie* grupujemy klasy do jednostek zwanych pakietami. Konwencja nakazuje nazywać pakiety małymi literami, a użycie domeny internetowej ma gwarantować unikalność nazw.
- Pakietowi odpowiada odpowiednia struktura w systemie plików (kropki odpowiadają ukośnikom). Wszystkie pliki **.class** wygenerowane z poniższego listingu powinny znaleźć się w katalogu *pl/edu/mimuw/students/lz248258/*.

```
// Wszystkie klasy zaimplementowane w tym pliku
// beda nalezec do ponizszego pakietu:
package pl.edu.mimuw.students.lz248258;

// Aby uzyc klas z innych pakietow wystarczy je zaimportowac:
import java.util.ArrayList;
// Ponizszego nie musimy pisac (importowane automatycznie)
import java.lang.*;

class JakasKlasa {}
```

Pliki **.jar**

- Program w *Javie* fizycznie jest drzewem katalogów z porozrzucanymi tu i ówdzie plikami **.class**. Przy wdrażaniu jest to trochę niewygodne.
- Na szczęście istnieje narzędzie **jar**, które algorytmem *zip* pakuje wszystko do jednego pliku **.jar** (ang. *Java ARchive*). Przy okazji wykonywane są jeszcze inne drobne zabiegi.
- Archiwum to może dodatkowo zawierać plik z manifestem, który np. informuje wirtualną maszynę którą klasę uruchomić (wtedy program uruchamiamy wpisując w wierszu poleceń **java -jar archiwum.jar**).

Zmienna środowiskowa **CLASSPATH**

- Zmienna ta listuje wszystkie (oprócz standardowych) ścieżki do używanych pakietów.
- Jeśli potrzebne pakiety są zarchiwizowane do pliku **.jar**, to dodatkowo wymieniamy nazwę pliku przechowującego archiwum.
- Z pozoru łatwo to wygląda, ale w praktyce często konfiguracja tej zmiennej nastręcza wiele problemów, więc najlepiej zrzucić obowiązek jej ustawienia na jakieś zintegrowane środowisko programistyczne.

Zmienna środowiskowa **CLASSPATH** (przykład)

Załóżmy, że interesujący nas system plików wygląda jak poniżej. Dodatkowo załóżmy także, że program **SpalaczKomputera** importuje pakiety z pliku **biblioteka.jar** i razem z klasą ***Zapalnik*** znajduje się w jednym pakiecie ***pl.cracker***.

```
/home/user/java
 |- biblioteka.jar
 ^- pl
    ^- cracker
       |- Zapalnik.class
       ^- SpalaczKomputera.class
```

Zmienna środowiskowa **CLASSPATH** (przykład)

Mamy w takim wypadku co najmniej dwie możliwości uruchomienia programu:

1. O ile zmienna **CLASSPATH** ma wartość **/home/user/java/archiwum.jar:.** (ważne: zawiera domyślną "kropkę", czyli katalog bieżący), z katalogu */home/user/java* wywołujemy polecenie **java pl.hacker.SpalaczKomputera**.
2. Jeśli do zmiennej **CLASSPATH** dołożymy ścieżkę **/home/user/java**, będziemy mogli poprzedniego polecenia użyć już w każdym katalogu w systemie.

Idea

1. Ten sposób obsługi błędów istniał jeszcze przed programowaniem obiektowym. Jednakże doskonale współgra z obiektowymi językami.
2. Wyjątki to jedyny, a także wymuszany przez kompilator sposób obsługi błędów w *Javie*. Kod który ignoruje wyjątki po prostu się nie skompiluje.
3. Przy pojawieniu się błędu reagujemy na niego, albo jeśli wymaga on wiedzy z szerszego kontekstu, przekazujemy go "wyżej".

Zalety i wady

1. Wyjątki oddzielają kod obsługujący błędy od tego pełniącego jakąś logiczną funkcję. Pozwala to skupić się na głównym przebiegu algorytmu z jednoczesnym braniem pod uwagę sytuacji wyjątkowych (innymi słowy błędów). Język także gwarantuje nam, że każdy błąd znajdzie jakiś kod obsługi (być może pusty, jeśli tak chcemy).
2. Pomimo prostoty składni, wyjątki wymagają od programisty konsekwencji i doświadczenia. W skrajnych sytuacjach ciągle wszystkie są wyrzucane wyżej (wtedy na pewnym poziomie ciężko powiedzieć co konkretnego znaczą), albo wszystkie są zbierane w jeden i nie da się potem powiedzieć dokładnie co było powodem wystąpienia wyjątku (a co za tym idzie, nie wiadomo jak zareagować).

Składnia

1. Nowe wyjątki tworzymy dziedzicząc po jakimś innym wyjątku, możliwie podobnym do naszego. Korzeniem hierarchii wyjątków jest ***Throwable***, ale aby stworzyć swój własny wyjątek najlepiej dziedziczyć nie wyżej niż po ***Exception*** (wyżej są wewnętrzne wyjątki języka).
2. Wyjątek wyrzucamy instrukcją ***throw***.
3. Mamy obowiązek poinformowania jakie wyjątki wyrzuca metoda poprzez słowo kluczowe ***throws***.
4. Jednocześnie musimy zagwarantować, że każdy wyjątek wyrzucany przez metodę której używamy, zostanie wyrzucony wyżej (wspomniane ***throws***) albo obsłużony (blok ***try*** i kod obsługi wyjątku w bloku ***catch***).

Przykład

Wyrzucanie wyjątku:

```
void metodaWyrzucajacaWyjatek(bool b)
throws ExampleExcention1, ExampleException2 {
        if (b)
                throw new ExampleException1();
        else
                throw new ExampleException2("To jest opis bledu.");
}
```

Obsługa wyjątku:

```
try {
        obiekt.czestoNawala(); // wyrzuca wyjatek OftenException
} catch(OftenException e) { // lapiemy wyjatek celem obsluzenia go
        obiekt.zrobTolnaczej();
} catch(Exception e) { // ten blok wylapie wszystko niewylapane wyzej
        System.err.println("Blad krytyczny.")
        System.err.println("Nie spodziewano sie takiego wyjatku.");
        throw e; // albo np. 'throw new CriticalException(e)'
} finally { // blok ten pelni podobna funkcje jak metoda .finalize()
        obiekt.zwolnijZasobyZewnetrzne();
}
```

Asercje

1. Jest to dodatkowy mechanizm sprawdzający pewne niezmienniki (wyrażenia logiczne) w kodzie. Nie jest domyślnie włączony.
2. Gdy asercja zawodzi, wyrzuca wyjątek ***AssertionError***.
3. Często asercje służą jako komentarz na temat własności w danym miejscu kodu.

```
// Ponizsze oznacza tyle, ze sterowanie nigdy nie powinno tutaj dojsc.
// Jesli dojdzie, to wyrzucony zostanie AssertionError.
assert false;
```

Najpierw tablice

1. Mogą przechowywać zarówno obiekty jak i typy podstawowe.
2. Posiadają ustalony przy tworzeniu rozmiar i ten już nigdy nie podlega zmianie.
3. Kontrola zakresu tablicy jest wbudowana w język (w razie błędu wyrzuca wyjątek *ArrayIndexOutOfBoundsException*).
4. Bardzo szybkie, ale mało elastyczne (np. ciężko wstawić element między dwa inne).

```
Double[] tab1 = {3.14159, 12.4};

int[] tab2 = new int[10];
for (int i = 0; i < 10; i++)
        tab2[i] = i + 1;

for (Char e : new char[] {'F', 'a', 'j', 'n', 'i', 'e', '!'})
        System.out.println(e);
```

Rodzaje kontenerów

1. Kontenery mogą przechowywać tylko obiekty. Jako typy uogólnione umożliwiają dospecyfikowanie jakie mają to być obiekty. Zaletą jest za to dostosowywanie rozmiaru do ilości przechowywanych elementów.
2. Mamy tak naprawdę dwa różne interfejsy do kontenerów: ***Collection*** i ***Map*** (stąd dla rozróżnienia nie nazywamy w tym podrozdziale wszystkich razem kolekcjami).
3. Interfejs ***Collection*** posiada trzy rodzaje podinterfejsów: ***List***, ***Set*** i ***Queue***.
4. Dodatkowo interfejs ***Collection*** dziedziczy po interfejsie ***Iterable***, przez co nakazuje implementację metody zwracającej iterator. To dzięki temu można obiektów implementujących ***Collection*** używać w tzw. pętli ***foreach***.

Interfejs *List*

1. Kontenery typu ***List*** zachowują kolejność wstawianych elementów. W dostępie przypominają tablice. Najważniejsze klasy implementujące ten interfejs to ***ArrayList*** i ***LinkedList***.
2. ***ArrayList*** jest szybka jeśli chodzi o losowy dostęp do elementów, ale mało wydajna, gdy chcemy coś wstawić "w środek".
3. ***LinkedList*** oparta jest na implementacji listowej, więc jest idealna, gdy co chwilę coś wyrzucamy i wstawiamy w różnych miejscach.

```
List<Integer> lista = new ArrayList<Integer>();
Collections.addAll(lista, 123, 234, 345, 456, 567, 678, 789);

lista.add(890);
lista.set(4, 765);
```

Interfejs *Set*

1. Kontenery typu ***Set*** przechowują po jednym elemencie. Do porównywania elementów używają metody ***.equals()*** (zresztą tak samo jak inne kontenery). Najważniejsze klasy implementujące ten interfejs to ***HashSet***, ***TreeSet*** i ***LinkedHashSet***.
2. ***HashSet*** jest optymalizowany pod względem wyszukiwania elementu. Kolejność elementów jest za to przypadkowa (zależna od funkcji haszującej).
3. ***TreeSet*** przechowuje elementy w postaci drzewa zrównoważonego. Stąd elementy są posortowane, ale dzieje się to kosztem wydajności.
4. ***LinkedHashSet*** jest praktycznie tak samo wydajny jak ***HashSet***. Poza tym pamięta kolejność wstawiania elementów.

Interfejs *Set* (przykład)

```
Set<Character> zbior = new HashSet<Character>();
for (char c : "Ciekawe ile bedzie z tego literek ...".toCharArray())
        zbior.add(c);

Set<Character> zbiorTree = new TreeSet<Character>(zbior);

for (char c : zbior)
        System.out.print(c);
System.out.println();

for (char c : zbiorTree)
        System.out.print(c);
System.out.println();
```

Program o treści jak na powyższym listingu wypisuje na wyjście dwa wiersze:

```
g debCaol.kiwtrz
 .Cabdegiklortwz
```

Interfejs *Queue*

1. Kontenery typu ***Queue*** posiadają wszystko co trzeba, aby dać nam dostęp do struktury danych implementującej kolejkę. Najważniejsze klasy implementujące kolejkę to ***PriorityQueue*** i znana nam już ***LinkedList***.
2. ***PriorityQueue*** zachowuje się jak zwykła kolejka do czasu, aż włożymy do niej obiekt z większym niż inne elementy priorytetem. Mamy wtedy gwarancję, że zostanie od wyciągnięty jako pierwszy.
3. ***LinkedList*** to udana, listowa implementacja tradycyjnej kolejki FIFO. Przy okazji, dziedzicząc także po interfejsie ***List***, jest ona wszystkim co trzeba, aby mieć w programie stos.

Interfejs *Map*

1. Kontenery typu ***Map*** to słowniki, czyli odwzorowania obiektu na obiekt. Zwane są także tablicami asocjacyjnymi. Najważniejsze klasy implementujące ten interfejs to ***HashMap***, ***TreeMap*** i ***LinkedHashMap***.
2. ***HashMap*** jest optymalizowany pod względem dostępu do wartości o podanym kluczu. Jest szybki, ale kolejność wstawiania nie ma nic wspólnego z kolejnością wypisywania.
3. ***TreeMap*** implementuje interfejs słownika jako drzewo zrównoważone. Klucze są zatem posortowane, jednak skutkuje to pewnym spadkiem wydajności podczas wstawiania i wyszukiwania wartości.
4. ***LinkedHashMap*** jest praktycznie taki sam jak ***HashMap***. Pamięta dodatkowo kolejność wstawiania par klucz-wartość.

Interfejs *Map* (przykład)

```
Map<String, Integer> wiek = new HashMap<String, Integer>();
wiek.put("Znachor", 67);
wiek.put("Czarownica", 88);
wiek.put("Uczennica", 16);

System.out.println(wiek.keySet());
System.out.println(wiek.values());
```

Program o treści jak na powyższym listingu wypisuje na wyjście dwa wiersze:

```
[Uczennica, Znachor, Czarownica]
[16, 67, 88]
```

Interfejs *Iterator*

1. Obiekty implementujące interfejs ***Iterator*** są uzyskiwane dzięki wywołaniu metody ***.iterator()*** na obiekcie implementującym interfejs ***Iterable*** (czyli m.in. na ***List***, ***Set***, ***Queue***).
2. Dodatkowo obiekty typu ***List*** zwracają iterator implementujący interfejs ***ListIterator***, który potrafi przebiegać kontener w obu kierunkach.

```
List<Integer> lista = new LinkedList<Integer>();
Collections.addAll(lista, 123, 234, 345, 456, 567, 678, 789, 890);

ListIterator<Integer> it = lista.listIterator();
while (it.hasNext()) it.next();
while (it.hasPrevious())
        System.out.print(it.previous() + " ");
System.out.println();
```

Inne możliwości języka

1. Wielowątkowość. Każdy obiekt dziedziczy po klasie ***Object*** mechanizmy implementujące funkcjonalność monitora.
2. Refleksja. Umożliwia w czasie wykonania rozpoznanie typu używanego obiektu, którego nie możemy znać nawet przy kompilacji (zastosowania: aplikacje wielowarstwowe, RMI).
3. Typy uogólnione. Są odpowiednikiem szablonów z *C++*. Mają trochę inny cel, ale używa się ich praktycznie tak samo.
4. Adnotacje. Metadane dla programisty, kompilatora lub innych narzędzi. Jedną z użyteczniejszych jest ***@Override***.

```
class Podklasa extends Klasa {
        @Override
        void metoda() {}
}
```

Popularne biblioteki

1. *Swing*. Jest to sztandardowe *GUI*, niezależne od platformy. Dosyć ascetyczne.
2. *SWT*. Także *GUI*. Jest nieco bardziej zbliżone do środowiska wykonania. Zazwyczaj czerpie z niego także wygląd.
3. *Socket*. Tego nie trzeba nikomu tłumaczyć - podstawa w sieciach TCP/IP.
4. *JUnit*. Prosty framework do przeprowadzania powtarzalnych testów jednostkowych.

Bibliografia

- Bruce Eckel, Thinking in Java, wyd. IV
- Dokumentacja do Java SE 6
- java.sun.com/features/1998/05/birthday.html
- blogs.sun.com/jonathan/entry/better_is_always_different
- en.wikipedia.org/wiki/Java_(programming_language)
- java.sun.com/docs/white/langenv/Intro.doc2.html
- en.wikipedia.org/wiki/Java_Platform,_Micro_Edition