Praktyka języków programowania

Krzysztof Barteczko

Cele przedmiotu

A. Zapoznanie z prostymi w użyciu konstrukcjami języka Groovy, po to by można łatwo tworzyć skrypty ułatwiające wykonywanie różnych zadań, przydatnych w studiach i pracy (np. przeszukiwanie i porządkowanie plików, automatyczne generowanie dokumentów, listów itp., automatyczne ściąganie i analizowanie informacji z Internetu, uruchamianie różnych aplikacji, komunikowanie się z nimi i integrowanie wyników ich działania).

B. zapoznanie z bardziej zaawansowanymi koncepcjami (np. z dziedziny programowania funkcyjnego, a także mechanizmami metaprogramowania i budowy DSL), które ew. mogą posłużyć do jakiegoś rodzaju prac badawczych w zakresie informatyki praktycznej

Wprowadzenie:
Języki programowania.
Java i języki JVM

© Krzysztof Barteczko, PJWSTK 2012-2017

Programowanie

PROGRAMOWANIE JEST PRZYJEMNE

Programming ... it's the only job I can think of where I get to be both an engineer and an artist. There's an incredible, rigorous, technical element to it, which I like because you have to do very precise thinking. On the other hand, it has a wildly creative side where the boundaries of imagination are the only real limitation.

~Andy Hertzfeld, about programming

PROGRAMOWANIE JEST POTRZEBNE (każdemu)

Efektywna praca na komputerze wymaga automatyzacji rutynowych czynności, inaczej całe godziny spędzamy na mało kocepcyjnych zajęciach, sprowadzających się do klikania myszką.

Automatyzacja = programowanie.

Dla informatyków ...

PROGRAMOWANIE JEST POTRZEBNE W ZAWODZIE

In software, the term architect means many things. (In software any term means many things.)

In general, however it conveys a certain gravitas, as in "I'm not just a mere programmer - I'm an architect".

This may translate into "I'm an architect now - I'm too important to do any programming".

... .

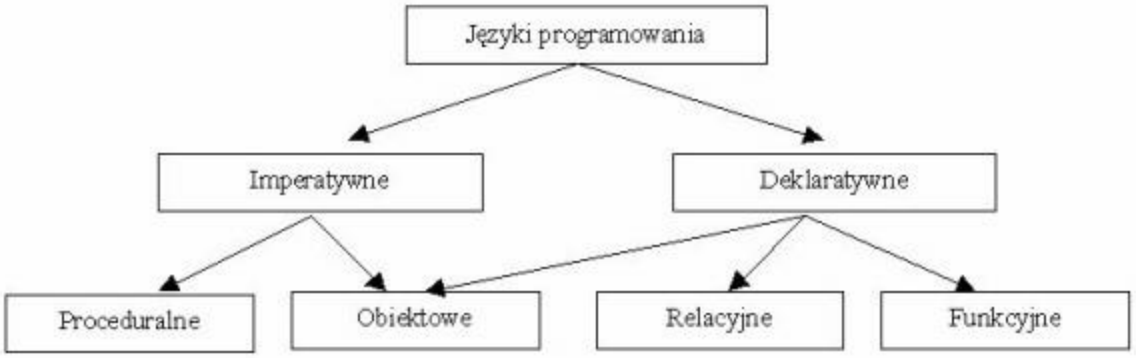
XP calls the leading technical figure the "Coach".

The meaning is clear: in XP technical leadership is shown by teaching the programmers and helping them make decisions.

It's one that requires good people skills as well as good technical skills.

~Martin Fowler, "Is design dead?"

Języki programowania - klasyfikacja



Języki imperatywne wymagają od programisty wyspecyfikowania konkretnej sekwencji kroków realizacji zadania, natomiast **języki deklaratywne** - opisują relacje pomiędzy danymi w kategoriach funkcji (**języki funkcyjne**) lub reguł (**języki relacyjne**, języki programowania logicznego)

Podejście obiektowe polega na łącznym rozpatrywaniu danych i możliwych operacji na nich, dając możliwość tworzenia i używania w programie nowych typów danych, odzwierciedlających dziedzinę problemu, **programowanie proceduralne** rozdziela dane i funkcje i nie dostarcza sposobów prostego odzwierciedlenia dziedziny rozwiązywanego problemu w strukturach danych, używanych w programie.

Języki programowania - przykłady

Przykładami języków proceduralnych są:

ALGOL, FORTRAN, PL/I, C.

Języki obiektowe to np.

SmallTalk, Java, C++, Python, C#, Ruby, Groovy, Scala

Najbardziej znanymi językmi funkcyjnymi są **Haskell, Erlang, Ocam i Clojure**, zaś językiem programowania logicznego - **Prolog**. Do operowania na bazach danych służy język relacyjny **SQL**.

Języki takie jak **Python, C#, Ruby, Groovy** czy **Scala** łączą podejście obiektowe z elementami programowania funkcyjnego.

Również **poczynając od wersji 8 w Javie dostępne są elementy programowania funkcyjnego**.

Renesans programowania funkcyjnego

lata 50-te (Lisp, McCarthy) ... akademickie.... (chmura) -> dziś

funkcje jako "first class object"

funkcje = jak w matematyce (bez efektów ubocznych)

niemodyfikowalność (ważna przy przetwarzaniu równoległym np. w chmurze)

rekurencja zamiast iteracji

na poziiomie API programowanie w kategoriach tego co się chce osiągnąc, a nie tego jak to osiągnąć

Języki kompilowane

W językach kompilowanych tekst programu (program źródłowy) tłumaczony jest na kod binarny (pośredni) przez specjalny program nazywany kompilatorem.

Kompilator jednocześnie sprawdza składniową poprawność programu, sygnalizując wszelkie błędy. Proces kompilacji jest więc nie tylko procesem tłumaczenia, ale również weryfikacji składniowej poprawności programu.

Zazwyczaj inny program zwany linkerem - generuje z kodu pośredniego gotowy do działania binarny kod wykonywalny i zapisuje go na dysku w postaci pliku typu wykonywalnego (np. z rozszerzeniem EXE lub z nadanym atrybutem "zdolny do wykonywania").

W ten sposób działają takie języki jak C czy C++. Czasem kompilator produkuje symboliczny kod binarny, który jest wykonywany za pomocą interpretacji przez program zwany interpreterem. Tak właśnie dzieje się w przypadku języka Java.

Języki interpretowane

W **językach interpretowanych** kod programu (źródłowy lub pośredni) jest odczytywany przez specjalny program zwany interpreterem, który na bieżąco - w zależności od przeczytanych fragmentów programu - przesyła odpowiednie polecenia procesorowi i w ten sposób wykonuje program.

Przykładami języków interpretowanych są: REXX, ObjectREXX, Perl, PHP.

Czasami mówimy też o takich językach jako o językach skryptowych.

Zazwyczaj języki skryptowe zapewniają dynamiczne typowanie (typy danych nie muszą być z góry ustalone i mogą zmieniać się w trakcie wykonania programu).

Plan działań

(o czym będziemy mowa?)

Baza: Groovy

- bo Groovy ma wiele elementów obecnych w innych nowoczesnych językach, jest bardzo łatwy i bardzo praktyczny w użyciu; na tym tle odniesienia do innych języków.

Oprócz podstawowych umiejętności pisania prostych programów w Groovy, poznamy też ważne koncepcje: domknięcia i lambdy, currying, złożenia, monady, kontynuacje, mataprogramowanie, tworzenie własnych wyspecjalizowanych języków (DSL) – to nie tylko ciekawe, ale istotne dla praktyki.

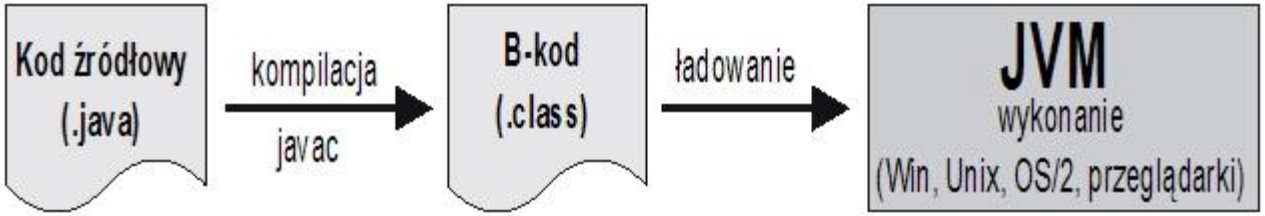
Główny cel wszystkiego: zastosowania praktyczne – jak coś szybko i łatwo zrobić, osiągnąć.

Groovy bazuje na Javie, więc na początku trochę przypomnienia/zaznajomienia z koncepcjami Javy.

Java

- uniwersalny język programowania,
- język wieloplatformowy

- składniowe podobieństwo do C/C++
- obiektowość
- automatyczne odśmiecanie pamięci
- brak wskaźników
- ścisła kontrola typów
- bezpieczne konwersje
- wymuszanie obsługi błędów za pomocą wyjątków
- wbudowane elementy współbieżności



+ bogaty zestaw standardowych bibliotek i narzędziowych interfejsów programistycznych (API), które umożliwiają w jednolity, niezależny od platformy sposób programować:

- graficzne interfejsy użytkownika (GUI)
- dostęp do baz danych
- działania w sieci,
- aplikacje rozproszone,
- aplikacje WEB,
- oprogramowanie pośredniczące (middleware),
- zaawansowana grafikę, gry i multimedia,
- aplikacje na telefony komórkowe i inne "małe" urządzenia.

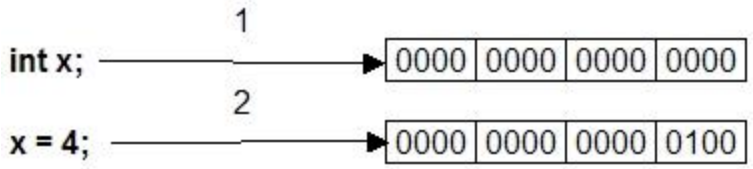
Java - mix obiektowości i nieobiektowości

- dane przetwarzane w programie: liczby i znaki (typów prostych) oraz obiekty (typów referencyjnych),
- obiekty są tworzone za pomocą wyrażenia new.
- do składowych obiektów odwołujemy się za pomocą kropki (w szczególności wywołujemy tak metody na rzecz obiektów)

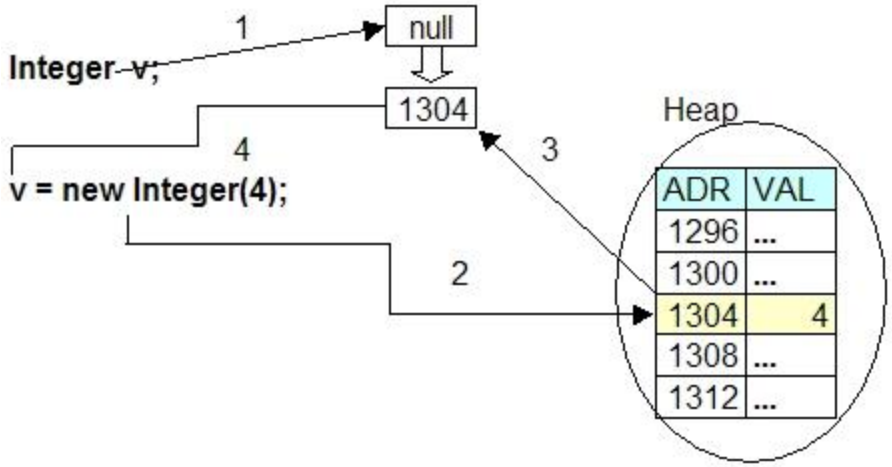
```
int x = 5;
Integer v = new Integer(5);     // Integer v = 5;      // (autoboxing)
String s = new String("Pies");  // String s = "Pies";  // (pula literałów)
Date d = new Date();
JButton b = new JButton("Ok");
s = s.substring(0,1); // wywołanie metod na rzecz obiektów
b.setText("Ok, ok");
```

Java - referencje

Referencja to wartość, która oznacza lokalizację (adres) obiektu w pamięci.



1 - przydzielenie obszaru pamięci do przechowania liczby całkowitej
2 - wpisanie do tego obszaru wartości 4



Na obiektach działamy za pomocą referencji

1 Przydzielenie pamięci zmiennej v do przechowania referencji do obiektu (4 bajty w pamięci). Referencja jest nieustalona, ma wartość **null**, co oznacza, że nie odnosi się do żadnego obiektu.

2 Opracowanie wyrażenia new powoduje przydzielenie pamięci dla obiektu klasy Integer na **stercie** pod jakimś wolnym adresem (tu symbolicznie 1304).

3 Wartością wyrażenia new jest referencja (adres 1304). Jest ona umieszczana w uprzednio przydzielonym zmiennej v obszarze pamięci

4 Zmienna v ma teraz wartość = referencji do obiektu klasy Integer, któremu w kroku 2 przydzielono pamięć na stercie (adres 1304).

Java - klasy

Realizacja idei: abstrakcji obiektowej, enkapsulacji i hermetyzacji, ponownego użycia. Więcej szczegółów: w tym materiale i o ponownym użyciu: TUTAJ

[*modCl*] **class** *ClassName* [extends *BaseClassName*] [implements *IForAC1* [, ... [,*IForACn*]]]
{

// pola klasy

[*modF*] *type VarName* [*init*]

// konstruktory

[modCtor| *ClassName*([*arg_list*]) [*throwsD*] { *body* }

// metody

[*modM*] [*type* | void] *methodName*([*arg_list*]) [*throwsD*] { *body* }

}

Specyfikatory dostępu:

brak (dla wszystkich)

public (dla wszystkich)

protected (modF i modM)

private (dla wszystkich)

Static (modF i modM, modCl) składowe statyczne są niezależne od obiektów

Abstract (modM i modCl) abstrakcyjna metoda - bez implementacji, klasa - jako baza dla klas konkretnych

- każda klasa bezpośrednio lub pośrednio pochodzi od klasy Object,
- nie ma wielodziedziczenia klas, ale można implementować dowolną liczbę interfejsów.

Java - static vs non-static

Kontekst statyczny istnieje dla klas, nie obiektów. Odwołania do składowych statycznych:

NazwaKlasy.składowa

Pola:

class S { static String s = "xxx" } // jest tylko jedna i ta sama instancja dla wszystkich obiektów klasy.

Metody:

mogą być wołane nawet wtedy gdy nie istnieje żaden obiekt klasy.

JOptionPane.showInputDialog("Enter text"");

System.out.println("ok");

Z kontekstu statycznego nie można sięgać do kontekstów niestatycznych.

Java - interfejsy (1)

Interfejs - publiczne metody abstrakcyjne, **domyślne** lub **prywatne** i/lub statycznych stałych. Ustala kontrakt dla implementujących klas.

```
interface Runnable {
    void run();
}
class Dog implements Runnable {
  void run() {  // definicja metody run dla Psa
    // jak biegnie pies?
  }
}
class Cat implements Runnable {
  void run() {  // definicja metody run dla kota
    // jak biegnie kot?
  }
}
```

Java - interfejsy (2)

Interfejsy wyznaczają typy : "loose coupling" i polimorfizm wszerz hierarchii dziedziczenia.

```
interface List
class AbstractList implements List, ...
class AbstractSequentialList extends AbstractList
class ArrayList extends AbstractList
class LinkedList extends AbstractSequentialList
...
void process(List l) { ... }
...
List list1 = new ArrayList();
List list2 = new LinkedList();
process(list1);
process(list2);
```

```
interface Speakable { ... )
interface Moveable {
  start();
  stop();
}
class Zwierz { ... }
class Vehicle implements Moveable
```

```
class Dog extends Animal implements
Speakable, Moveable
```

```
class Car extends Vehicle
```

```
void startRace(Moveable ... arg) {
  for (Moveable m : arg) m.start()
}
startRace (new Dog(.), new Car(.))
```

Java 8/9 - implementacje metod w interfejsach

W Javie 8 wprowadzono możliwość dostarczania gotowych definicji metod w interfejsach (publicznych metod statycznych oraz publicznych niestatycznych metod z użyciem słowa kluczowego ***default).***

Istotą tych zmian jest umożliwienie rozbudowy już istniejących interfejsów w taki sposób, by nie zakłócić zgodności z klasami, które już implementują te interfejsy.

Rzeczywiście, do interfejsów, które już zostały zaimplementowane przez jakieś klasy nie powinniśmy dodawać metod abstrakcyjnych, bowiem w takim przypadku klasy te przestaną działać i będą wymagać zmian w kodzie - implementacji nowych metod interfejsu.

Z drugiej strony czasem dodanie do istniejącego interfejsu nowych metod byłoby wskazane. Oczywiście, to nie mogą być metody abstrakcyjne.

(c) Krzysztof Barteczko 2015

Przykład

Wyobraźmy sobie, że po zdefiniowaniu interfejsu Speakable i użyciu go w kilku klasach, doszliśmy do wniosku, że warto by było w tym interfejsie mieć: statyczną metodę zwracającą informację o możliwych natężeniach głosu, oraz metodę String getVoice(), która upraszcza nam programowanie, bo nie wymaga podania argumentu-natężenia głosu, tylko "zwraca głos" domyślny (np. cichy). W Javie 8 możemy to zrobić tak:

```
public interface Speakable {

  int QUIET = 0;
  int LOUD = 1;

  String getVoice(int voice);

  static String getAvailableVoiceForce() {
    return "Speakable.QUIET Speakable.LOUD";
  }

  default String getVoice() {
    return getVoice(QUIET);
  }

}
```

Teraz dla wszystkich klas implementujących interfejs Speakable dostępne są te metody i np. nie zmieniając nic w definicji klasy Psa, możemy programować:

```
    Pies kuba = new Pies("Kuba");
    System.out.println(Speakable.getAvailableVoiceForce());
    System.out.println(kuba.getVoice());
```

(c) Krzysztof Barteczko 2015

Java 9 – metody prywatne interfejsów

Poczynając od Javy w wersji 9 w interfejsach możliwe jest definiowanie (dostarczanie konkretnej implementacji) metod prywatnych.
Dzięki temu można wyodrębniać i zapisywać w jednym miejscu kody używane przez różne metody domyślne czy statyczne.
Prywatne metody interfejsów są definiowane z użyciem kwalifikatora dostępu private i:

- muszą mieć implementację (ciało),
- a zatem nie mogą być abstrakcyjne (użycie specyfikatora abstract jest niedopuszczalne),
- nie mogą być opatrzone specyfikatorem default (metody domyślne są publiczne),
- mogą być niestatyczne i statyczne.
- skoro są prywatne, to mogą być wywoływane wyłącznie z innych metod interfejsu; oczywiście niestatyczne nie mogą być wywoływane z metod statycznych (czy to prywatnych czy nie).

Java 8 i 9 - mixiny

Dzięki metodom domyślnym interfejsów staje się w Javie możliwe wielodziedziczenie implementacji metod, czyli tworzenie tzw. mieszanek (ang. **mixin**).

Mixin oznacza kombinację metod z różnych klas, która może być dodana do innej klasy.

Zobaczmy przykład.

```
interface Bear {
  default String bear() { return "Bear"; }
}

interface Cat {
  default String cat() { return "Cat"; }
}

class Bintorung implements Bear, Cat {
  private String name;

  public Bintorung(String name) {
    this.name = name;
  }

  public String toString() {
    return "Nazywam się " + name
            + "\ni jestem Bintorung,"
            + "\nczyli " + bear() + cat();
  }
}
```

Podobne znaczenie mają:

mixins w Groovy

traits w Groovy

traits (cechy) w Scali

(c) Krzysztof Barteczko 2014

Java 8 - mixiny w użyciu

Poprzez implementację dwóch interfejsów w klasie Bintorung, uzyskaliśmy mixin domyślnych (zaimplementowanych) metod tych interfejsów, co pozwala na rzecz obiektu typu Bintorung wywoływać metody i bear() i cat().

Dzięki temu nasz bintorung (takie miłe zwerzątko z Azji Południowo-Wschodniej, znane także jako bearcat) po:

```
Bintorung wayan = new Bintorung("Wayan");
System.out.println(wayan);
```

może o sobie powiedzieć:

```
Nazywam się Wayan
i jestem Bintorung,
czyli BearCat
```

(c) Krzysztof Barteczko 2015

Java - klasy wewnętrzne

Klasy wewnętrzne - definiowane w innych klasach. Anonimowe = bez nazwy, służą do tworzenia ad hoc obiektów klas dziedziczących inne lub implementujących interfejsy.

```
public class Events extends JFrame {
  JButton b = new JButton("Show");
  public Events() {
      b.addMouseListener( new MouseAdapter() {
      public void mouseEntered(MouseEvent e) {
        b.setForeground(Color.RED);
      }
      public void mouseExited(MouseEvent e) {
        b.setForeground(Color.BLACK);
      }
    });
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println(e.getActionCommand());
      }
    });
   // ...
  }

  public static void main(String[] args) {
    EventQueue.invokeLater( new Runnable() {
      public void run() {
        new Events();
      }
    });

  }
}
```

Definicja anonimowej klasy wewn. dziedziczącej z klasy MouseAdapter

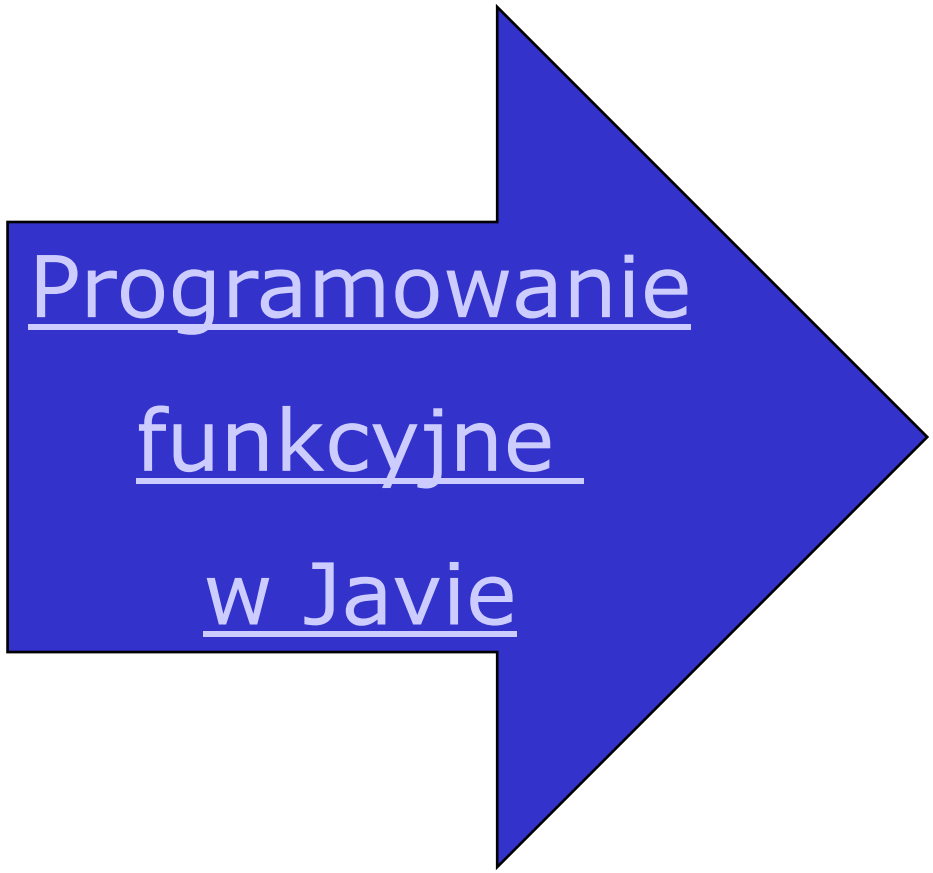
Definicja anonimowej klasy wewn. implementującej interfejs ActionListener

Definicja anonimowej klasy wewn. implementującej interfejs Runnable

Więcej szczegółów na temat interfejsów, klas wewnętrznych oraz ich praktycznego użycia można zanleźć pod następującymi odnośnikami:

Dla zinteresowanych – poniższy link.

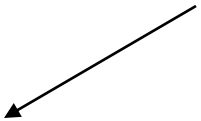
Należy jednak pamiętać, że dalej poznamy elementy programowania funkcyjnego w Groovy, które są znacznie mocniejsze co do możliwości i bardziej intuicyjne w użyciu niż w Javie.



Java - toString, equals, compareTo

Metoda **String toString()** zwraca napisową reprezentację obiektu.

```
class Para {
  int a, b;
  // ...
  public String toString() {
     return "[ " + a + ", " + b + " ]";
  }
```



Wiele kontekstów użycia:

System.out.println(p);

String msg = "Para: " + p;

showMessageDialog(null, msg);

Ma zastosowanie do przedstawiania obiektów na wizualnych listach czy tablicach (itp.)

Metoda **boolean equals(...)** - czy treść obiektów jest taka sama.

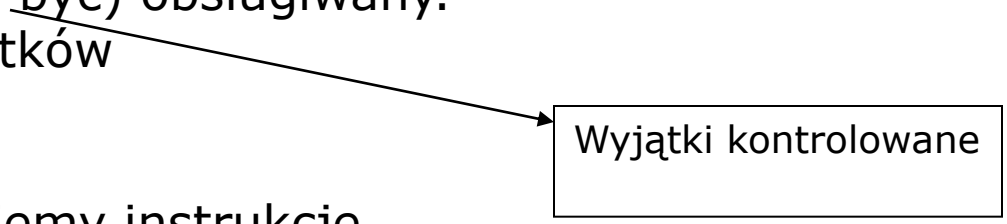
```
String s1  = "Ala ma kota", s2 = "Ala";
s2 =s2 + " ma kota";
s1 == s2 ? // NIE
s1.equals(s2) ? // TAK
```

Metoda **int compareTo()** z interfejsu Comparable służy do porównywania obiektów (większe, mniejsze, równe).

Java - wyjątki

Wyjątek - to sygnał o błędzie w trakcie wykonania programu

Wyjątek powstaje na skutek jakiegoś błędu.
Wyjątek jest zgłaszany (lub mówiąc inaczej - sygnalizowany).
Wyjątek jest (może lub musi być) obsługiwany.
Prosty schemat obslugi wyjątków




```
try {
   // ... w bloku try ujmujemy instrukcje,
   // które mogą spowodować wyjątek
} catch(TypWyjątku exc)  {
   // ... w klauzuli catch umieszczamy obsługę wyjątku
}
```

Gdy w wyniku wykonania instrukcji w bloku try powstanie wyjątek typu TypWyjatku to sterowanie zostanie przekazane do kodu umieszczonego w w/w klauzuli catch.

Zgłaszanie wyjątków

throw new *TypWyjątku*(*msg*)

Typowe:

IllegalArgumentException Przekazany metodzie lub konstruktorowi argument jest niepoprawny,

IllegalStateException Stan obiektu jest wadliwy w kontekście wywołania danej metody

NullPointerException Referencja ma wartość null w kontekście, który tego zabrania.

IndexOutOfBoundsException Indeks wykracza poza dopuszczalne zakresy

Java - pakiety

Każda klasa należy do jakiegoś pakietu.
Klasy kompilowane z deklaracją pakietu **package** ... należą do pakietu o nazwie podanej w deklaracji.
Klasy kompilowane bez deklaracji pakietu należę do pakietu "bez nazwy" (domyślnego).

Po co są pakiety? - wyznaczają przestrzenie nazw

```
class A {
   JButton b = new JButton("Ok");  // tworzymy obiekt klasy JButton
    ....                           // skąd wziąć klasę JButton?
}
```

Nazwy kwalifikowane

Kwalifikowana nazwa klasy (typu) znajdującej się w nazwanym pakiecie ma postać:

nazwa_pakietu.nazwa_klasy

np. javax.swing.JButton

```
class A {
 javax.swing.JButton b = new javax.swing.JButton("Ok");
....
}
```

Java - importy

```
import java.awt.Button;  // importuje nazwę klasy java.awt.Button

class A {
   java.awt.Frame f = new java.awt.Frame("Tytuł");
   Button b = new Button("Ok");  // użycie prostej nazwy klasy
                                 // java.awt.Button
....
}


import java.awt.*;     // importuje wszystkie nazwy klas pakietu
class A {
     Frame f =  new Frame("Tytuł");
     Button b = new Button("Ok");
....
}
```

Java - statyczne importy

Statyczny import pozwala na dostęp do statycznych składowych bez kwalifikowania ich nazwą klasy

import static TypeName.Identifier;

import static TypeName.*;

Dlatego po:

```
import static javax.swing.JOptionPane
```

można pisać:

```
showInputDialog("Enter info");
```

Java - struktura programu

Program w Javie jest zestawem definicji klas.

Poza ciałem klasy nie może być żadnego kodu programu - oprócz dyrektywy package, dyrektyw importu oraz komentarzy.

Struktura programu:

```
package ...   // deklaracja pakietu (niekonieczna)
import ...    // deklaracje importu; zwykle, ale nie zawsze potrzebne
import ...


// To jest klasa A

public class A {
...
}


// To jest klasa B

class B {
...
}
...
```

Program może być zapisany w jednym lub wielu plikach źródłowych (.java) (w szczególności: wszystkie klasy składające się na program można umieścić w jednym pliku albo każdą klasę można umieścić w odrębnym pliku).
W pliku źródłowym może by c tylko jedna publiczna klasa. Jej nazwa musi być dokładnie taka sama jak nazwa pliku.

Java - wykonanie aplikacji

Wykonanie zaczyna się od metody:

public static void main(String[] args)

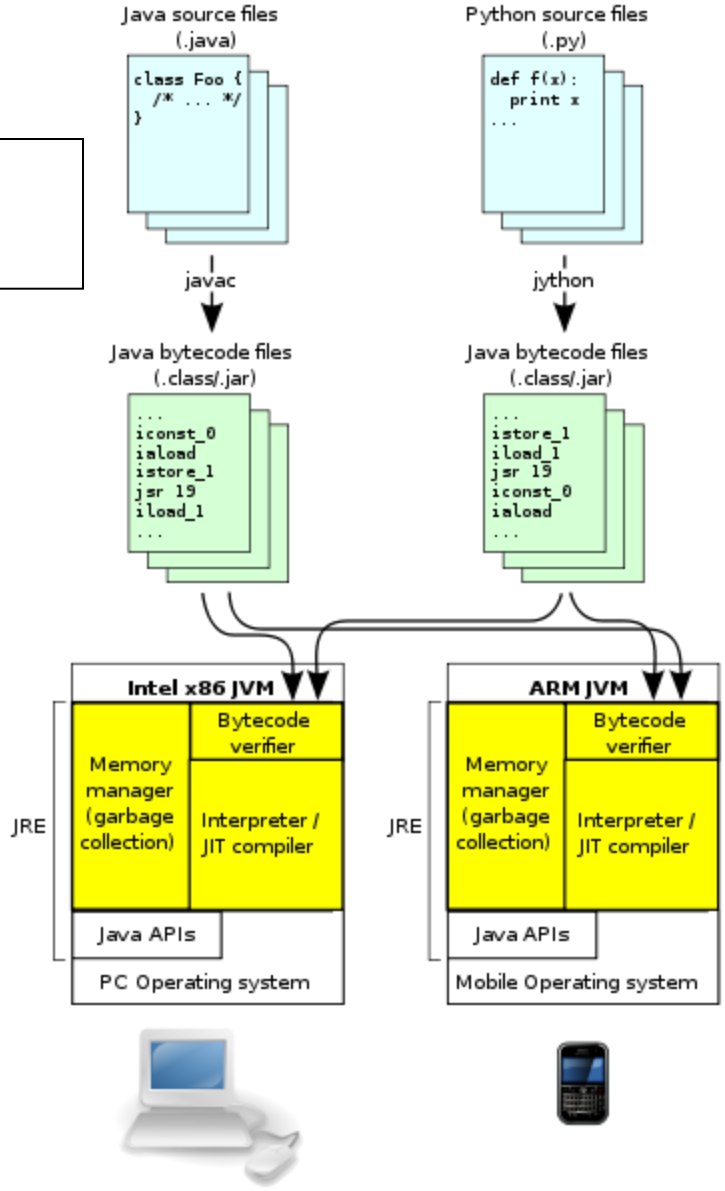
która musi być zdefiniowana w jednej z klas.

Języki JVM

JVM wykonuje bytecode.

Języki JVM = języki kompilowane do b-kodu.

Źródło rys. : Carl Byström blog



Źródło rys. : WIKI

JVM a języki dynamiczne

Języki dynamiczne (np. Groovy, JRuby) - kontrola typów w fazie wykonania, a nie kompilacji.

Zalety:

- mniejszy = łatwiejszy do pisania i czytelniejszy kod,
- większa elastyczność od języków ze statycznym typowaniem

Wady:

- mniejsza efektywność działania (ale poprawiająca się),
- wiele błędów wykrywana w fazie wykonania, a nie kompilacji.

Od Javy 7 nowa instrukcja b-kodu: invokedynamic, ulatwiająca implementację języków z dynamicznym typowaniem oraz poprawiająca efrktiwnośc ich działania

zob. Ed Ort. New JDK 7 Feature: Support for Dynamically Typed Languages in the Java Virtual Machine