Programowanie komputerów I

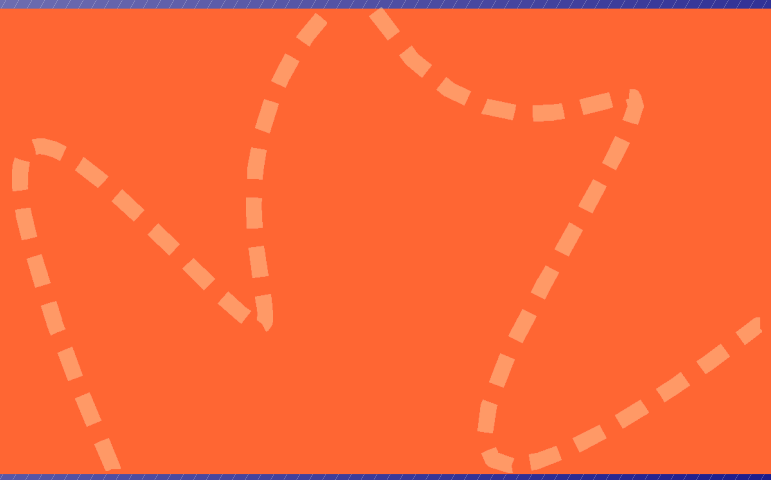
Wprowadzenie do programowania komputerów

Willy Picard
Katedra Technologii Informacyjnych
Akademia Ekonomiczna w Poznaniu
<*picard@kti.ae.poznan.pl*>

Agenda

- Cel(e) wykładu
- Od Asemblera do Javy
- „Style” programowania
- Interpretacja vs kompilacja
- Podsumowanie

Cel(e) wykładu



Ogólny cel

- Wprowadzić podstawowe pojęcia programowania komputerów
- Przedstawić programowanie w języku Java

Przegląd wykładu

Podstawowe pojęcia

- 1: Wprowadzenie
- 2: Podstawowe struktury danych & instrukcje
- 3: Programowanie obiektowe I
- 4: Programowanie obiektowe II
- 5: Programowanie obiektowe III
- 6: Zaawansowane struktury danych
- 7: Wątki & Wyjątki

CELE WYKŁADU

Przegląd wykładu

Java

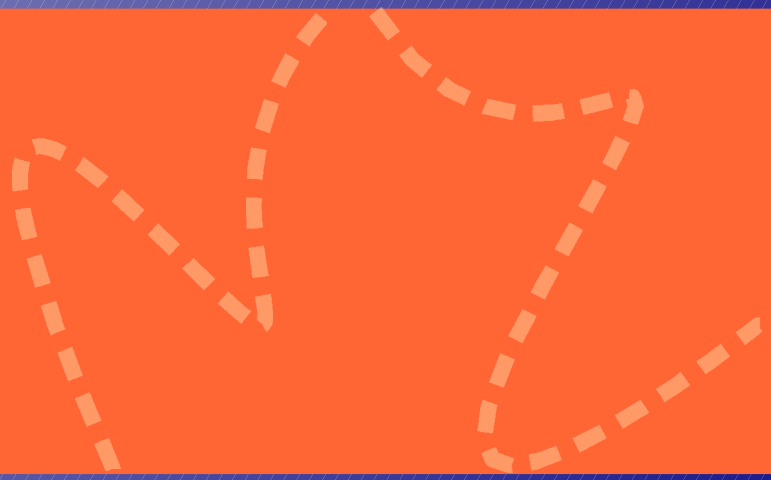
- 8: Przykład podsumowujący
- 9: Standardowy pakiet
- 10: Interfejsy graficzne – AWT
- 11: Interfejsy graficzne – Swing
- 12: Programowanie We/Wy
- 13: Programowanie sieciowe
- 14: JAR & Refleksja
- 15: Podsumowanie

CELE WYKŁADU

Cel na dziś

Wprowadzić
podstawowe
koncepcje
programowania
komputerów

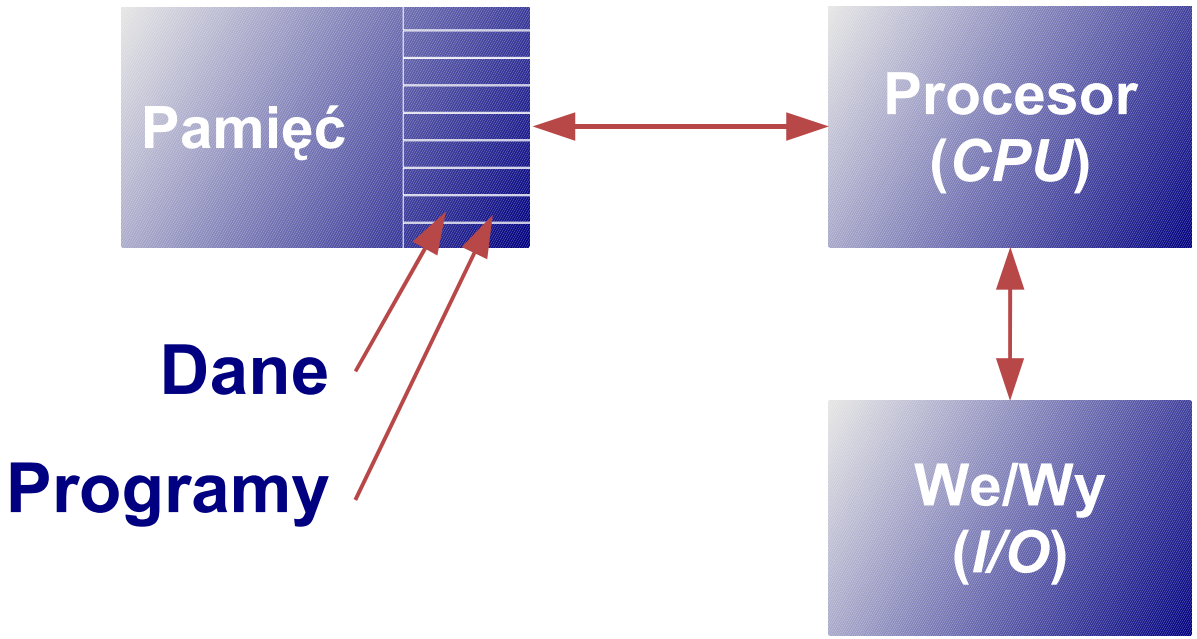
Od Asemblera do Javy



Princeton *vs* Harvard

- Architektura Harwardzka
  - Dane i programy przechowywane osobno
  - ENIAC 1946
- Architektura Princetona
  - Dane i programy przechowywane razem
  - Znana także jako architektura von Neumanna

Architektura Princeton



Zalety

- Różne zadania dla komputera
  - Programy w pamięci, nie w sprzęcie
- Programy generowane przez programy
  - Od źródła (kodu) do obiektu (program)
- Kod samo-modyfikujący się

Asembler

- Architektura zbioru instrukcji procesora
  - Ang. *CPU Instructions Set Architecture*
  - Instrukcje
  - Zasoby dostępne w instrukcjach
    - Rejestry
    - Jednostki funkcjonalne
    - Pamięć
    - Urządzenia we/wy (*I/O*)
- Asembler
  - Nazwy symboliczne

Przykład w Asemblerze

- x := y + z

- `movl y,%eax`
- `movl z,%edx`
- `leal (%edx,%eax),%ecx`
- `movl %ecx,x`

Za i przeciw Asemblera

- Kontrolowanie na niskim poziomie
- Szybkość

- Nie przenośny kod
- Za dużo detali
- Trudny w utrzymaniu
- Lepszy kod generowany przez kompilatorów

Języki wysokiego poziomu są potrzebne!!!

Java

- Skład Javy
  - Maszyna wirtualna Javy
  - Ang. *Java Virtual Machine (JVM)*
  - Specyfikacja JVM
  - Język Java
  - Narzędzia programistyczne
  - Biblioteki Java
    - Interfejsy graficzne (*GUI*)
    - Sieć
    - We/wy (*IO*)

Przykład w Javie

- Nowe okienko graficzne

```
import javax.swing.*;
...
JFrame okno = new Jframe("Moje okno");
okno.pack();
okno.setVisible(true);
```

Przykład w Asemblerze

- x := y + z

- `movl y,%eax`
- `movl z,%edx`
- `leal (%edx,%eax),%ecx`
- `movl %ecx,x`

Za i przeciw Javy

- Kontrolowanie na niskim poziomie
- Szybkość

- Przenośny kod
- Dużo ukrytych detali
- Stosunkowo prosty w utrzymaniu

Używajmy języki programowania wysokiego poziomu!!!

„Style” Programowania

Klasyfikacja języków

- Języki deklaratywne
  - Funkcjonalne — *Lisp, XSLT*
  - Logiczne — *Prolog*
- Języki proceduralne
  - Imperatywne — *Fortan, Pascal, Basic, C*
  - Obiektowe — *C++, ADA, Java*

STYLE

Języki deklaratywne

- Powiązania między zmienymi jako
  - Funkcje
    - Jeżeli student uczęszcza na „Prog. Komp.”, jego satysfakcja równa się 1, jeżeli nie, 0.
    - $f_{satysfakcja} = \begin{cases} 1, & jeżeli\ uczęszcza\ na\ "Programowanie\ Komp." \\ 0, & inaczej \end{cases}$
  - Reguły wnioskowania
    - 1. Albo student uczęszcza na „Prog. Komp.”, albo jest smutny
    - 2. Student nie jest smutny,
    - Zatem student uczęszcza na „Prog. .Komp.”
    - Formalnie: ((A lub B) & nie B) => A

Języki proceduralne

- Definicja sekwencji instrukcji
  - Manipulacja stanu systemu komputerowego
  - Możliwe efekty uboczne

```
private static int licznik=0;
public int wracajWartośćLicznika(){
   return licznik++;
}
```

Iteracja

- Powtórzenie sekwencji instrukcji
  - Zbiór początkowych warunków
  - Krok iteratywne
  - Warunek końcowy
- Pętle w językach proceduralnych

Przykład iteracji

- Silnia
- n! = 1 x 2 x 3 x 4 x ... x n, jeżeli n ≠ 0, 1 inaczej

```
public int wracajSilnię(int n) {
   if (n == 0)
      return 1;
   int silnia = 1;
   for (int i = 1; i<n+1; i++){
      silnia = silnia*i;
   }
   return silnia;
}
```

Rekurencja

- Funkcja lub procedura wywołuje samą siebie
  - Wywołanie samej siebie
  - Warunek końcowy
- Programowanie funkcjonalne

Przykład rekurencji

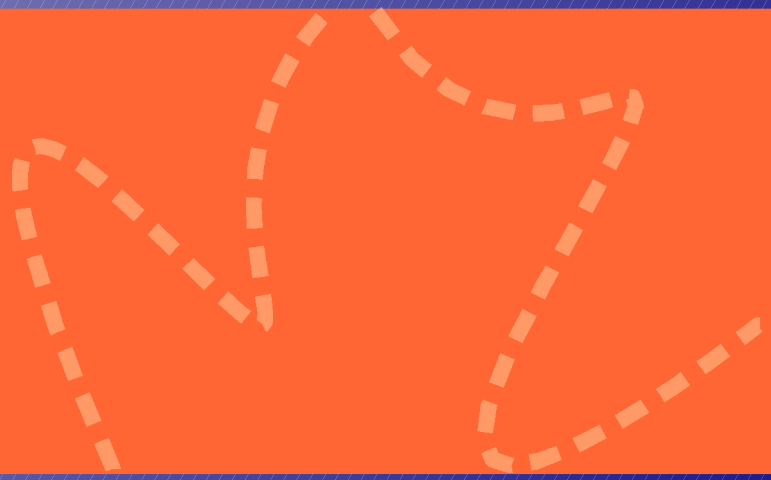
- Silnia
- n! = (n-1)! x n, jeżeli n ≠ 0, inaczej 1

```
public int wracajSilnię(int n) {
   if (n == 0)
      return 1;
   return n*wracajSilnię(n-1);
}
```

Rekurencja vs Iteracja

- Rekurencja i iteracja są ekwiwalentne
  - Każda rekurencja może być wyrażona jako iteracja i odwrotnie
- Przeciw rekurencji
  - Dużo zasobów potrzebnych
  - Często wolniejsza z powodu dużej liczby wywołań
- Za rekurencją
  - Często mniejszy kod
  - Algorytmy często w formie rekurencji
  - Niektóre struktury danych rekurencyjne (np. drzewa)

Kompilacja vs Interpretacja



Składnia i semantyka

- Składnia
  - Reguły dot. strukturę kodu w danym języku
  - Przykład: w Javie, po nazwie zmiennej numerycznej mogą wystąpić dwa znaki „+”
- Semantyka
  - Znaczenia konstrukcji językowych w danym języku
  - Przykład: w Javie, ‘j = i++;’ znaczy „najpierw wartość zmiennej *j* równa się wartości zmiennej *i*, potem dodaj 1 do wartości zmiennej *i*”

Kompilacja

- Transformacja
  - Kodu źródłowego (*source code*)
    - napisanego przez programistę w języku programowania
  - Do kodu obiektowego (*object code*)
    - wykonywalnego przez komputera
- Sprawdzenie składni i semantyki
- Kod obiektowy
  - wykonywalny
  - dla danej architektury

Interpretacja

- Wykonanie kodu źródłowego
- Wolniejsze niż wykonanie skompilowanego programu
- Szybsza niż kompilacja+wykonanie
  - Dobrze przystosowane dla prototypów i testowania
- Nie wymaga danej architektury
- Środowisko wykonania (interpreter)

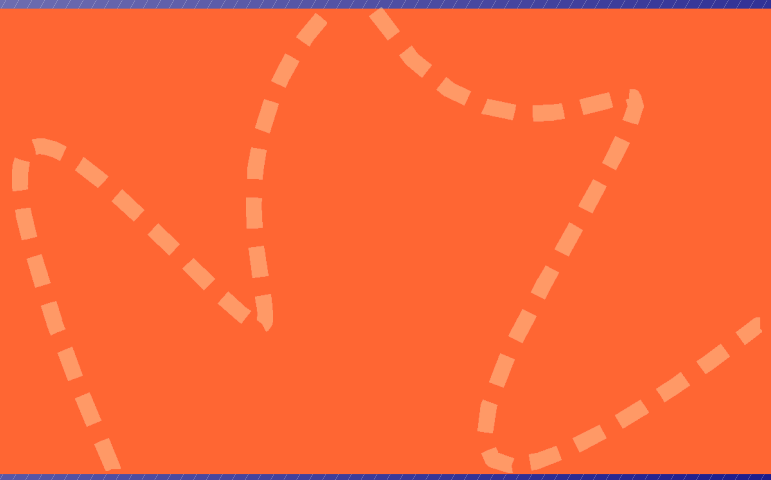
Podejście hybrydowe

- Dwa kroki
  - Kompilacja do kodu pośredniego (*byte-code*)
  - Interpretacja kodu pośredniego
- Kod pośredni
  - Zoptymalizowana i skompresowana reprezentacja kodu źródłowego
  - Nie wykonywalny bezpośrednio przez komputer
  - Nie wymaga danej architektury
  - Interpretowany przez Wirtualną Maszynę (*Virtual Machine*)

Przypadek Javy

- Podejście hybrydowe
- Najpierw, kompilacja
  - `javac MójProg.java`
  - Tworzenie `MójProg.class`
- Potem, interpretacja
  - `java MójProg`
  - `MójProg.class` jest interpretowany
- Maszyna Wirtualna Javy
  - *Java Virtual Machine (JVM)*
  - Przenośność programów w Javie

Podsumowanie



Podsumowanie

- Wiele różnych języków programowania
  - Od Asemblera
  - Do Javy
- Wybraliśmy Javę bo
  - Wysoki poziom abstrakcji
  - Przenośny kod
  - Popularny język
  - Stosunkowo łatwy do nauki ☺

Przykład

```
package pl.poznan.ae.compProg;

import java.util.*;

public class Sorter {
  private List _words;

  public void sort(String[] words){
     _words = Arrays.asList(words);
     Collections.sort(_words);
  }
  public String getSortedWords(){
     String sortedString = "";
     for (int i = 0; i< _words.size(); i++){
       sortedString += _words.get(i);
     }
     return sortedString;
  }
 public static void main(String[] args){
     Sorter sorter = new Sorter();
     sorter.sort(args);
     System.out.println(sorter.getSortedWords());
 }
}
```

Do zobaczenia za tydzień