PROGRAMOWANIE W JAVIE

Andrzej Marciniak

Uniwersytet Zielonogórski
Instytut Sterowania i Systemów Informatycznych

Obsługa wyjątków
- Model wyjątków w Javie
- Typy wyjątków
- Definiowanie wyjątków

Programowanie wielowątkowe
- Model wątku w Javie
- Synchronizacja wątków
- Grupowanie wątków

Podstawy obsługi wyjątków

Wyjątkiem jest sytuacja nietypowa pojawiająca się podczas działania programu, która zakłóca jego prawidłowe wykonanie.

Mechanizm obsługi wyjątków w Javie umożliwia zaprogramowanie "wyjścia" z takich sytuacji krytycznych, dzięki czemu program nie zawiesi się po wystąpieniu błędu wykonując ciąg operacji obsługujących wyjątek. Generowanie i obsługę sytuacji wyjątkowych w Javie zrealizowano przy wykorzystaniu paradygmatu programowania zorientowanego obiektowo.

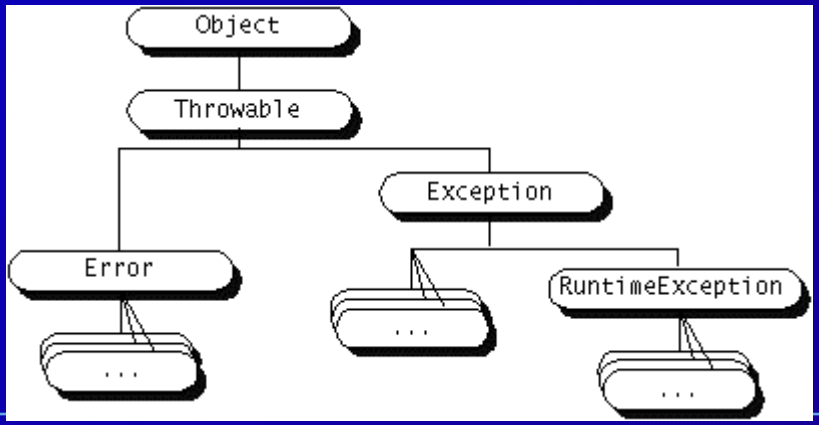
Z obsługą wyjątków związane są następujące słowa kluczowe: try, catch, throw, throws, finally .

Postać konstrukcji obsługi wyjątków

```
try
{   //blok instrukcji gdzie może wystąpić wyjątek
}
catch (PodklasaThrowable nazwaZmiennej){

    //blok instrukcji obsługujących wystąpienia sytuacji wyjątkowej
    //jest wykonywany tylko, gdy wystąpi wyjątek typu takiego jak
// typ zmiennej będącej parametrem bloku catch
}
catch (PodklasaThrowable nazwaZmiennej)
{  . . . }
catch (PodklasaThrowable nazwaZmiennej)
{   . . . }
finally //opcjonalnie
{
    // domyślna instrukcja obsługi wyjątku,
    // wystąpi nawet jeśli blok try zawiera instrukcję
    // return lub spowodował wystąpienie wyjątku
}
```

Typy wyjątków

W Javie każdy wyjątek jest reprezentowany przez obiekt określonego typu. Wszystkie wyjątki, jakie mogą wystąpić w programie muszą być podklasą klasy Throwable.



Typy wyjątków

- Wyjątki typu Error występują wtedy, gdy wystąpi sytuacja specjalna w maszynie wirtualnej (np. błąd podczas dynamicznego łączenia) -nie powinny być obsługiwane w "zwykłych" programach Javy.
- W większości programów generowane są i obsługiwane obiekty, które dziedziczą z klasy Exception. Wyjątek tego typu oznacza, że w programie wystąpił błąd, lecz nie jest to poważny błąd systemowy.
- Klasa *RuntimeException* reprezentuje wyjątki, których wystąpienie zostało spowodowane przez system czasu przebiegu Javy - są generowane automatycznie w następstwie nieprawidłowego działania programu (np.: NullPointerException, ClassCastException, IllegalThreadStateException i ArrayOutOfBoundsException).

Obsługa wyjątków przez system czasu przebiegu

```
class Wyjatek
{   public static void main(String args[])
{
 int d=0;
 int ą=42/d;
}
}
...
java.lang.ArithmeticException: / by zero
    at Wyjatek.main(Wyjatek.java:5)
Exception in thread "main"
```

System czasu przebiegu generuje (ang. throws) wyjątek. W podanym przykładzie nie została zdefiniowana procedura obsługi wyjatku, więc wykonana zostaje procedura domyślna, należąca do systemu czasu przebiegu (wyświetlająca wartość obiektu typu String).

Obsługa wyjątków przez program

```
class Wyjatek
{   public static void main(String args[])
{
    try {
        int d=0; int a=42/d;
        int c[]={1};
        c[42]=99;
        }
    catch (ArithmeticException e)
    {System.out.println("dzielenie przez zero");
    }
    catch (ArrayIndexOutOfBoundsException e)
    {System.out.println("Przekroczenie zakresu tablicy");
    }
}
}
```

Zagnieżdżone instrukcje try

```
class MultiWyjątek
{
static void procedure()
{   try
    {   int c[]={1}; c[42]=99;
        try {
             int d=0; int a=42/d;
            }
        catch (ArithmeticException e)
        {System.out.println("dzielenie przez zero");
        }

    }
    catch (ArrayIndexOutOfBoundsException e)
    {System.out.println("Przekroczenie zakresu tablicy");


} }
```

Generacja sytuacji wyjątkowych

Wyrażenie throw przekazuje sterowanie do skojarzonego z nim bloku catch (łap, blok obsługujący wystąpienie sytuacji wyjątkowej).

Jeśli nie ma bloku catch w bieżącej metodzie, sterowanie natychmiastowo, bez zwracania wartości, przekazywane jest do metody, która wywołała bieżącą funkcję. W tej metodzie szukany jest blok catch.

Jeśli blok catch nie zostanie znaleziony, przekazuje sterowanie do metody, która wywołała tę metodę. Sterowanie przekazywane jest zatem zgodnie z łańcuchem wywołań metod, aż do momentu znalezienia bloku catch odpowiedzialnego za obsługę wyjątku.

Jeśli wyjątek nie może być przechwycony, należy go zabezpieczyć (przy pomocy throws).

Generacja sytuacji wyjątkowych

```
public class WywolajWyjatek
{ static public void main(String args[]) throws Exception
    {   Liczba liczba = new Liczba();
        liczba.dziel(1);
    }
}
class Liczba
{   int m_i = 10;
    int dziel(float i) throws Exception
    {   if (i/2 != 0)
            throw new Exception("Liczba nieparzysta!");
        if (i == 0)
            throw new Exception("Dzielenie przez zero!");
        return (int)(m_i/i);
    }
}
.....
java.lang.Exception: Liczba nieparzysta!
    at Liczba.dziel(WywolajWyjatek.java:16)
    at WywolajWyjatek.main(WywolajWyjatek.java:6)
Exception in thread "main"
```

Słowo kluczowe throws

Jeżeli metoda jest zdolna generować wyjątek którego sama nie przechwytuje, należy to zaznaczyć w tekście programu, aby metody ją wywołujące mogły się przed nim zabezpieczyć. Listę wszystkich wyjątków generowanych przez daną metodę tworzy się za pomocą słowa kluczowego throws.

Próba skompilowania poprzedniego programu bez użycia throws zakończyłaby się wyświeteleniem komunikatu:
Unhandled exception type Exception

Słowo kluczowe throws - przykład

```
public class WywolajWyjatek {
    static public void main(String args[])
   {   Liczba liczba = new Liczba();
        try {
            liczba.dziel(1);
        }
        catch(Exception e){System.out.println("wyjątek");}
                // wyłapuje wyjątek "wyrzucany" poprzez funkcję dziel
    }
} class Liczba {
    int m_i = 10;
    int dziel(float i) throws Exception
                //Exception "załatwia" wszystkie podklasy
    {   if (i/2 != 0)
            throw new Exception("Liczba nieparzysta!");
        if (i == 0)
            throw new Exception("Dzielenie przez zero!");
        return (int)(m_i/i);  }
} ....
wyjątek
```

Słowo kluczowe finally

Sterowanie opuszcza blok try w przypadku wystąpienia instrukcji return lub sytuacji wyjątkowej. Java pozwala jednak zdefiniować blok instrukcji, które będą wykonane zanim sterowanie opuści metodę niezależnie od tego, czym jest to spowodowane. Jest to blok finalny (ang. finally block), nazywany tak od słowa kluczowego finally .

Zastosowanie bloku finally pozwala uniknąć dublowania kodu, który musiałby być napisany zarówno dla przypadku, gdy wystąpi wyjątek, jak i dla normalnego toku wykonania programu. Blok finalny jest odpowiednim miejscem do zwolnienia zasobów zarezerwowanych przez metodę, ponieważ zasoby te powinny być zwolnione niezależnie od tego, czy wykonanie programu przebiegło w sposób zaplanowany, czy też wystąpił wyjątek.

Przykład konstrukcji bloku finalnego

```
...
 PrintWriter out = null;
        try {
             out = new PrintWriter(new FileWriter("plik.txt"));
                 // FileWriter generuje wyjątek jeśli plik
                 //     nie może być otwarty
        out.println("tekst do pliku");
      } catch (IOException e) {
          System.err.println("Wyłapany IOException: " + e.getMessage());
      } finally {
          if (out != null) {
              System.out.println("Zamykanie PrintWriter");
              out.close();
          } else {
              System.out.println("PrintWriter nie został otwarty");
          }
        }
```

Tworzenie podklas klasy Exception

W Javie umożliwiono definiowanie klasy wyjątków, które będą obsługiwały sytuacje, uznane przez programistę za wyjątkowe.

```
class NaszException extends Exception
{
  private int detail;

    NaszException(int a)
    { detail=a; }

    public String toString()
    {
    return "NaszExeption[" + detail + "]";
    }
}
```

Przykład przechwycenia zdefiniowanego wyjątku

```
class ExceptionDemo
{
    static void oblicz(int a) throws NaszException
    {        System.out.println("Wywołana metoda oblicz (" + a +")");
             if (a>10)   throw new NaszException(a);
             System.out.println("Wszystko OK");
    }
}
    public static void main(String args[])
    {
        try {
            oblicz(1);
            oblicz(20);
            }
        catch (NaszException e)
            System.out.println("Przechwycony wyjątek: " + e);
    }
}
```

Programowanie wielowątkowe

W programowaniu sekwencyjnym , każdy program ma początek, sekwencje instrukcji do wykonania i koniec. W każdym momencie działania programu możemy wskazać miejsce, w którym znajduje się sterowanie. Taki program stanowi zatem pojedynczy, sekwencyjny przepływ sterowania, zwany wątkiem (ang. *thread*).

Program może składać się z wielu przepływów sterowania, co określamy jako programowanie wielowątkowe.

Do zadań wykonywanych współbieżnie należy m.in. odczyt i zapis plików, korzystanie z zasobów sieciowych, reagowanie na dane wprowadzane przez użytkownika, wykonywanie skomplikowanych obliczeń, wyświetlanie animacji i interfejsu użytkownika.

Wielozadaniowość vs. wielowątkowość

- W środowisku wielozadaniowym procesy nazywają się zadaniami(*tasks*) lub procesami ciężkimi (*heavy-weight processes*). Zadania zajmują oddzielne przestrzenie adresowe i z punku widzenia użytkownika są różnymi programami działającymi w tym samym systemie (np. Word i Excel). Komunikowanie się zadań, zmiana aktywnego zadania czy zmiana bieżącego kontekstu wymagają znacznych nakładów i podlegają istotnym ograniczeniom. Zwykle w danej chwili korzysta się tylko z jednego programu, nawet jeśli uruchomionych zostało więcej.
- W środowisku wielowątkowym procesy nazywa się wątkami. Współdzielą one tę samą przestrzeń adresową i należą do tego samego zadania. Dlatego ich wzajemne komunikowanie się i zmiana aktywnego procesu przebiegają szybko i są integralną częścią wykonywania pojedynczego programu.

Wątek ...

- ma początek, sekwencję instrukcji i koniec,
- nie jest niezależnym programem, jest wykonywany jako część programu,
- może być wykonywanych jednocześnie z innymi, a każdy z nich może wykonywać w tym samym czasie odmienne zadania (ang. tasks),
- jest obiektem zdefiniowanym za pomocą specjalnego rodzaju klas.
- ma swoje zarezerwowane zasoby (jak np. licznik instrukcji), lecz oprócz tego może korzystać z zasobów programu, w którym jest wykonywany.
- definiujemy jako podklasę klasy Thread i/lub implementując interfejs Runnable.

Środowisko wielowątkowe

- Jeśli program napisany wielowątkowo (ang. multithreaded), wykonywany jest na maszynie wieloprocesorowej, to różne wątki mogą być wykonywane w tym samym czasie na różnych procesorach. Sterowanie programu w takich przypadkach przebiega współbieżnie (ang. concurrent).
- Na komputerach jednoprocesorowych wykonanie programów wielowątkowych jest tylko emulowane. Emulacja ta polega na naprzemiennym przydzielaniu czasu procesora poszczególnym wątkom wg. pewnego algorytmu (zaimplementowanego w systemie operacyjnym). To, w jakim stopniu wątek będzie mógł wykorzystywać procesor, zależy od priorytetu wątku.

Model wątków w Javie

- środowisko programowania Javy jest asynchroniczne (nie występuje pojedyncza pętla zdarzeń),
- jeśli jakiś wątek został zablokowany (np. czeka na wprowadzenie danych użytkownika) to może zostać zawieszony, a w tym czasie zostanie wykonany inny. Wznowiony wątek kontynuuje działanie od miejsca w którym został zawieszony,
- wątek może sam zwolnić procesor lub może zostać wywłaszczony przez wątek o wyższym priorytecie,
- priorytety wątków są liczbami całkowitymi 1 - 10,
- jeśli dwa wątki o takim samym priorytecie zabiegają o czas procesora, to każdy z nich musi sam oddać sterowanie drugiemu (jeden nie może wywłaszczyć drugiego)

Model wątków w Javie cd

- istnieje możliwość synchronizacji wątków (konieczne np.gdy dwa wątki mają współdzielić skomplikowaną strukturę danych) za pomocą modelu synchronizacji wewnątrzprocesowej Hoare'a (zwanej monitorem),
- monitor jest "małą skrzynką" zdolną pomieścić tylko jeden wątek - pozostałe muszą czekać aż monitor zostanie zwolniony. W Javie każdy obiekt ma swój własny monitor, do którego każdy wątek może wejść przez wywołanie jednej z metod oznaczonych słowem synchronized. W czasie gdy jeden wątek wykonuje metodę synchronizowaną, żaden inny nie może wywołać jakiejkolwiek innej metody synchronizowanej tego samego obiektu,
- wątki wymieniają między sobą informacje za pomocą metod wait()i notify(). Każdy wątek może rozpocząć wykonanie synchronizowanej metody i czekać w niej na zajście pewnego zdarzenia tak długo aż inny watek nie zawiadomi go o tym że zdarzenie to już wystąpiło.

Demony (ang. daemon thread)

- Każdy wątek Javy może zostać demonem, czyli zajmować się obsługiwaniem innych wątków uruchomionych w tym samym procesie, co wątek demona. Metoda run wątku demona jest przeważnie nieskończoną pętlą, w której demon czeka na zgłoszenia zapotrzebowania na usługi dostarczane przez ten wątek.
- Program kończy się z chwilą zakończenia ostatniego wątku, który nie jest demonem (demony nie mają już dla kogo dostarczać usług i ich dalsze istnienie nie ma sensu).
- Aby wątek (obiekt klasy Thread) został demonem używamy metody setDaemon z argumentem równym true. W celu sprawdzenia, czy wątek jest demonem używana jest metoda isDaemon.

Przykłady demonów to garbage collector, czy obsługa myszki.

Klasy pomocnicze Timer i TimerTask

```
import java.util.Timer;
import java.util.TimerTask;
/* Planowanie wykonania zadania po 5 sekundach */

public class Reminder {
    Timer timer;
    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);}

    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Czas minął!");
            timer.cancel(); //Przerywa wątek zegara (timera)
    } }
    public static void main(String args[]) {
        System.out.println("Planowanie zadań");
        new Reminder(5);
        System.out.println("Zadanie zaplanowane.");
    }
}
```

Klasy pomocnicze Timer i TimerTask

Wykonanie poprzedniego programu spowoduje wyświetlenie najpierw "Zadanie zaplanowane", a następnie "Czas minął" - choć w programie kolejność występowania tych instrukcji jest odwrotna. Przykład ilustruje:

- wykorzystanie podklasy TimerTask (implementującej Runnable). Kod wykonawczy znajduje się w metodzie run().
- tworzenie wątku poprzez utworzenie instancji klasy Timer,
- tworzenie obiektu zadania (new RemindTask()),
- ustalenie harmonogramu zadań do wykonania za pomocą metody schedule(...), gdzie zadanie zegara jest pierwszym argumentem, a opóźnienie w milisek. drugim.

Ogólnie obie klasy służą do sekwencyjnego uruchamiania wątków.

Klasy pomocnicze Timer i TimerTask

Innym sposobem harmonogramowania zadań jest specyfikacja czasu, kiedy zadanie powinno być wykonane. Na przykład poniższy kod ustawia wykonanie zadania na godz. 23:01.

```
//---------ustawienie daty i godz. na 23.01 dzisiaj--
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 23);
calendar.set(Calendar.MINUTE, 1);
calendar.set(Calendar.SECOND, 0);
Date time = calendar.getTime();
//-------------------------------------------------

//==========harmonogramowanie=======================
timer = new Timer();
timer.schedule(new RemindTask(), time);
//=================================================
```

Klasy pomocnicze Timer i TimerTask

Program wykonywany jest tak długo dopóki wątki jego zegara są wykonywane. Wątek zegara można zatrzymać na cztery sposoby:

- wywołując metodę cancel dla zegara (obiektu Timer), co można zrobić z dowolnego miejsca w programie,
- generując wątek zegara jako proces typu "demon", poprzez zastosowanie new Timer(true). Jeśli wszystkie czynne wątki w programie są demonami, program przestaje się wykonywać,
- po wykonaniu wszystkich zaplanowanych zadań i usunięciu wszystkich referencji do obiektu Timer,
- po wywołaniu metody System.exit, która kończy działanie całego programu.

Powtarzanie zadań przy pomocy Timer i ...

```
                              //...niezbędne importy ...
public class Reminder {
    Timer timer;
    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), 0, seconds*1000);}
                             // parametr 0 określa opóźnienie początkowe
    class RemindTask extends TimerTask {
    int ile_Razy=3;          //tyle razy wypisze na ekranie
        public void run() {
        if (ile_Razy>0)
        { System.out.println("Czas minął!"); ileRazy--;}
        else  timer.cancel(); //Przerywa wątek zegara (timera)
    } }
    public static void main(String args[]) {
        System.out.println("Planowanie zadań");
        new Reminder(5);
        System.out.println("Zadanie zaplanowane.");
    }
}
```

Ciało wątku

Wszystkie zadania, jakie ma wykonywać wątek umieszczone są w metodzie run wątku. Po utworzeniu i inicjalizacji wątku, środowisko przetwarzania wywołuje metodę run .
W ciele metody run często pojawia się pętla. Na przykład, wątek odpowiedzialny za animację w pętli w metodzie run może wyświetlać serię obrazków. Niekiedy metoda run wątku wykonuje operacje, które zajmują dużo czasu np. ładowanie i odgrywanie dźwięków lub filmów.
Istnieją dwie metody dostarczenia metody run

- tworzenie podklasy klasy Thread i przykrywanie metody run,
- implementowanie interfejsu Runnable.

Reguła wyboru: jeśli klasa musi dziedziczyć po innej klasie (np. applet), to należy wybrać opcję drugą.

Tworzenie podklasy klasy Thread...

```
public class WatekDemo extends Thread {
    public WatekDemo(String str)   { super(str); }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try sleep((long)(Math.random() * 1000));
             catch (InterruptedException e) {}
        }
        System.out.println("Zrobione! " + getName());}
}
public class DemoDwóchWątków {
    public static void main (String[] args) {
        new WatekDemo("Jamaica").start();
        new WatekDemo("Fiji").start();
    }
}
```

Metoda run w każdej iteracji wypisuje numer iteracji, nazwę wątku i usypia wątek na losowy okres czasu.

Implementowanie interfejsu Runnable

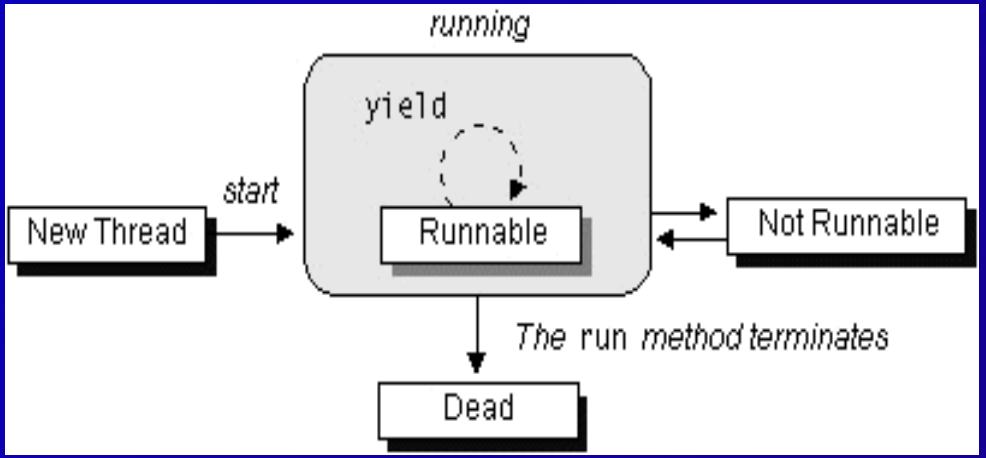
```
import java.awt.Graphics; import java.util.*;
import java.text.DateFormat; import java.applet.Applet;

public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;
    public void start() {  //startuje applet
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();  //startuje wątek zegara
        }
    }
    public void run() { //jedyna metoda interfejsu Runnable
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();      //odświeża applet
             try {    //musi być, bo metoda sleep wyrzuca wyjątek
                Thread.sleep(1000);
            } catch (InterruptedException e){}
        }
    }...
```

Implementowanie interfejsu Runnable cd

```
...
    public void paint(Graphics g) {        //wyświetla applet
        // pobiera czas i konwertuje do daty
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        // formatuje i wyświetla
        DateFormat dateFormatter = DateFormat.getTimeInstance();
        g.drawString(dateFormatter.format(date), 5, 10);
    }
    public void stop() { // zatrzymuje applet (a nie wątek)
        clockThread = null;
    }
}//KONIEC KODU
```

Szczegółowe omówienie programu na dalszych slajdach.

Cykl życia wątku



Cykl życia wątku - nowy wątek

Applet Clock jest uruchamiany metodą start() kiedy odwiedzamy odpowiednią stronę appletu.

```
public void start() {                    //startuje applet
    if (clockThread == null) {
                                      // tutaj tworzy nowy
        clockThread = new Thread(this, "Clock");
        ...
```

Po wykonaniu tej instrukcji mamy zaledwie pusty obiekt Thread. Żadne zasoby systemowe nie zostały jeszcze alokowane dla tego wątku. Kiedy wątek znajduje się w tym stanie, możemy jedynie wykonać metodę start, uruchamiającą wątek, lub stop, kończącą działania wątku. Wszelkie próby wywołania innych metod dla wątków w tym stanie nie mają sensu i powodują wystąpienie wyjątku IllegalThreadStateException.

Cykl życia wątku - wykonywanie

```
public void start() {                    //startuje applet
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();  //startuje wątek zegara
...
```

Metoda start()tworzy zasoby systemowe potrzebne do wykonania wątku, przygotowuje wątek do uruchomienia, oraz wywołuje metodę run. Od tego momentu wątek jest w stanie "wykonywany". Nie oznacza to jednak automatycznie, że wątek zostaje uruchomiony. Wiele komputerów ma tylko jeden procesor, co powoduje, że niemożliwe jest uruchomienie wielu wątków w tym samym momencie. Środowisko przetwarzania Javy musi implementować system przydziału czasu procesora (ang. scheduler), który dzieli czas procesora między wszystkie wątki będące w stanie "wykonywany". Przydzielanie czasu procesora jest omówione z priorytetami wątków.

Cykl życia wątku - nie wykonywanie

Wątek przechodzi do stanu "nie wykonywany" gdy zachodzi jedno z poniższych zdarzeń:

- wywołano jego metodę sleep,
- wywołano jego metodę suspend,
- wątek wykonuje swoją metodę wait,
- wątek jest zablokowany przy operacji wejścia / wyjścia (ang. I/O).

W applecie Clock metoda run wywołuje metodę sleep i usypia wątek.

```
public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();
             try Thread.sleep(1000);    //tutaj!!!!!!!!!!!!
             catch (InterruptedException e){} ...
```

Cykl życia wątku - przywracanie

Wątek przechodzi do stanu "wykonywany" ze stanu "nie wykonywany" gdy:

- jeśli wątek uśpiono (metoda sleep), musi upłynąć określona liczba milisekund,
- jeśli wątek zawieszono (metoda suspend), inny wątek musi wywołać metodę resume wątku, powodującą jego odwieszenie,
- jeśli wątek czeka na np. ustawienie jakiejś zmiennej, to obiekt, do którego należy ta zmienna, musi ją odstąpić a następnie wywołać metodę notify lub notifyAll,
- jeśli wątek jest zablokowany przy operacjach wejścia / wyjścia, wtedy operacje te muszą być zakończone.

W applecie Clock wątek jest przywracany po każdej sekundzie (1000 ms).

Cykl życia wątku - zakończenie działania

Wątek może zakończyć działanie z dwu powodów: albo naturalnie zakończy swe działanie (gdy jego metoda run kończy się normalnie) albo zostanie zabity (przez wywołanie metody stop w metodzie run) np.

```
clockThread.stop();
```

Metoda stop oznacza nagłe zakończenie wykonania metody run wątku i nie jest zalecana. W przypadku appletu Clock wątek kończy działanie w sposób naturalny równolegle z końcem działania appletu:

```
public void run() {
  Thread myThread = Thread.currentThread();
  while (clockThread == myThread) { // tutaj warunek końca wątku
        repaint();... // jeśli nie spełniony to dalej nie wyświetla
```

ponieważ

```
public void stop() {    // metoda stop appletu,
    clockThread = null;  //wywoływana przy jego zamknięciu
}
```

Cykl życia wątku - metoda isAlive

- Interfejs programistyczny dla klasy Thread zawiera metodę isAlive. Wynikiem wykonania metody isAlive jest wartość true, gdy wątek został uruchomiony a nie został jeszcze zakończony. Gdy wynikiem jest wartość true wiemy, że jest albo w stanie "wykonywany" lub "nie wykonywany".
- Gdy wynikiem wykonania metody isAlive jest wartość false oznacza to, że wątek jest albo w stanie "nowy wątek" lub "zakończony".

Priorytety wątków

W praktyce większość komputerów posiada tylko jeden procesor, więc w danej chwili czasu wykonywany może być tylko jeden wątek i wielowątkowość jest emulowana. Wykonanie wielu wątków na pojedynczym procesorze w jakiejś kolejności nazywane jest szeregowaniem (ang. scheduling). Java implementuje bardzo prosty, deterministyczny algorytm szeregowania znany jako "planowanie priorytetowe" (ang. fixed priority scheduling). Każdemu wątkowi przypisuje się pewien priorytet, po czym przydziela się procesor temu wątkowi, którego priorytet jest najwyższy (priorytety wątków są liczbami całkowitymi 1 - 10; stałe MIN_PRIORITY i MAX_PRIORITY klasy Thread).

Priorytety wątków

- nowy wątek dziedziczy priorytet z wątku, który go utworzył,
- domyślnie wartość priorytetu dla wątku wynosi 5 (NORM_PRIORITY),
- priorytety wątku mogą być modyfikowane w każdej chwili po utworzeniu wątku poprzez użycie metody setPriority .
- gdy wątek zatrzymuje się, ustępuje czas procesora (metoda yield) lub przechodzi do stanu "nie wykonywany", wątek o niższym priorytecie może być wykonywany

Priorytety wątków

Wybrany wątek będzie wykonywany do momentu, gdy jeden z poniższych warunków będzie spełniony

- wątek o wyższym priorytecie znalazł się w stanie "wykonywany",
- wątek ustępuje procesor lub metoda run kończy działanie,
- w systemie, w którym stosuje się segmentowanie czasu, przydział (kwant) czasu się skończył.

Uwaga: Nie jest całkowicie zagwarantowane że w danym momencie wątek o najwyższym priorytecie jest wykonywany. Program szeregujący wątki może wybrać do wykonania wątek z niższym priorytetem, aby uniknąć zagłodzenia. Z tego powodu nie należy całkowicie polegać na priorytetach (dla poprawności algorytmu), a raczej używać ich jako strategii harmonogramowania podnoszącej wydajność programu.

Segmentowanie czasu SC(ang. *time slicing*)

- gdy wątki o tym samym prioryt. czekają na przydział czasu procesora, program szeregujący wybiera jeden z nich i przydziela czas według algorytmu "szeregowania cyklicznego"(karuzeli - ang. round-robin);
- niektóre systemy (np. Windows) zwalczają "samolubne" zachowanie wątków poprzez strategię SC. W algorytmie tym ustala się małą jednostkę czasu, nazywaną kwantem czasu lub odcinkiem czasu. Planista przydziału procesora przegląda kolejkę cykliczną i każdemu wątkowi przydziela odcinek czasu nie dłuższy od jednego kwantu czasu. Gdy wątek ma czas wykonania dłuższy, niż kwant czasu, to nastąpi przerwanie wykonywania wątku i zostanie on odłożony na koniec kolejki.

Uwaga: Maszyna wirtualna Javy nie wspiera SC (nie wywłaszcza wątków dla innych o tym samym priorytecie). Jakkolwiek niektóre systemy na których uruchamia się Javę robią to. Dla przenośności rozwiązań nie korzystaj z segmentowania czasu - lepiej porządnie zaimplementować metodę yield.

Priorytety wątków - przykład

Dana jest klasa reprezentująca wątek:

```
public class SelfishRunner extends Thread {

    private int tick = 1;
    private int num;

    public SelfishRunner(int numer_wątku) {
        this.num = numer_wątku;
    }
    public void run() {
        while (tick < 400000) {
            tick++;
            if ((tick \% 50000) == 0)
                System.out.println("Wątek #" + num + ",
                                        czas = " + tick);
        }
    }
}
```

Priorytety wątków - przykład

Klasa Wyścig pozwala sprawdzić czy system operacyjny pozwala na segmentowanie czasu. Jeśli tak, to dwa wątki powinny wykonywać się naprzemiennie, co można zauważyć na ekranie gdyż metoda run klasy SelfishRunner to wyświetla.

```
public class Wyścig {
    private final static int NUMRUNNERS = 2;
    public static void main(String[] args) {
        SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];
        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i] = new SelfishRunner(i);
            runners[i].setPriority(2);
        }
        for (int i = 0; i < NUMRUNNERS; i++)
            runners[i].start();
    }
}
```

Synchronizacja wątków

Gdy dwa lub więcej wątków ubiega się jednocześnie o dostęp do współdzielonych danych, to trzeba spowodować by korzystały one z nich po kolei. W Javie każdy obiekt ma swój własny monitor, do którego każdy wątek może wejść przez wywołanie jednej z metod oznaczonych słowem synchronized. W czasie gdy jeden wątek wykonuje metodę synchronizowaną, żaden inny nie może wywołać jakiejkolwiek innej metody synchronizowanej tego samego obiektu.

Segment kodu w programie, w którym następuje dostęp do tej samej danej z różnych wątków nazywany jest sekcją krytyczną i oznaczany jest słowem synchronized (paradygmat programowania obiektowego wymaga by była to metoda).

Producent/Konsument

Jedną z sytuacji współdzielenia zasobów jest problem typu Producent/Konsument (ang. producer/consumer), gdzie producent generuje strumień danych, które są wykorzystywane (konsumowane) przez konsumenta. Strumień ten stanowi wspólny zasób, wątki muszą być zatem synchronizowane.

Niech Producent generuje liczby od 0 do 9, które są następnie składowane w obiekcie typu Pudelko. Producent, po włożeniu do pudełka liczby i wydrukowaniu jej na ekranie, zostaje uśpiony na losowo wybrany czas (0 a 100 msek.), zanim przejdzie do następnego cyklu produkcji liczby.

Producent - Konsument

```
class Producent extends Thread {
    private Pudelko pudelko;
    private int m_nLiczba;
    public Producent(Pudelko c, int liczba)
    {   pudelko = c;
        this.m_nLiczba = liczba;
    }
    public void run()
    {
        for (int i = 0; i < 10; i++)
        {   pudelko.wloz(i);
            System.out.println("Producent #" + this.m_nLiczba
                                              + " wlozyl: " + i);
            try
            {   sleep((int)(Math.random() * 100));  }
            catch (InterruptedException e) {  }
        }
    }
}
```

Producent - Konsument

Konsument podczas swego działania konsumuje wszystkie liczby złożone w pudełku, wyprodukowane przez Producenta, tak szybko, jak staną się one dostępne.

```
class Konsument extends Thread
{   private Pudelko pudelko;
    private int m_nLiczba;
    public Konsument(Pudelko c, int Liczba)
    {
        pudelko = c;
        this.m_nLiczba = Liczba;
    }
    public void run()
    {       int wartosc = 0;
            for (int i = 0; i < 10; i++)
        {
            wartosc = pudelko.wez();
                System.out.println("Konsument #" + this.m_nLiczba
                                            + " wyjal: " + wartosc);
            }
    }
}
```

Pudełko

Producent i konsument w tym przykładzie współdzielą dane przez wspólny obiekt typu Pudelko. Konsument ma prawo pobrać każdą wyprodukowaną liczbę tylko raz. Synchronizacja między tymi dwoma wątkami występuje w metodach wez() i wloz() obiektu Pudelko.

Metody wait i notifyAll służą do koordynacji wkładania i wyjmowania liczb z Pudełka. Wątek Konsumenta woła metodę wez i zajmuje monitor obiektu Pudelko na czas wykonania metody wez. Na końcu metody wez, wywołanie metody notifyAll budzi wątek Producenta oczekujący na monitor obiektu Pudelko. W tym momencie wątek Producenta zajmuje monitor i wykonuje swoje zadanie.

Pudełko

```
class Pudelko {
    private int m_nZawartosc; //zmienna warunkowa,
    private boolean m_bDostepne = false;
    public synchronized int wez()
    {   while (m_bDostepne == false)
        {  try {   wait(); }
            catch (InterruptedException e) { }
        }
        m_bDostepne = false;
        notifyAll(); // zawiadomienie Producenta
        return m_nZawartosc;
    }
    public synchronized void wloz(int wartosc)
    {   while (m_bDostepne == true)
        {   try { wait();   }
            catch (InterruptedException e) { }
        }
        m_nZawartosc = wartosc;
        m_bDostepne = true;
        notifyAll(); //zawiadomienie Konsumenta
    }
}
```


Pudełko - obiekt z monitorem

- W Javie każdy obiekt, który ma metody synchroniczne posiada swój monitor.
- Metody wait i notifyAll należą do klasy java.lang.Object i mogą być wywoływane tylko przez wątki, które założyły blokadę.
- Metoda notifyAll informuje wszystkie wątki oczekujące na monitor zajęty przez bieżący wątek o zwolnieniu tego monitora i budzi te wątki. Przeważnie jeden z oczekujących wątków zajmuje monitor i wykonuje swoje zadanie
- Metoda wait powoduje zwolnienie posiadanego monitora i przejście w stan oczekiwania do czasu, aż inny wątek powiadomi (ang. notify) go o zwolnieniu monitora obiektu.

Producent/Konsument Demo

```
class ProdKonsTest {
    public static void main(String[] args) throws Exception
    {
        Pudelko c = new Pudelko();
        Producent p1 = new Producent(c, 1);
        Konsument c1 = new Konsument(c, 1);
        p1.start();
        c1.start();
        pauza();
    }
    static void pauza() throws java.io.IOException
    {
        System.out.println("Nacisnij Enter...");
        System.in.read();
    }
}
```

Grupowanie wątków

- Każdy wątek Javy jest członkiem grupy wątków (ang. thread group). Grupowanie wątków w jednym obiekcie pozwala na jednoczesne manipulowanie wszystkimi zgrupowanymi wątkami (np. uruchamianie lub zawieszanie).
- Tworząc nowy wątek, możemy środowisku Javy pozwolić na umieszczenie wątku w domyślnej grupie wątków (w przypadku aplikacji - grupa main) lub możemy jawnie zadeklarować nową grupę wątków i dodać do niej nasz wątek. Dalej nie można już przenosić wątku do innej grupy.
- Gdy tworzymy wątek w aplecie, nowe wątki mogą być członkami innych grup wątków niż 'main', zależnie od przeglądarki, w której aplet jest uruchamiany

Tworzenie grup wątków

Klasa Thread ma trzy konstruktory pozwalające ustawić nową grupę wątków:

- public Thread(ThreadGroup, Runnable)
- public Thread(ThreadGroup, String)
- public Thread(ThreadGroup, Runnable, String)

Grupę można utworzyć

```
ThreadGroup mojaGrupaW = new ThreadGroup("Moja grupa");
Thread mojWatek = new Thread(mojaGrupaW, "wątek w mojej grupie");
```

Klasa ThreadGroup