# Wstęp do Java

## Operacje Wejścia-Wyjścia
## Programowanie Wielowątkowe

dr Krzysztof Podlaski

Wydział Fizyki i Informatyki Stosowanej
28.03.2018 Łódź

# Obsługa procesów We-Wy

- Biblioteki standardowe
  - Obszerny zestaw Klas i Interfejsów
    - Obsługa
      - Strumieni We-Wy
      - Dysków
      - Sieci
        - » java.io, java.nio
        - » java.net

# Klasa File

- Klasa File
  - Obiekty <-> pliki, katalogi
    - wykonywanie operacji
      - .list()  - zawartość katalogu
      - .mkdir() – tworzenie podkatalogów
      - .delete() – kasowanie
      - inne właściwości plików i katalogów
    - Częsty argument wejściowy dla operacji na plikach

# Ciągi znaków

- Klasa String
  - Przeładowany operator +
- Klasy przyspieszające operacje na ciągach znaków
  - StringBuilder (Java SE 5, 6 )
  - StringBuffer (wielowątkowe programowanie, wolniejsza)
    - append(String), toString()

# Operacje Wejścia i Wyjścia

- InputStream i OutputStream
  - Java  1.0
    - Mało intuicyjna
    - Operacje na bitach
- Reader i Writer
  - Java 1.1
    - Operacje na znakach (wsparcie Unicode)
    - Szybsze działanie
- Mieszanie się tych grup
  - Obsługa poprzez klasy pośredniczące

# Obsługa Strumieni

- Strumień – abstrakcyjne podejście
  - przepływ informacji źródło -> ujście danych
    - Typy źródeł
      - Pamięć
        - » Tablice bajtów
        - » Obiekty String
      - Pliki
      - Potoki (pipe)
      - Połączenia sieciowe
  - Interfejsy InputStream, OutputStream.

# Strumienie Wejścia InputStream

| | |
|---|---|
| • ByteArrayInputStream | Tablica bufor w pamięci |
| • StringBufferInputStream | String |
| • FileInputStream | Plik na dysku |
| • PipedInputStream | Potok „wylot” |
| • SequenceInputStream | Kilka strumieni |
| • FilterInputStream | Abstrakcyjna klasa |

- Wykorzystanie FilterInputStream – adaptacja strumienia dla konkretnych zastosowań

# Pochodne FilterInputStream

| | |
|---|---|
| DataInputStream | Wczytywanie danych typów podstawowych |
| BufferedInputSream | Odczyt z „buforem” |
| LineNumberInputStream | Śledzenie nr. wiersza |
| PushBackInputStream | Jednobajtowy bufor zwrotny |

# Strumienie Wyjścia OutputStream

- ByteArrayOutputStream — Tablica bufor w pamięci
- FileOutputStream — Plik na dysku
- PipedOutputStream — Potok „wlot”
- FilterOutputStream — Abstrakcyjna klasa

- Wykorzystanie FilterOutputStream

# Pochodne FilterOutputStream

| | |
|---|---|
| DataOutputStream | Zapis danych typów podstawowych |
| BufferedOutputSream | Zapis z „buforem” |
| PrintStream | Formatowanie zapisu |

# Klasy Reader i Writer

- Rozszerzenie operacji na strumieniach
  - Analogiczna hierarchia Klas
    - InputStream => Reader
    - FileInputStream => FileReader

      ...
    - OutputStream => Writer
    - FileOutputStream => FileWriter

# Modyfikacja zachowań

- FilterInputStream => FilterReader
- FilterOutputStream => FilterWritter
- BufferedInputStream => BufferedReader
- BufferedOutputStream => BufferedWriter
- PrinterStream => PrnterWriter

….

– I jeszcze inaczej

- RandomAccessFile - seek()
  - obsługa plików o rekordach z określonym rozmiarem.

# Przykłady - odczyt

- Buforowany odczyt pliku
  - Przyklad3_1
- Odczyt z Pamięci (Stringu)
  - Przyklad3_2
- Formatowany odczyt pamięci
  - Przyklad3_3

# Przykłady - zapis

- Zapis do pliku
  - Przyklad3_4
- Uproszczony zapis do pliku Java 5
  - Przyklad3_5
- Zapis-odczyt danych
  - Przyklad3_6

# System.{in, out, err}

- Standardowe Strumienie we-wy.
  - Czytanie z konsoli (System.in)
    - Przyklad3_7
  - Przekierowanie  PrintWriter -> System.out
    - Przyklad3_8
  - Przekierowania System.in, out i err
    - System.setIn(InputStream)
    - System.setOut(PrintStream)
    - System.setErr(PrintStream)

# PROGRAMOWANIE WIELOWĄTKOWE

# Wątki w programowaniu

- Tradycyjnie (jednowątkowo)
  - Cały program jednym wątkiem
    - Wszystko dzieje się liniowo
    - Jawnie określona kolejność czynności
- Wielowątkowo
  - Program zbiorem wątków
    - Każdy wątek jest liniowy
    - Nie można określić szybkości wykonania wątków
      - Nieokreślona kolejność zdarzeń

# Strona techniczna

- System wieloprocesorowy
  - procesory są niezależne
    - Wiele czynności (wątków) jednocześnie
    - Współdzielona pamięć i urządzenia
- System jednoprocesorowy
    - Dzielenie czasu procesora
      - » JVM robi to za nas
    - Współdzielenie zasobów

# Strona techniczna cd

- Podstawowe cechy
  - Wykonywanie współbieżne
  - Wspólne korzystanie z zasobów
    - Problem „kontaktu” pomiędzy wątkami
  - Problemy z testowaniem
    - Trudno wychwycić błędy
    - Zależność od platformy
      - System jednoprocesorowy
      - System wieloprocesorowy

# Po co nam wątki

- Prawie wszystko możliwe bez wątków
- Argumenty za
  - Szybkość
    - Czasami nawet przy jednym procesorze
      - » np. niema zbędnych zastojów
    - W pełni wykorzystanie wielu procesorów
  - Podział logiczny kodu
    - Programowanie poprzez działania

# Wątki a czas procesora

- Przydział czasu procesora dla wątków
  - Wielowątkowość kooperacyjna
    - Zadanie „dobrowolnie” zatrzymuje się
      - » Wtedy uruchamia się nowy wątek
      - » Wymaga ścisłej kontroli programisty
    - Zalety
      - » Niższy koszt obsługi zmiany wątku
      - » Nieograniczona ilość wątków
    - Wady
      - » Przydział pracy w systemie wieloprocesorowym

# Wątki a czas procesora cd

- Wielowątkowość z wywłaszczeniem
  - Mechanizm szeregowania zadań
    - » Każdy wątek ma wyznaczany „czas” procesora
  - Zalety
    - » Prostota kodu
    - » Wykorzystanie w pełni wielu procesorów
      - Dodaj procesor a szybciej zadziała
  - Wady
    - » Wyższe koszta obsługi
    - » Ograniczenia co do ilości wątków

# Do czego to się przyda

- Interfejs Użytkownika
  - „Natychmiastowa” reakcja na zdarzenia
    - przyciśnięcie klawisza
    - nie zmniejsza efektywności programu
  - Wiele niezależnych „okienek” (działań)
- Programowanie serwerowe
  - Obsługa wielu użytkowników jednocześnie

# Wątki w języku java

- Interfejs Runnable
  - metoda run()
  - Przyklad3_9
- Klasa Thread implementuje Runnable
  - konstruktory
    - np. Thread(Runnable)
  - metody
    - start()
    - sleep(int)
    - i wiele innych.
  - Przyklad3_10

# Pierwsze wniosek

- Nigdy nie wiadomo co  i w jakiej kolejności zostanie wykonane !!
- Ile było wątków w Przykładzie3_10 ?
    - 4
        - nie zapominajmy o wątku głównym (metoda main)

# Priorytet Wykonania

- Priorytet wątków
  - pole priority
    - niestety nie ma ujednoliconej numeracji pomiędzy JVM
      - » MAX_PRIORITY, NORM_PRIORITY, MIN_PRIORITY
  - Przyklad3_11
- Usypianie wątków
  - metoda sleep(int)
    - Java 5SE możliwość ustalenia jednostek czasu
  - Przyklad3_12

# Wątki-Demony

- Koniec programu
  - Jeżeli zakończą się wszystkie wątki
    - Chyba że są to wątki-demony
- Przyklad3_13

# Wstrzymywanie, łączenie wątków

- Jeżeli chcemy przeczekać
  - metoda yield()
    - wstrzymuje wykonywanie i zwalnia dostęp do procesora
  - Przyklad3_14
- Wątek ma ruszyć dopiero po zakończeniu innego
  - metoda join()
    - Wstrzymuje dopóki nie zakończy się inny wątek
  - Przyklad3_15

# Współdzielenie zasobów

- Współdzielenie zasobu
  - Dwa wątki (lub więcej)
  - Jeden dzielony zasób

  = Wielka szansa na błędy
  - Przyklad3_16
- Wyjście

  – Synchronizacja
  - Przyklad3_17
  - Zsynchronizowane elementy nie mogą „czytać” obiektów jednocześnie

# Współdziałanie wątków

- Wątki nie zawsze są niezależne
  - Symulacja produkcji samochodu
    - Możliwe wątki
      - tworzenie podwozia
      - tworzenie nadwozia
      - Elementy wyposażenia
      - Spawanie
    - Osobne czynności ale nie zawsze wykonywane całkowicie niezależnie.

# Współdziałanie cd.

- Możliwości współdziałania
  - Określona kolejność
    - Bądź momenty wspólne
  - Przekazywanie informacji

# Określanie następstw

- Synchronizowanie wątków
  - Wątek musi zaczekać
    - wait()  - oczekując zwalnia zablokowane zasoby
  - Poinformowanie wątków czekających
    - notify(), notifyAll()
  - Przyklad3_18

# Przesyłanie informacji

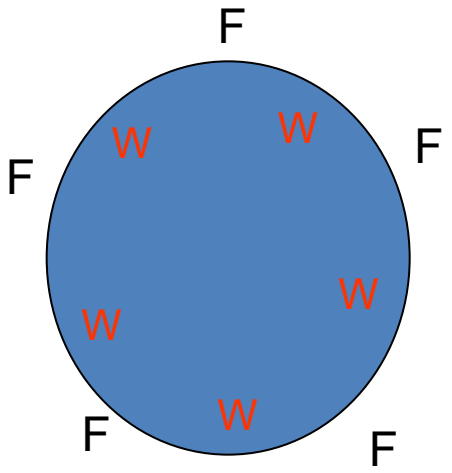
- Komunikacja pomiędzy wątkami
  - Wątek nadający
    - Tworzy Potok
  - Wątek odczytujący
    - Pobiera z Potoku
  - Przyklad3_19

# Zakleszczenie programu

- Stosując blokady (synchronizację)
  - Można doprowadzić do zablokowania wszystkich wątków
    - Zakleszczenie (deadlock), blokada wzajemna
  - Klasyczny przykład
    - Problem Ucztujących Filozofów (E. Dijkstra)

# Ucztujący filozofowie

- Filozof
  - Myśli lub je
  - do jedzenia potrzebuje dwu sztućców (widelce)
- 5-ciu filozofów
  - Filozofowie są biedni więc
    - tylko 5 sztućców
- Siedzą przy okrągłym stole
  - Filozof najpierw podnosi widelec po lewej stronie potem po prawej



# Warunki Zakleszczenia

- Program może się zablokować jeżeli
  1. Wzajemne wykluczanie przy dostępie do wspólnego zasobu
  2. Przynajmniej jeden wątek posiada zasób (blokuje go) i oczekuje na inny zasób
  3. Nie ma możliwości wywłaszczania zasobów, sam wątek musi się zrzec zasobu
  4. Możliwość zapętlonego oczekiwania
  - Muszą być spełnione wszystkie te warunki jednocześnie !

# Rozwiązanie problemu

- Wystarczy nie spełnić choć jednego z warunków
  - Np. „Odwrócić” ostatniego filozofa
    - Bierze widelce w odwrotnej kolejności
  - Oczekując zwalniać zasoby
    - np. wykorzystać metody wait() i notify()
- Java nie posiada narzędzi chroniących przed takim przypadkiem

# Błędy w programach

- Trudno wyszukać błędy
  - Dostęp do zasobów
  - Zakleszczenie
  - Problemy z Testowaniem
    - Błąd może występować bardzo rzadko
- Poprawa Błędów
  - Drobne Korekty
  - Zaprojektowanie programu od nowa
- Najlepiej unikać przy projektowaniu ☺

# Wątki Podsumowanie

- Wykonywanie wiele czynności „jednocześnie”
- Zadania mogą na siebie wpływać
    - Potrzebna dobra diagnoza potencjalnych luk i błędów
- Zły projekt prowadzi do problemów
    - BARDZO TRUDNYCH DO WYKRYCIA
        - Klientowi nie działa a u nasz przy wszystkich testach jest ok.

# Wątki Podsumowanie cd.

- Zalety
  - Lepsze wykorzystanie sprzętu
  - Czytelność kodu
  - Wygodny Interfejs Użytkownika
- Wady
  - Spowolnienie przy obsłudze wątków i synchronizacji
  - Wzrost złożoności projektu
  - Patologie (jednoczesna zmiana zasobów, zakleszczenie …)
  - Różnice pomiędzy platformami