Wprowadzenie do programowania współbieżnego

Grafika, Proste Animacje

Procesy i wątki

- **Proces** to wykonujący sie program wraz z dynamicznie przydzielanymi mu przez system zasobami (np. pamięcią operacyjna, zasobami plikowymi). Każdy proces ma własną przestrzeń adresowa. Systemy wielozadaniowe pozwalają na równoległe (teoretycznie) wykonywanie wielu procesów, z których każdy ma swój kontekst i swoje zasoby.
- **Wątek** to sekwencja działań, która wykonuje sie w kontekście danego procesu (programu) Każdy proces ma co najmniej jeden wykonujący sie watek. W systemach wielowątkowych proces może wykonywać równolegle (teoretycznie) wiele wątków, które wykonują sie jednej przestrzeni adresowej procesu.

Wątki

Z punktu widzenia programisty wspólny dostęp wszystkich wątków jednego procesu do kontekstu tego procesu ma zarówno zalety jak i wady.

Zaletą jest możliwość łatwego dostępu do wspólnych danych programu.

Wadą jest brak ochrony danych programu przed równoległymi zmianami, dokonywanymi przez różne wątki, co może prowadzić do niespójności danych, a czego unikanie wiąże się z koniecznością synchronizacji działania wątków.

(więcej szczegółów nt. modelu pamieci w filmiku szkoleniowym Google: „Java Memory Model” : http://www.youtube.com/watch?v=WTVooKLLVT8)

Klasa Thread

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

- Podstawowa klasa pozwalająca na uruchamianie tworzenie, uruchamianie i zarządzanie wątkami.

- Oprócz szeregu metod służących do zarządzania wątkami, klasa **Thread** implementuje interfejs **Runnable,** zawierający jedną metodę run() – która jest wykonywana w momencie uruchamiania wątku

- Aby uruchomić watek, należy utworzyć obiekt klasy **Thread** i dla tego obiektu wywołać metodę **start()**:

Klasa Thread

Definiowanie klasy dziedziczącej po klasie **Thread**:

```
class MyThread extends Thread {
        . . .
        public void run() {
             // kod do wykonania
        }
}
```

Tworzenie i uruchamianie wątku:

```
MyThread t = new MyThread();
t.start();
```

Metoda **run()** nie jest wywoływana jawnie lecz pośrednio poprzez metodę **start()**. Użycie metody **start()** powoduje wykonanie działań zawartych w ciele metody **run()**. Jeśli w międzyczasie nie zostanie przerwane zadanie, w ciele którego dany wątek działa, to końcem życia wątku będzie koniec działania metody **run()**.

ThreadExample.java

```
class MyThread extends Thread{

        final int threadId;

        public MyThread(int threadId) {
           super();
           this.threadId = threadId;
        }

        @Override
        public void run() {
           for(int ii = 0; ii < 10; ii++){
                System.outprintln("Thread " +
                Thread.currentThread().getName()
                + " prints " +
                ThreadExample.nextNumber());
            }
        }

}
```

ThreadExample.java

```
public class ThreadExample {
        static int currentInt = 0;

        public static void main(String[] args) {
           int nthreads = 5;
           Thread[] threads = new Thread[nthreads];

           for(int ii = 0; ii < nthreads; ii++){
                threads[ii] = new MyThread(ii);
           }

           for(int ii = 0; ii < nthreads; ii++){
                threads[ii].start();
           }
        }

        static int nextNumber(){
                currentInt++;    return currentInt;
        }

}
```

Interfejs Runnable

Inny sposób tworzenia wątków polega na implementacji do obiektu interfejsu **Runnable:**

1. Zdefiniować klasę implementującą interfejs Runnable (np. `class Klasa implements Runnable`).
2. Zdefiniowanie metody `run ()`.
3. Utworzenie obiekt tej klasy (np. `Klasa k = new Klasa();`)
4. Utworzenie obiektu klasy **Thread**, przekazując w konstruktorze referencję do obiektu utworzonego w kroku 3 (np. `Thread thread = new Thread(k);`)
5. Wywołać na rzecz nowoutworzonego obiektu klasy Thread metodę start (`thread.start();`)

RunnableExample.java

```
class MyRunnableExample implements Runnable{

  final int threadId;

  public MyRunnableExample(int threadId) {
       super();
       this.threadId = threadId;
  }

  public void run() {
       for(int ii = 0; ii < 10; ii++){
            System.out.println("Thread " +
            Thread.currentThread().getName() + " prints " +
                 RunnableExample.nextNumber());
       }
  }
}
```

RunnableExample.java

```
public class RunnableExample {

static int currentInt = 0;

public static void main(String[] args) {
    int nthreads = 5;
    Thread[] threads = new Thread[nthreads];

    for(int ii = 0; ii < nthreads; ii++){
       threads[ii] = new Thread(new MyRunnableExample(ii)); }

    for(int ii = 0; ii < nthreads; ii++){
       threads[ii].start();  }

}

static int nextNumber(){
    currentInt++;
    return currentInt;
}

}
```

„Wykonawcy” – Executors

Samodzielne zarządzanie wątkami może być czasem kłopotliwe, dlatego od Java 1.5 zaleca się uruchamiać wątki przy pomocy tzw. klas „wykonawców” (executors). Pozwalają one na odseparowanie zadań do wykonania od mechanizmów tworzenia i uruchamiania wątków.

```
ExecutorService exec =
        Executors.newFixedThreadPool(2);
        //Executors.newSingleThreadExecutor();

exec.execute(b1);
exec.execute(b2);
// b1, b2 – obiekty implementujace Runnable

exec.shutdown();
```

„Wykonawcy” – Executors

Executors.newSingleThreadExecutor() - Wykonawca uruchamiający podane mu zadania w jednym wątku (po kolei)

Executors.newFixedThreadPool(int n) - Wykonawca, prowadzący pulę wątków o zadanych maksymalnych rozmiarach

Pule wątków pozwalają na ponowne użycie wolnych wątków, a także na ew. limitowanie maksymalnej liczby wątków w puli.

JButtonRunnable.java

```
public class JButtonRunnable extends JButton implements Runnable {
(...)
String[] tekst = {"To", "jest", "animowany", "przycisk"};
(...)
public void run() {
        int i = 0;
        while(czynny){

                if (i < tekst.length-1 ) i++; else i = 0;

                setText(tekst[i]);
                try {
                        Thread.sleep(pauza);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                        }
                }

}
```

JButtonRunnable.java – metoda main()

```
JFrame f = new JFrame();
f.setLayout(new GridLayout(2,1));
f.setSize(200, 200);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JButtonRunnable b1 = new JButtonRunnable();
f.add(b1);

String[] innyTekst = {"inny", "tekst", "do", "anim", "przycisku"};
// wykorzystanie drugiego konstruktora pozwalajacego zmienic tekst i
szybkosc:
JButtonRunnable b2 = new JButtonRunnable(innyTekst, 1600);
f.add(b2);

ExecutorService exec = Executors.newFixedThreadPool(2);

exec.execute(b1);
exec.execute(b2);
exec.shutdown();

f.setVisible(true);
```

ScheduledExecutorService

ScheduledExecutorService - wykonawca zarządzający tworzeniem i wykonaniem wątków w określonym czasie lub z określoną periodycznością:

schedule(Runnable command, long delay, TimeUnit unit)

Jednokrotnie uruchomienie zadania po upływie czasu „delay”

scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

Periodyczne wykonywanie zadania (z możliwym opóźnieniem startu) , czas liczony miedzy rozpoczeciem kolejnych iteracji

scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)

Periodyczne wykonywanie zadania (z możliwym opóźnieniem startu) - czas liczony od zakonczenia jednej iteracji do rozpoczecia nastepnej

ScheduledExecutorExample.java

```
import static java.util.concurrent.TimeUnit.*;
(...)


final ScheduledExecutorService scheduler =
        Executors.newScheduledThreadPool(2);

scheduler.scheduleAtFixedRate(r, 0, 1, SECONDS);

scheduler.scheduleWithFixedDelay( (new Runnable() {
    public void run() {
    System.out.println("Po 5 sekundach - potem co 3 sekundy");
    }
 }), 5, 2, SECONDS);


import static java.util.concurrent.TimeUnit.*;
MILLISECONDS, NANOSECONDS…
```

ScheduledExecutorExample.java

```
scheduler.schedule(new Runnable() {
                @Override
        public void run() { System.out.println("Koniec
                            programu po 15 sekundach");
                scheduler.shutdownNow();
                System.exit(0);}
            }, 15, SECONDS);
```

JButtonScheduled.java

```
public class JButtonScheduled extends JButton implements
Runnable {

  String[] tekst = {"To", "jest", "przycisk",
       "animowany", "przez", ScheduledExecutorService"};
  int i = 0;

  public JButtonScheduled() {
       super();
  }

  public void run() {
       if (i < tekst.length-1 ) i++; else i = 0;
       setText(tekst[i]);
  }
```

JButtonScheduled.java

```
final ScheduledExecutorService scheduler =
        Executors.newScheduledThreadPool(3);

    // zadanie powtarzane cyklicznie - czas liczony od uruchomienia
poprzedniego wykonania takie zadanie moze byc przerwane jedynie
przez anulowanie - patrz nizej
final ScheduledFuture<?> sc1 = scheduler.scheduleAtFixedRate(b1,
1000, 50, MILLISECONDS);

// zadanie powtarzane - czas liczony od zakonczenia poprzedniego
wykonania
scheduler.scheduleWithFixedDelay(b2, 2, 1, SECONDS);

// jednokrotne wywolanie metody run z zadanym opoznieniem
scheduler.schedule(b3, 5, SECONDS);

//Anulowanie pierwszego watku po 15 sekundach
scheduler.schedule (new Runnable() {
        public void run() { sc1.cancel(true);}
        }, 15, SECONDS);
```

Wątki w Swingu

- W Java wątek odpowiadający za obsługę GUI nosi nazwę **Event Dispatch Thread** (EDT). Jak sama nazwa mówi zajmuję się on obsługą kolejki zdarzeń i informowaniem o nich obiektów nasłuchujących (czyli Listenerów), dodatkowo zarządza rozłożeniem komponentów, ich wyświetleniem, zmianą właściwości komponentów (np. dezaktywacja przycisku) i obsługą zadań. Zadaniami tymi powinny być tylko i wyłącznie krótkotrwałe procesy.

Wątki w Swingu

Zaleca się, żeby każda aplikacja tworzyła i uruchamiała GUI poprzez metodę „invokeLater”, np..

```
public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable(){
            public void run() {
               JFrame f = new JFrame();
               f.setSize(200,200);
               f.setVisible(true);
             }
        });
}
```

```
//lub w niemal równoważny sposób:
public static void main(String[] args) {
          EventQueue.invokeLater(
              new Runnable(){
                    public void run() {
                    JFrame f = new JFrame();
                    f.setSize(200,200);
                    f.setVisible(true);
              }
          });
}
```

Swing - EDT

- invokeLater (Runnable r)
- invokeAndWait (Runnable r)

Obie te metody przyjmują instancję Runnable a jej metodę run wykonują w EDT. Różnica jest taka że wywołanie invokeAndWait kończy się z chwilą wykonania metody run, a wykonanie metody invokeLater z chwilą zlecenia zadania do EDT.

Blokowanie GUI

Umieszczanie zbyt długich zadań w **Event Dispatch Thread** (EDT) może skutecznie zablokować GUI.

przykład: **LongTaskInEDT.java**

Klasa javax.swing.Timer

- Pozwala uruchomić jedno lub kilka zdarzeń akcji (ActionEvent) z zadanym opóźnieniem lub interwałem czasowym.
- Zdarzenia będą wykonywane w ustalonych interwałach w wątku EDT

```
timer = new Timer(speed, this);
timer.setInitialDelay(pause);
timer.start();
timer.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
      // kod wykonywany cyklicznie
          }
      });
```

Klasa javax.swing.Timer

Przykład: NoClick.java

```
Timer timer = new Timer(2500, null);

timer.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e)
             {
                autoPress.doClick();
             }
});

timer.start();
```

Klasa java.util.Timer

- Działanie zbliżone do **scheduledExecutor** omawianego wcześniej…

- Przykład: UtilTimer.java

```
Timer timer = new Timer(true);
timer.scheduleAtFixedRate(new TimerTask() {
      // ewentualnie w invokeLater:
      public void run() {
            autoPress.etBackground(newColor(rand.nextInt()));
                  }
    }, 250, 250 );
```

SwingWorker -

http://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html

Klasa SwingWorker została zaprojektowana do szybkiego tworzenia wątków, które pracują równolegle do EDT.

```
public abstract class SwingWorker<T,V> extends
Object implements RunnableFuture
```

SwingWorker to klasa abstrakcyjna, czyli możemy utworzyć obiekt tylko i wyłącznie klasy po niej dziedziczącej. Implementowany interfejs jest połączeniem interfejsów Runnable i Future (z niego właśnie pochodzi metoda get() zwracająca rezulat obliczeń).

Parametr T – określa typ zwracany przez metody *doInBackground()* i *get()* o których później. Jest to po prostu rezultat naszego zadania np. obrazek.

Parametr V – określa typ danych pośrednich, które może produkować zadanie, przykładowo linie tekstu z wczytywanego pliku. Dane te można wyłuskać za pomocą metody *process(List dane)*, a które przekazuje do niej metoda publish(V... dane), która powinna być używana w implementacji metody *doInBackground()* .

Uwaga! Aby metody nie zwracały nic (void) w deklaracji klasy należy podstawić parametr Void (przez duże „V”)

SwingWorker – wybrane metody

abstract protected doInBackground() - w niej powinniśmy dostarczyć zadanie do wykonania. Jeśli przesłonimy także metodę process(List dane), możemy przy pomocy publish(V... dane) przekazywać do process(List dane) cząstkowe wyniki działania zadania.

protected process(List dane) - dzięki tej metodzie możemy operować na pośrednich danych zwróconych przez publish(V... dane). W tej metodzie możemy bezpiecznie operować na komponentach graficznych ponieważ działa ona asynchronicznie w EDT.

protected void publish(V... chunks) - przesyła szczątkowe dane metodzie protected process(List dane), nie ma potrzeby jej przesłaniania, ale można to zrobić w przypadku gdy np. chcemy wykonać jakieś extra operację na dostarczanych danych.

protected void done() - wywoływana po zakończeniu zadania, wykonywana w EDT, można w niej przeprowadzić „sprzątanie” i zaprezentować w GUI główny rezultat wykonywanego zadania, czyli pobrać rezultat działania doInBackground() przy pomocy metody get()

SwingWorkerDemo.java

```
button.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
               SwingWorker<String, Void> worker = new
                              SwingWorker<String, Void>(){
            protected String doInBackground() throws Exception {
                        progressBar.setIndeterminate(true);
                        Thread.sleep(5000);
                        return "Element zwracany przez metode";
                    }
            protected void done() {
                        try {
                            etykieta.setText(get());
                            progressBar.setIndeterminate(false);
                        } catch (Exception ex) {
                            ex.printStackTrace();
                        }
                    }
                };
         worker.execute();
         }
});
```

SwingWorkerCopyFiles.java

- OnCopyActionListener - interfejs wyboru plików i uruchamiania SwingWorkera:
  copySwingWorker.execute();

- protected Void doInBackground() – kopiowanie…

- Wykorzystanie metod setProgress(), getProgress() i interfejsu „zmiany stanu”:
  setProgress(progress);
  addPropertyChangeListener (…
  progressBar.setValue(getProgress());

- Przedstawione zostały wybrane sposoby tworzenia i uruchamiania wątków.

- W ogólności tworzenie aplikacji wielowątkowych jest dość trudnym zadaniem
  (często trzeba uwzględnić np. synchronizowanie wątków, dostępu do zasobów, komunikację między wątkami itp.)

GRAFIKA

Kilka przykładów wykorzystania grafiki w Javie

- Wyświetlanie plików Graficznych:
  ImagePanelDemo.java + ImagePanel.java
  Logo.java

```
URL resource =
   getClass().getResource("obrazki/zdjecie.JPG");
```

- pobieranie zasobu z „popakietu”

GRAFIKA (Java2D)

Klasa Graphics2D – dużo większe możliwości tworzenia grafiki (m.in. łatwa obsługa przezroczystości, wypełnienia gradientowe, transformacje geometryczne, …)

- **SimpleGraphics.java** – przykład wykorzystania klasy Graphics2d
- **Gradient.java** – przykład wypełniania gradientowego
- **TextureGraphics,java** – przykład wykorzystania tekstur
- **SaveImage.java** – przykład tworzenia buforrowanego obrazka i zapisu do pliku graficznego.

Animacje

Ogólny schemat tworzenia animacji:

- Wykorzystać jedną z omawianych wcześniej metod tworzenia wątków do periodycznego przerysowywania wybranego komponentu (np. dziedziczącego z JPanel…) z każdorazową zmianą rysowanej „sceny”
- Przerysowywanie komponentu (np.Component.repaint()) zaleca się umieszczać w EDT (metodą invokeLater),

  (dla prostych animacji pominięcie tego będzie praktycznie bez wpływu na działanie…)

Przykładowe proste animacje

- **ProstaAnimacja.java** – animowany JPanel

- **PlikiGraficzne0.java**, **PlikiGraficzne.java** – animacja z kilku plików jpg (periodyczne setIcon() w JLabel)
- **PlikiGraficzneTimer.java** – j.w. z wykorzystaniem klasy Timer (swing) i możliwością sterowania wątkiem

- **Prostokat.java, Rysowanie7.java** – animacja obiektu Prostokat (na podstawie przykładów RysowanieMysz6.java z wykładu 4)

Animacje – eliminowanie migotania

- Jeśli animacje „migają” warto zastosować tzw. „podwójne buforowanie”, które polega na utworzeniu obiektu pośredniego (np. klasy BufferedImage) na którym następuje rysowanie sceny, a odświeżanie polega tylko na podmienieniu obrazka wyświetlanego w metodzie paintComponent (np. *g.drawImage(pilkaImg, 0, 0, this);*) – przykład w **Rysowanie8.java** i **RysowanieUproszczone.java**

- Tworzenie złożonej sceny może trwać dłużej (np. wczytanie pliku tła, obliczanie współrzędnych animowanych obiektów) – zdecydowanie powinno być poza EDT, natomiast odświeżanie obrazka w EDT

Polecana dalsza lektura dla zaawansowanych

- http://fivedots.coe.psu.ac.th/~ad/jg/index.html - internetowa wersja obszernej książki „Killer Game Programming in Java”