Język programowania JAVA

Vincent Van GOGH: *Mężczyzna pijący filiżankę kawy*

W?tki

Proces to wykonujący się program wraz z dynamicznie przydzielanymi mu przez system zasobami (np. pamięcią operacyjną, zasobami plikowymi). Każdy proces ma własną przestrzeń adresową.
Wątek to sekwencja działań, która wykonuje się w kontekście danego procesu (programu)
Każdy proces ma co najmniej jeden wykonujący się wątek. W systemach wielowątkowych proces może wykonywać równolegle (teoretycznie) wiele wątków, które wykonują się w jednej przestrzeni adresowej procesu.

Programowanie wielowątkowe

Cechą charakterystyczną **programowania wielowątkowego** jest to, że tworzone programy dzieli się na wiekszą liczbę procesów, które mogą być wykonywane równocześnie. Wiele zadań występujących w interakcyjnych programach jest wykonywanych współbieżnie. Zadania, które mogą być wykonywane współbieżnie to np:

- odczytywanie i zapisywanie plików
- korzystanie z zasobów sieciowych
- reagowanie na dane wprowadzane przez użytkownika
- wykonywanie skomplikowanych trwających długo obliczeń
- wyświetlanie animacji i interfejsów użytkonika

Jednowątkowa pętla zdarzeń

W systemach jednowątkowych stosowane jest rozwiązanie zwane **pętlą zdarzeń z odpytywaniem**. W takim modelu istnieje jedna pętla nieskończona, w której wykonywane są kolejne wątki pobierane z kolejki procesów. Poszczególne wątki wykonywane są przez odpowiadające im procedury obsługi zdarzeń. Do zakończenia bieżącego wątku, nie może być w systemie podjęte żadne inne działanie.

Systemy wielowątkowe

W sytemie wielowątkowym wykorzystuje się fakt, że większość czasu wykonywania wątku to czas oczekiwania na dostępnąść zasobu. Jeżeli wszystkie procesy są niezależnymi wątkami, a system potrafi przełaczać się pomiędzy wątkiem oczekującym na dany zasób, a wątkiem gotowym do działania, to w tym samym czasie można wykonać znacznie więcej działań.

W Javie nie ma pojedyńczej pętli zdarzeń. Jeżeli jeden wątek np.:

- odczytuje dane z sieci
- czeka na dane wprowadzane przez użytkownika
- albo wyświetla animacje, nie wykonując żadnych innych działań

to pozostałe wątki mogą działać bez jakichkolwiek przerw

Systemy wielowątkowe

Równoległość działania wątków osiągana jest przez mechanizm przydzielania czasu procesora poszczególnym wykonującym się wątkom. Każdy wątek uzyskuje dostęp do procesora na kwant czasu, po czym przekazuje procesor innemu wątkowi. Zmiana wątku wykonywanego przez procesor może dokonywać się na zasadzie:

- **współpracy** – wątek sam decyduje, kiedy oddać czas procesora innym wątkom,
- **wywłaszczania** – o dostępie wątków do procesora decyduje systemowy zarządca wątków, który przydziela wątkowi kwant czasu procesora, po upływie którego odsuwa wątek od procesora i przydziela kolejny kwant czasu innemu wątkowi.

Java jest językiem wieloplatformowym, a różne systemy operacyjne stosują różne mechanizmy udostępniania wątkom procesora. Programy wielowątkowe powinny być tak pisane, by działały zarówno w środowisku „współpracy” jak i „wywłaszczania”.

Klasa Thread

Podobnie jak wszystkie pojęcia Javy, wątki reprezentowane są przez obiekty określonej klasy. Wątki reprezentowane są przez obiekty klasy **Thread**, która zawiera wszystkie metody do zarządzania wątkami. Klasa Thread jest jedynym elementem Javy pozwalającym na zarządzanie wątkami. Każdy wątek w pewnm momencie zaczyna się,wykonuje określoną liczbę działań, po czym kończy się.

Metody klasy Thread

Metody klasy **Thread** dzielimy na

- Metody statyczne – wywołuje się je bezpośrednio z nazwą klasy:

```
Thread.currentThread()
```

- Metody związane z obiektami – nie są metodami statycznymi, można z nich korzystać dopiero po utworzeniu obiektu klasy **Thread**.

Metody statyczne – zestawienie

- **Metoda currentThread()** – zwraca obiekt klasy **Thread** reprezentujący bieżący wątek.
- **Metoda yield()** – poleca systemowi czasu przebiegu Javy przerwanie wykonywania bieżacego wątku i przekazanie procesora następnemu oczekującemu wątkowi. Jest to jedyny sposób uzyskania pewności, że wątki o najniższych priorytetach będą kiedykolwiek wykonane.
- **Metoda sleep(int n)** – poleca systemowi czasu przebiegu uśpienie bieżacego wątku na n milisekund. Po upływie tego czasu wątek będzie gotowy do działania.

Metody związane z obiektami – zestawienie

Trzy podstawowe metody klasy **Thread**:

- **Metoda start()** – poleca utworzenie i uruchomienie nowego wątku. Jego działanie polega na wykonaniu metody **run()** należącej do obiektu wskazanego podczas tworzenia danego wątku. Niedozwolone jest wielokrotne wywoływanie tej metody dla tego samego wątku.
- **Metoda run()** – Jest to zasadnicza cześć każdego wątku. Nie należ ona do klasy Thread, jest to jedyna metoda interfejsu **Runnable**. Wywołuje ją metoda **Metoda start()** po poprawnym utworzeniu nowego wątku. Zakończenie działania metody **run()** oznacza zatrzymanie związanego z nią wątku.
- **Metoda inerrupt()** – umożliwia przerwanie działania wątku.

Inne metody klasy Thread – zestawienie

- **Metoda suspend()** – zawieszenie wątku
- **Metoda resume()** – wznawienie wykonywania zawieszonego wątku
- **Metoda setPriority(int p)** – Ustawia priorytet wątku
- **Metoda getPriority()** – Pobiera i zwraca priorytet wątku
- **Metoda setName(String name)** – Ustawia nazwę wątku
- **Metoda getName()** – Pobiera i zwraca nazwę wątku
- **Metoda stop()** – powoduje natychmiastowe zatrzymanie wątku. Wątek przestaje istnieć.

i inne: wait(), notify(), notifyAll(), isAlive(), join()...

Tworzenie wątków

Program korzystający z więcej niż jednego wątku możemy utworzyć na dwa sposoby:

- Bezpośrednie poprzez dziedziczenie klasy **Thread**
- Pośrednie poprzez implementację interfejsu **Runnable** .

Dziedziczenie klasy Thread

Bezpośrednie dziedziczenie z klasy Thread

```
1  class MyThread extends Thread {
2  . . .
3  public void run() {
4  . . .
5  }
6  }
```

Tworzenie obiektu i uruchamianie wątku:

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

Implementacja Runnable

Pośrednie poprzez implementację interfejsu Runnable.

```
1  class MyThreadR implements Runnable {
2  . . .
3  public void run() {
4  . . .
5  }
6  }
```

Tworzenie obiektu i uruchamianie wątku:

```
1  MyThreadR r = new MyThreadR();
2  Thread t = new Thread(r);
3  t.start();
```

Wykorzystanie interfejsu jest szczególnie przydatne, gdy dana klasa (klasa wątku) dziedziczy już z innej klasy (w Javie nie można dziedziczyć z dwóch różnych klas).

Tworzenie i uruchamianie wątków – przykład

Przykładowa implementacja interfejsu **Runnable**:

```
1  public class Watek1 implements Runnable {
2      public void run() {
3          System.out.println("to jest watek implementujacy interfejs Runnable");
4      }
5  }
```

Rozszerzenie klasy **Thread**:

```
1  public class Watek2 extends Thread {
2      public void run() {
3          System.out.println("to jest watek rozszerzajacy klase Thread");
4      }
5  }
```

A teraz tworzymy i uruchamiamy nasze wątki:

```
1  public class Uruchom {
2
3      public static void main( String[] args ) {
4          Watek1 w1 = new Watek1();
5          Watek2 w2 = new Watek2();
6
7          (new Thread(w1)).start();
8          w2.start();
9
10     }
11 }
```

Omówienie przykładu

Tworzymy dwa wątki w1 i w2. Pierwszy z nich to obiekt klasy implementującej interfejs **Runnable**, drugi to obiekt klasy dziedziczącej z klasy **Thread**. Tutaj widać podstawową różnicę pomiędzy obiema metodami tworzenia klas. Zaimplementowanie interfejsu wymusza stworzenie obiektu **Thread**, któremu jako parametr konstruktora dajemy obiekt implementujący **Runnable**. Dopiero tak utworzony obiekt posiada metodę start(), która uruchamia wątek. Drugie podejście powoduje iż nie musimy tworzyć dodatkowego obiektu.
Które podejście jest lepsze? Oczywiście poprzez implementację interfejsu **Runnable** ponieważ:

- opiera się na interfejsach co jest znacznie bardziej elastyczną metodą niż dziedziczenie.
- nie zaburza hierarchii klas poprzez dziedziczenie z klasy **Thread**.

Priorytety wątków

Java z każdym wątkiem programu kojarzy priorytet, informujący o tym jak należy traktować wątek w stosunku do innych wątków programu.
Priorytet jest liczbą całkowitą z zakresu od **MIN_PRIORITY(1)** do **MAX_PRIORITY (10)** . Domyślnym priorytetem jest **NORM_PRIORITY (5)**.
Ogólna zasada mówi, że wątki o wysokich priorytech powinny być wykonywane częściej niż wątki o niskich priorytetech. Do ustawiania priorytetu wątku służy metoda: **setPriority(int p)**.

Synchronizacja

Tworząc wątki należy pamiętać, że współdzielą one pomiędzy sobą pamięć i zasoby. Może to prowadzić do kolizji, a te do błędów. Błędy spowodowane przez kolizje jest bardzo ciężko znaleźć i usunąć. Jeżeli dwa albo więcej wątków ubiega się do współdzielonych zasobów to musimy zapewnić im synchronizację. W Javie realizowana jest ona za pomocą wbudowanych elementów językowych, tzw **monitorów**, czyli obiektu używanego jako blokady, umożliwiającej wzajemne wykluczanie się wątków. W danej chwili tylko jeden wątek może znajdować się w monitorze – pozostałe są w zawieszeniu, czekając na jego zwolnienie.

Przykład bez synchronizacji

```
1  public class Uruchom {
2      public static void main( String[] args ) {
3          Watek1 w1 = new Watek1();
4          Watek2 w2 = new Watek2();
5
6          (new Thread(w1)).start();
7          (new Thread(w2)).start();
8      }
9  public class Watek1 implements Runnable {
10     public void run() {
11         for(int i = 0; i<100000; i++){
12             System.out.println("Wątek1");
13         }
14     }
15 }
16
17 public class Watek2 implements Runnable {
18     public void run() {
19         for(int i = 0; i<100000; i++){
20             System.out.println("Wątek2");
21         }
22     }
23 }
24 }
```

Po uruchomieniu programu okaże się, że przez pewien czas wypisywane są naprzemiennie oba napisy. Z tego wnosek, że wątki wykonywane są naprzemiennie.

Synchronized

Jeżeli istnieje jedna lub wiecej metod mających wpływ na wewnętrzy stan obiektu w środowisku wielowątkowym należy zastosować mechanizm synchronizacji. Poprzez umieszczenie tych metod w blokach synchronizowanych używakjąc słowa kluczowego **synchronized** . Ogólna postać bloku synchronizownego:

```
1 synchronized (obiekt) instrukcja
```

- **obiekt** – to odwołanie do dowolnego obiektu.
- **instrukcja** – to najczęsciej instrukcja złożona zawierająca wywołanie jednej z metod **obiektu**.

albo:

```
1 synchronized metoda(){}
```

gdzie **metoda** jest synchronizowaną metodą.

Przykład z synchronizacją

```
1  public class Uruchom {
2      public static void main( String[] args ) {
3          Watek1 w1 = new Watek1();
4          Watek2 w2 = new Watek2();
5
6          (new Thread(w1)).start();
7          w2.start();
8      }
9
10 public static synchronized void wypisz(String tekst){
11         for(int i = 0; i<10000; i++)
12             System.out.println(i+": "+tekst);
13     }
14 }
15
16 public class Watek1
17     implements Runnable {
18     public void run() {
19         Uruchom.wypisz("Watek1");
20     }
21 }
22
23 public class Watek2
24     extends Thread {
25     public void run() {
26         Uruchom.wypisz("Watek2");
27     }
28 }
```

Po uruchomieniu programu nie zaobserwujemy już wymieszania się tekstów pochodzących z różnych watków

Wyjątki

Wyjątek jest to nietypowa sytuacja pojawiająca się podczas wykonywania programun np. wystąpienie błędu polegającego na próbie otwarcia pliku, który nie istnieje.

Java posiada mechanizm obsługi wyjątków. Mechanizm obsługi wyjątków w Javie umożliwia zaprogramowanie "wyjścia" sytuacji krytycznych, dzięki czemu program nie zawiesi się po wystąpieniu błędu wykonując ciąg operacji obsługujących wyjątek. Wystąpienie sytuacji wyjątkowej przerywa "normalny" tok wykonania programu.

Obsługa wyjątków

W Javie do obsługi wyjątków służy pięć słów kluczowych:

- **try** – określa sekwencję instrukcji, podczas których może wystąpić błąd.
- **catch** – przechwytuje wyjątek określonego typu
- **throw** – służy do generowania wyjątków określonego typu
- **throws** – określa listę wszystkich wyjątków generowanych przez daną metodę
- **finally** – blok instrukcji do obsługi wyjątków wykonywany zawsze, niezależnie od tego czy dla danego wyjątku istnieje czy nie istnieje blok **catch**

Blok try ... catch

Schemat: do obsługi wyjątków służy blok **try ... catch**.

```
1  try {
2    blok instrukcji mogący spowodować wyjątek
3  }
4  catch (Typ_Wyjatku_1 identyfikator_Wyjatku_1) {
5    instrukcja obsług1 wyjątku_1
6  }
7  catch (Typ_Wyjatku_2 identyfikator_Wyjatku_2) {
8    instrukcja obsług1 wyjątku_2
9  }
10 catch (Typ_Wyjatku_N identyfikator_Wyjatku_N) {
11 instrukcja obsług1 wyjątku_N
12 }
13 [finally instrukcja_obsługi wyjątku]
```

Po try następuje blok instrukcji mogących spowodować wyjątek. Jeżeli podczas ich wykonywania zostanie on wygenerowany, wykonywanie zostanie przerwane, a sterowanie przekazane do bloku **catch**. Tu sprawdzane jest, czy któraś z tych instrukcji odpowiada wygenerowanemu wyjątkowi. Jeśli tak, wykonany zostanie kod po niej występujący, jeżeli nie to kod z bloku **finally** (jeżeli taki istnieje). Instrukcje z bloku **finally** wykonywane są zawsze.

Przykład – dzielenie przez zero

```
1  class dzielenie_przez_zero
2  {
3      public static void main(String args[])
4      {
5          int dzielnik = 0;
6          int dzielna = 100;
7          int iloraz;
8          try
9          {
10             iloraz := dzielna / dzielnik;
11         }
12         catch
13         {
14          (ArithmeticException e) System.out.println("Niedozwolone dzielenie przez
15         }
16     }
17 }
```

Klasa Throwable

Nadrzędną klasą reprezentującą wyjątki jest klasa **Throwable**. Każda klasa występująca w bloku **try ... catch** jest jedną z podklas klasy **Throwable**. Ma ona dwie bezpośrednie podklasy:

- **Exception** – używa się jej z wyjątkami generowanymi i przechwytywanymi przez system.
- **Error** – określa wyjątki, które w normalnej sytuacji nie powinny być przechwytywane (są to bardzo poważne błędy).

Szczególną podklasą klasy **Exception** jest klasa **RunTimeException** dla wyjątków spowodowanych przez system czasu przebiegu Javy, są one generowane automatycznie.

Zgłaszanie wyjątków – instrukcja Throw

Oprócz bloku **try...catch** Java dostarcza jescze inny sposóbe zgłaszania wyjątków. Jest to instrukcja **throw**. Jest szczególnie przydatna w przypadku wyjątków, których nie da się obsłużyć, ich obsługa nie ma sensu, lub po prostu ich wystąpienie może spowodować nieprawidłową pracę w dalszych etapach programu. Jej składnia :

```
1  typ nazwa_metody() throws typ_wyjątku1, typ_wyjątku2, ...
2  {
3                kod...
4                throw new typ_wyjatku_1();
5                throw new typ_wyjatku_2();
6                kod
7  }
```

Throws

W nagłówku metody musimy zadeklarować, korzystając z instrukcji **throws**, listę wyjątków jakie dana metoda może wygenerować. Później w kodzie danej metody można je zgłosić (zazwyczaj np w instrukcji if) korzystając z instrukcji **throw**. Zgłaszać w ten sposób można wszystkie rodzaje wyjątków, czyli obiekty klas, które pośrednio, lub bezpośrednio dziedziczą z klasy Throwable.

- Kompilator wymaga deklarowania wyjątków reprezentowanych przez klasę Exception oraz większość jej podklas.
- wyjątki klasy **Error** i **RunTimeException** oraz wszystkie jej podklasy nie wymagają deklarowania.

Throw – przyklad

Program dzielący liczbę 1000 przez liczbę podaną przez użytkownika.

```
1  import java.util.Scanner;
2
3  public class Main {
4      public static void main(String[] agrs) throws ArithmeticException
5      {
6          double liczba_1 = 1000;
7          double liczba_2;
8          double iloraz;
9          Scanner dzielnik = new Scanner(System.in);
10         System.out.print("Podaj dzielnik: ");
11         liczba_2 = dzielnik.nextDouble();
12         if(liczba_2 == 0)
13            throw new ArithmeticException("Nie mozna dzielić przez 0");
14         else
15         {
16             iloraz = liczba_1/liczba_2;
17             System.out.println("wynik dzielenia: "+ iloraz);
18         }
19      }
20 }
```