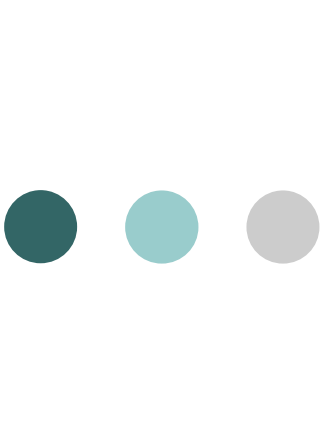


Java 5.0

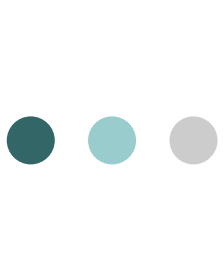
Nowości w języku

Wstęp

- Nowa wersja Javy, oznaczona 5.0 (1.5 wg. wewnętrznej numeracji Sun) wyszła 29 września 2004 roku.
- Określana jako najbardziej innowacyjna edycja Javy od czasu wyjścia języka.

O czym będę mówił

- Ograniczam temat do nowości w języku, bez omawiania nowości w Wirtualnej Maszynie Javy, bibliotekach itd..
- Duży nacisk na typy generyczne oraz na machanizm metadanych (annotations)



Historia

- 4 grudnia 1998 r. – J2SE 1.2 Playground
- 8 maja 2000 r. – J2SE 1.3 Kestler
- 12 luty 2002 r. – J2SE 1.4 Merlin
- 29 września 2004 r. – J2SE 5.0 (1.5) Tiger

Zmiany w języku

- Typy generyczne
- Typy wyliczeniowe
- Nowa pętla for
- Mechanizm zmiennej liczby argumentów
- Autoboxing/Unboxing
- Statyczny import
- Mechanizm metadanych

Nowa pętla For

- Nowa, ładniejsza metoda przechodzenia po kolekcjach.
- Do tej pory przejście wyglądało mniej więcej tak:

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

Nowa pętla For

- Z użyciem nowej pętli:

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

Nowy For – kiedy stosować

- Zalecana zawsze, kiedy tylko można.
- Nie da się zastosować jeżeli we wnętrzu pętli potrzebujemy dostęp do iteratora, np. do filtrowania kolekcji.

Typy generyczne – wstęp

- Nowy mechanizm, znany już z innych języków.
- Największe zastosowanie: różnego rodzaju kontenery obiektów, np. kolekcje.
- Powody wprowadzenia:
  - Umożliwia tworzenie klas nieograniczanych typami danych
  - Wymuszenie zgodności typów
  - Zwiększenie przejrzystości kodu

Typy generyczne

- Lista w starej wersji

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

- Lista z typami generycznymi

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

Typy generyczne – tylko mniej pisania?

- Sprawdzanie zgodności typów w momencie kompilacji, nie działania programu.
- Wymusza utrzymanie zależności pomiędzy obiektami
- Pokazuje zamiar programisty

Typy generyczne

- Definiowanie samemu klasy działającej na typach generycznych

```
public interface List<E> {
        void add(E x);
        Iterator<E> iterator();
    }
```

- Mechanizm inny niż ten z C++!
  - Nie tworzy osobnej klasy przy każdym ukonkretnieniu.
  - Nie umożliwia metaprogramowania.

Typy generyczne - dziedziczenie

- Problem – czy jeżeli B jest podklasa A, a C – klasa generyczna, to C<B> jest podklasa C<A>. Przykład: czy List<Integer> jest też List<Object>?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
lo.add(new Object());
String s = ls.get(0);
```

Typy generyczne - wildcards

- Poprzednia własność komplikuje operowanie na kolekcjach nieznanego typu, np. przejść i wypisać wszystkie elementy kolekcji.

```
void printCollection(Collection<Object> c) {
        for (Object e : c) {
            System.out.println(e);
    }}
```

Typy generyczne - wildcards

- Aby jednak umożliwić działanie na nieznanych kolekcjach wprowadzono mechanizm wildcards: <?> = nieznane.

```
void printCollection(Collection<?> c) {
        for (Object e : c) {
            System.out.println(e);
    }}
```

To już działa

Typy generyczne - wildcards

- Nie wszystko jest dozwolone przy stosowaniu wildcards:

```
Collection<?> c = new ArrayList<String>;
c.add(new Object());
```

Typy generyczne – wildcards

- Można wprowadzać ograniczenia na klasy, do jakich <?> będzie pasować. Przykład: klasa Ksztalt z podklasami Kolo i Kwadrat

```
void drawAll(Collection<? extends Ksztalt> c) {
        ...
    }
```

Metody generyczne

- Sytuacje, kiedy nie można użyć wildcards – metoda na nieznanych kolekcjach, ale musząca je modyfikować. Przykład:

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
        for (T o : a) {
            c.add(o);
    }}
```

- Warto wiedzieć:
  - Nie trzeba podawać ukonkretniania przy korzystaniu z takiej metody – jest to zadanie kompilatora żeby znaleźć najbardziej dokładne ukonkretnienie.

Typy generyczne – kiedy znikają?

- Mechanizm typów generycznych jest stosowany w momencie kompilacji.
- Kompilator dokonuje sprawdzania zgodnosci typów

Kompilacja typu generycznego

- Kompilator zna informacja o typie dla klasy
  - Jest to też zachowane dla Reflection
  - Niewykorzystywane w działającym programie
  - Usuwane z sygnatur metod
- Informacja o typie nie jest zachowywana w instancjach klasy
  - Parametryzacja jest znana tylko dla kompilatora
  - W momencie działania programu List foo jest tym samym co List<bar> foo.

Autoboxing/Unboxing

- Przerzucono z programisty na kompilator zadanie opakowywania wartości typów pierwotnych przy operacjach na kolekcjach.
  Kiedyś:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = (list.get(0)).intValue();
```

  Teraz:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Typ wyliczeniowy

- Rozwiązania stosowane do tej pory:
  - Ręczne wyliczenie na wartościach całkowitych:

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_FALL = 1;
```

  - Typesafe enum:

```
public final class Enum {
        public static final Enum TRUE = new Enum ();
        public static final Enum FALSE = new Enum ();
        private Enum () {}
    }
```

Typ wyliczeniowy – wady poprzednich rozwiązań

- Ręczne wyliczanie wartości:
  - Brak ochrony typu
  - Brak przestrzeni nazw
  - Wypisane wartości są bezużyteczne
- Typesafe Enum
  - Nie można użyć przy switch
  - Źle znoszą serializację

Typ wyliczeniowy

- Nowy mechanizm wyliczeniowy:

```
enum Season { WINTER, SPRING, SUMMER, FALL };
```

- Każdy typ wyliczeniowy posiada metodę values() zwracającą tablicę wartości.

Typ wyliczeniowy

- Typ Enum jest normalną klasą – można dodawać dodatkowe dane, dodawać specyficzne zachowanie, implementować interfejsy itd. Interfejsy Serializable i Comparable są od razu implementowane.

```
public enum Planet {
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public double mass()   { return mass; }
    public double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight(double oMass) {
        return oMass * surfaceGravity();
    }
}
```

Varargs

- Mechanizm znany już z rodziny C.
- Do tej pory do zmiennej liczby argumentów potrzebna była tablica.
- Z nowym mechanizmem:

```
public static String format(String pattern, Object... arguments);
```

‘…’ oznacza możliwość wystąpienia w tym miejscu sekwencji lub tablicy argumentów.

Static import

- Do tej pory sięganie do klasowych pól i metod wymuszało wpisanie nazwę klasy.

```
double r = Math.cos(Math.PI * theta);
```

- Korzystając z mechanizmu static import można pominąć takie prefiksowanie:

```
import static java.lang.Math.*;

double r = cos(PI * theta);
```

Metadane

- Metadane były już obecne w poprzednich wersjach Javy – np. javadoc.
- Duże wzmocnienie i uelastycznienie tego mechanizmu.
  - Dodano nowe znaczniki
  - Umożliwienie użytkownikowi definiowanie własnych znaczników

Metadane – po co wprowadzono?

- Bardziej deklaratywne programowanie
- Automatyczne tworzenie standardowego kodu wymaganego przez niektóre API (np. JAX-RPC)
- Możliwość automatycznego tworzenia wielu zgodnych ze sobą plików na podstawie tylko jednego pliku.

Annotacje

- Przykłady annotacji:

```
@Preliminary public class TimeTravel { ... }

@Copyright("2002 Yoyodyne Propulsion Systems")
public class OscillationOverthruster { ... }

@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date     = "4/1/3007"
)
public static void travelThroughTime(Date destination) { ... }
```

Definiowanie annotacji

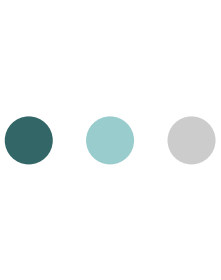
- Jak zdefiniować własną annotację:
  - Każda metoda definiuje jedno pole
  - Metody nie mogą przyjmować argumentów
  - Możliwe zwracane typy:
    - Typy pierwotne
    - String
    - Class
    - annotacja
    - Typy wyliczeniowe
    - Tablice powyższych

```
public @interface RequestForEnhancement {
        int    id();
        String synopsis();
        String engineer() default "[unassigned]";
        String date()    default "[unimplemented]";
    }
```

Metadane

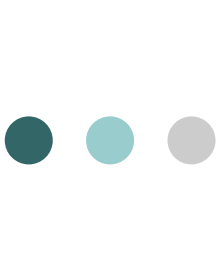
- Annotacje można podawać wszędzie tam, gdzie inne modyfikatory (np. public, private). W dobrym tonie jest podawać je jako pierwsze.
- Znaczniki można podawać bez nawiasów
- Annotacje z jednym polem nazwanym value: można w nawiasach podać tylko wartość pola
- Annotacje z wieloma polami wymienia się je w postaci
  nazwa = wartość;

```
@Preliminary public class TimeTravel { ... }

@Copyright("2002 AP Company")
public class OscillationOverthruster { ... }

@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date     = "4/1/3007"
)
public static void travelThroughTime(Date
destination) { ... }
```



Metadane

- W standardzie J5SE jest od razu zdefiniowane 7 annotacji.
  - java.lang
    - Override
    - Deprecated
    - supressWarnings
  - java.lang.annotations – meta-annotacje
    - Documented
    - Inherited
    - Target
    - Retention



Annotacje z java.lang

- Rozumiane przez kompilator javac (no.. prawie).
- Deprecated
  - Znaczenie jak javadoc @deprecated
  - Może zostać sprawdzone w działającym programie
- SupressWarnings
  - Oznacza, jakie uwagi kompilatora mają być pomijane
  - Dobre do zaznaczenia, że programista wie, co robi
  - W tej chwili nie jest wspierane przez javac

Annotacje - Override

- Służy do oznaczenia, że dana metoda ma przesłaniać metodę z klasy wyżej
- Jeżeli warunek nie zostanie spełniony kompilator zgłosi błąd
- Cenne do kontrolowania, czy na pewno przesłania się metodę zamiast tworzyć nową
- Widzi się brakujące metody w nadklasach

Meta-annotacje

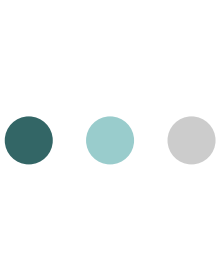
- Documented
  - Wskazuje, że element powinien być udokumentoway (np. przy użyciu javadoc)
- Inherited
  - Wskazuje, że dana annotacja ma być dziedziczona pomiędzy klasami
  - Działa tylko na klasach
    - Nie interfejsy
    - Nie przesłonięte metody

Meta-annotacje

- Target
  - Mówi, na jakich elementach dana annotacja może być użyta
  - Standardowo jest dostępna dla wszystkich elementów

Meta-annotacje

- Retention
  - Określa poziom, z jakiego widać annotacje
    - SOURCE – usuwane przy kompilacji
    - CLASS – zachowywane przez kompilator w pliku class, ale nie muszą być wyciągane przez VM
    - RUNTIME – zachowywane w pliku class i widoczne z poziomu VM
  - Domyślnie jest CLASS



Używanie annotacji

- Mało przydatne, jeżeli nie można do nich sięgnąć.
- Kiedy można użyć annotacji?
  - Tworzenie kodu
  - Budowa programu
  - Testy
  - Działająca aplikacja

Annotacje – gdzie i jak stosowane

- Tworzenie
  - Do oznaczenia stosowanych rozwiązań
- Budowa
  - Kompilator rozpoznaje annotacje z java.lang
  - Wiedza o annotacjach z poziomu SOURCE
  - Użycie apt
  - Tworzenie dodatkowych plików na podstawie istniejących

Annotacje – gdzie i jak stosowane

- Działająca aplikacja
  - Możliwe zastosowanie do aspetków
  - Dynamiczne tworzenie klas, np. BeanInfo w momencie wczytania Bean
  - Środowisko może szukać annotacji

Dojście do annotacji poprzez Reflection

- W działającym programie można zobaczyć tylko annotacje z poziomu RUNTIME
- Poprzez zastosowanie pakietów java.lang.reflect.* (Method, Constructor, Package…) można dotrzeć do annotacji dla wybranych elementów i odpowiednio zareagować.

Annotacje - przykład

```
import java.lang.annotation.*;

    public @interface Test { }

    public class Foo {
        @Test public static void m1() { }
        public static void m2() { }
        @Test public static void m3() {
            throw new RuntimeException("Boom");
        }
        public static void m4() { }
    }
```

Bardzo proste zastosowanie Reflection

```
import java.lang.reflect.*;

    public class RunTests {
    public static void main(String[] args) throws Exception {
      int passed = 0, failed = 0;
      for (Method m : Class.forName(args[0]).getMethods()) {
        if (m.isAnnotationPresent(Test.class)) {
          try {
            m.invoke(null);
            passed++;
          } catch (Throwable ex) {
            System.out.printf("Test %s failed: %s %n", m, ex.getCause());
            failed++;
          }
        }
      }
      System.out.printf("Passed: %d, Failed %d%n", passed, failed);
    }
    }
```

Działanie z annotacjami przy budowie programu - APT

- Annotation Processing Tool
- Narzędzie JDK przeznaczone do operowania z annotacjami
- Wspiera rekurencyjne wywoływanie dla tworzonych plików

Apt – żeby zadziałał

- Trzeba napisać AnnotationProcessorFactory
  - On będzie tworzył AnnotationProcessor odpowiedni dla zadanej annotacji
- Dodać tools.jar do classpath dla atp
- Wywołać apt – składnia bardzo podobna do javac, dodane opcje specjalne dla apt.
  - Automatycznie skompiluje kod wygenerowany przez AnnotationProcessor

com.sun.mirror

- Dostęp do APT poprzez paczki z com.sun.mirror
  - com.sun.mirror.apt – interfejs do apt
  - com.sun.mirror.declarations – elementy odpowiadające deklaracjom w kodzie (metody, pola, klasy …)
  - com.sun.mirror.types – typy w kodzie, wywołania elementów z deklaracji
  - com.sun.mirror.util – narzędzia do pracy na deklaracjach i typacj

AnnotationProcessorFactory

- public Collection<String> supportedAnnotationTypes()
  - Zwraca kolekcję annotacji, jakie potrafi obsłużyć dana fabryka
  - Możliwe wartości: ”*”, ”foo.*” ”foo.Bar”
- public Collection<String> supportedOptions()
  - Określa, jakie opcje z poziomu wywołania apt są dostępne
- public AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> types, AnnotationProcessorEnvironment env)
  - zwraca processor dla zadanych annotacji oraz podanego środowiska

AnnotationProcessor

- public void process()
  - Zrobić wszystko co potrzeba bezpośrednio w tej metodzie
  - Skorzystamy z mechanizmu Visitors z com.sun.mirror.util.*

Annotacje kontra Interfejsy

- W specyficznych sytuacjach można zastąpić Interface przez odpowiednią annotację
- Interface wskazuje żądane zdolności
  - Część języka
- Annotacja mówi o żądanych atrybutach
  - Mechanizm do osobnych narzędzi

Annotacja czy interface bez metod

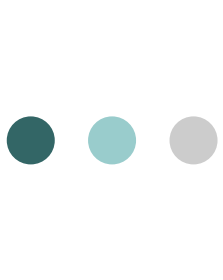
- Czasami można skorzystać z annotacji
  - serializable czy @serializable
    - Określa tylko pewną zdolność, bez metod
    - Ale: jak z annotacją wyrazić
      void saveToFile (Serializable object)
  - bean czy @bean
    - Interfejs nic tu nie dodaje, jedynie pokazuje podążanie pewnym wzorcem
    - Annotacja mogłaby być pomocna:
      - Automatyczne generowanie BeanInfo

Annotacje – ograniczenia

- Brak dziedziczenia
  @interface FixMe extends ToDo {…}

  - Meta-annotacja + apt?

    ```
    @target({ANNOTATION_TYPE})
    @interface extends { String value() }

    @extends(”ToDo”)
    @interface FixMe { … }
    ```



Annotacje - braki

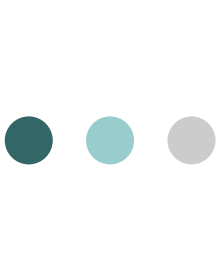
- Więcej standardowych annotacji
- Apt zintegrowany z javac
- Paczki apt i mirror w com.sun.*, dlaczego nie java.*?

Annotacje - podsumowanie

- Własne modyfikatory
- Nie zmieniają semantyki programu
  - Ale narzedzia z ich pomocą mogą to zrobić
- Raczej wyszukać i stosować standardowe annotacje niż pisać własne
- Gotowe narzędzia - apt

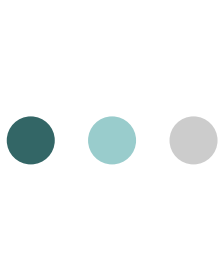
Podsumowanie

- Najbardziej znaczące w wydaniu wydają się typy generyczne oraz mechanizm annotacji
- Wprowadzenie konstrukcji skracających kod oraz poprawiających czytelność
- Największe rozszerzenie języka do tej pory.



Linki

- http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html
- http://www.javaworld.com/javaworld/jw-07-2004/jw-0719-tiger3.html
- http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html
- http://www.softwaresummit.com/2004/speakers/LandersAnnotations.pdf
- http://www.softwaresummit.com/2004/speakers/LandersGenerics.pdf



Koniec

jb209233@students.mimuw.edu.pl