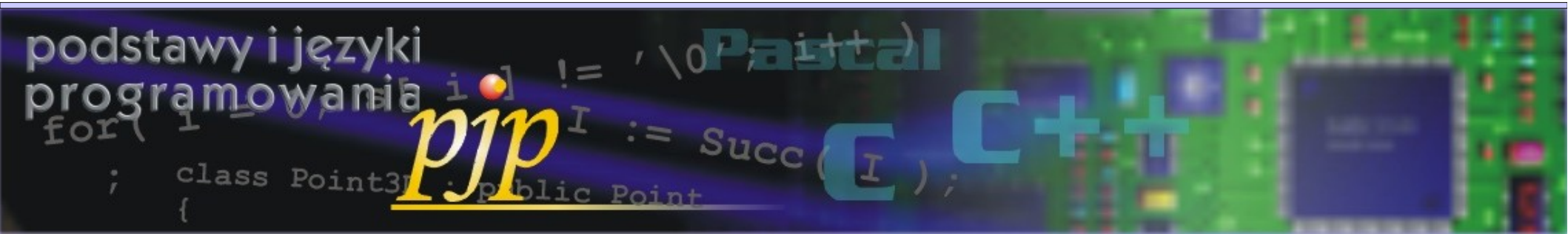
Projektowanie i programowanie obiektowe

Roman Simiński
roman.siminski@us.edu.pl
roman@siminskionline.pl
programowanie.siminskionline.pl

JAVA: Programowanie wielowątkowe
Synchronizacja



Problem - przykład, klasa MessageSender

```
class MessageSender
{
    public void Send( String msg )
    {
        connectionID = getConnectionID();
        System.out.printf( "\n%02d : wysyła: %s", connectionID, msg );
        // Tutaj wysłanie wiadomości msg z wykorzystaniem połączenia connectionID
        System.out.printf( "\n%02d : wysłano: %s", connectionID, msg );
    }

    int getConnectionID()
    {
       return gen.nextInt( 256 );
    }

    private int connectionID = 0;
    private Random gen = new Random();
}
```

A operacja powinna być atomowa

Problem - przykład, klasa MessageSender

```
class MessageSender
{
    public void Send( String msg )
    {
        connectionID = getConnectionID();
        System.out.printf( "\n%02d : wysyła: %s", connectionID, msg );
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println( "Przerwany watek" );
        }
        System.out.printf( "\n%02d : wysłano: %s", connectionID, msg );
    }

    int getConnectionID()
    {
       return gen.nextInt( 256 );
    }

    private int connectionID = 0;
    private Random gen = new Random();
}
```

„Symulacja" wysyłania wiadomości dla potrzeb demonstracji

Problem - przykład, wątek używający obiektu klasy MessageSender

```
class SendMessageThread extends Thread
{
    private MessageSender sender;
    private String msg;

    SendMessageThread( MessageSender sender, String msg )
    {
        this.sender = sender;
        this.msg = msg;
    }

    public void run()
    {
        sender.Send( msg );
    }
}
```

Wątek wysyła komunikat i kończy działanie

Problem - przykład, wysyłanie informacji w osobnych wątkach

```
MessageSender snd = new MessageSender();

SendMessageThread s1 = new SendMessageThread( snd, "Pierwsza" );
SendMessageThread s2 = new SendMessageThread( snd, "Druga" );


s1.start();
s2.start();

while( s1.isAlive() || s2.isAlive() )
{
    System.out.println( "Trwa wysyłanie wiadomości..." );
    // ...
}
System.out.println( "Wiadomości wysłane..." );
```

```
Trwa wysyłanie wiadomości...
97 : wysyła: Pierwsza
97 : wysyła: Druga
Trwa wysyłanie wiadomości...
97 : wysłano: Pierwsza
97 : wysłano: Druga
Wiadomości wysłane...
```

Synchronizacja

- Fragment kodu wykonywany w wątku może być wielokrotnie przerywany na rzecz innych wątków.
- Aby zapewnić nieprzerwane wykonanie pewnego fragmentu kodu, można użyć słowa kluczowego ***synchronized***.
- Słowo kluczowe *synchronized* zapewnia że:
  - Tylko jeden wątek w danym czasie może wykonywać wybrany dragment kodu.
  - Wątek „wchodzący” do synchronizowanego fragmentu kodu zastanie dane w stanie stabilnym – wszystkie modyfikacje zostały dokończone.
- Każdy egzemplarz klasy *Object* i jej podklas posiada *monitor* (ang. *lock*), który ogranicza dostęp do obiektu. Blokowanie obiektów jest sterowane słowem kluczowym *synchronized*.

Synchronizacja metody

- Typowe jest wykorzystanie słowa kluczowego *synchronized* dla metod. Zapewnia to, że w danym momencie tylko jeden wątek może wejść do wykonania takiej metody.
- Inne wątki korzystające z metody muszą czekać aż aktualny wątek zakończy wykonanie synchronizowanej metody.

```
Trwa wysyłanie wiadomości...
58 : wysyła: Pierwsza
Trwa wysyłanie wiadomości...
58 : wysłano: Pierwsza
200 : wysyła: Druga
Trwa wysyłanie wiadomości...
200 : wysłano: Druga
Wiadomości wysłane...
```

```
class MessageSender
{
    public synchronized void Send( String msg )
    {
        connectionID = getConnectionID();
        System.out.printf( "\n%02d : wysyła: %s", connectionID, msg );

        try { Thread.sleep(1000); }
        catch (Exception e) { System.out.println( "Przerwany watek" ); }

        System.out.printf( "\n%02d : wysłano: %s", connectionID, msg );
    }
    ...
}
```

Synchronizacja fragmentu kodu

- Słowo kluczowe synchronized może być używane również do zabezpieczenia wybranego fragmentu kodu.
- Wymaga to użycia „klucza” synchronizacji, którym może być obiekt albo łańcuch znaków.

```
Trwa wysyłanie wiadomości...
48 : wysyła: Druga
Trwa wysyłanie wiadomości...
48 : wysłano: Druga
00 : wysyła: Pierwsza
Trwa wysyłanie wiadomości...
00 : wysłano: Pierwsza
Wiadomości wysłane...
```

```
class MessageSender
{
    public void Send( String msg )
    {
        synchronized( this )
        {
            connectionID = getConnectionID();
            System.out.printf( "\n%02d : wysyła: %s", connectionID, msg );

            try { Thread.sleep(1000); }
            catch (Exception e) { System.out.println( "Przerwany watek" ); }

            System.out.printf( "\n%02d : wysłano: %s", connectionID, msg );
        }
    }
    ...
}
```

Synchronizacja fragmentu kodu

- Czasem nie można albo niecelowe wydaje się synchronizowanie kodu w zwykłej klasie, można przenieść synchronizację do klasy wątku.

```
class SendMessageThread extends Thread
{
    private MessageSender sender;
    private String msg;
    SendMessageThread( MessageSender sender, String msg )
    {
        this.sender = sender;
        this.msg = msg;
    }
    public void run()
    {
        synchronized( sender )
        {
            sender.Send( msg );
        }
    }
}
```

```
Trwa wysyłanie wiadomości...
123 : wysyła: Pierwsza
Trwa wysyłanie wiadomości...
123 : wysłano: Pierwsza
208 : wysyła: Druga
Trwa wysyłanie wiadomości...
208 : wysłano: Druga
Wiadomości wysłane...
```

Słowo kluczowe volatile

- Słowo kluczowe ***volatile*** każe kompilatorowi unikać jakiejkolwiek optymalizacji dla zmiennej zdefiniowanej z tym słowem.
- Dodatkowo taka zmienna zawsze powinna być odczytywana i zapisywana z przeznaczonej dla tej zmiennej lokalizacji, wartość zmiennej nie powinna być buforowana w rejestrach.

Zatrzymywanie wątku

- Istnieje metoda *stop*() powodująca natychmiastowe zatrzymanie wątku. Metoda ta jest wycofana (istnieje ale użycie jej *nie jest zalecane*).
- Wywołując tę metodę nie wiemy w jakim stanie jest zatrzymywany wątek, gdy wykonuje jakieś krytyczne operacje może pozostawić obiekt w nieprawidłowym stanie.
- Należy dążyć do zapewnienia programowego zakończenia pracy wątku poprzez odpowiednią organizację metody *run*() .

Zatrzymywanie wątku

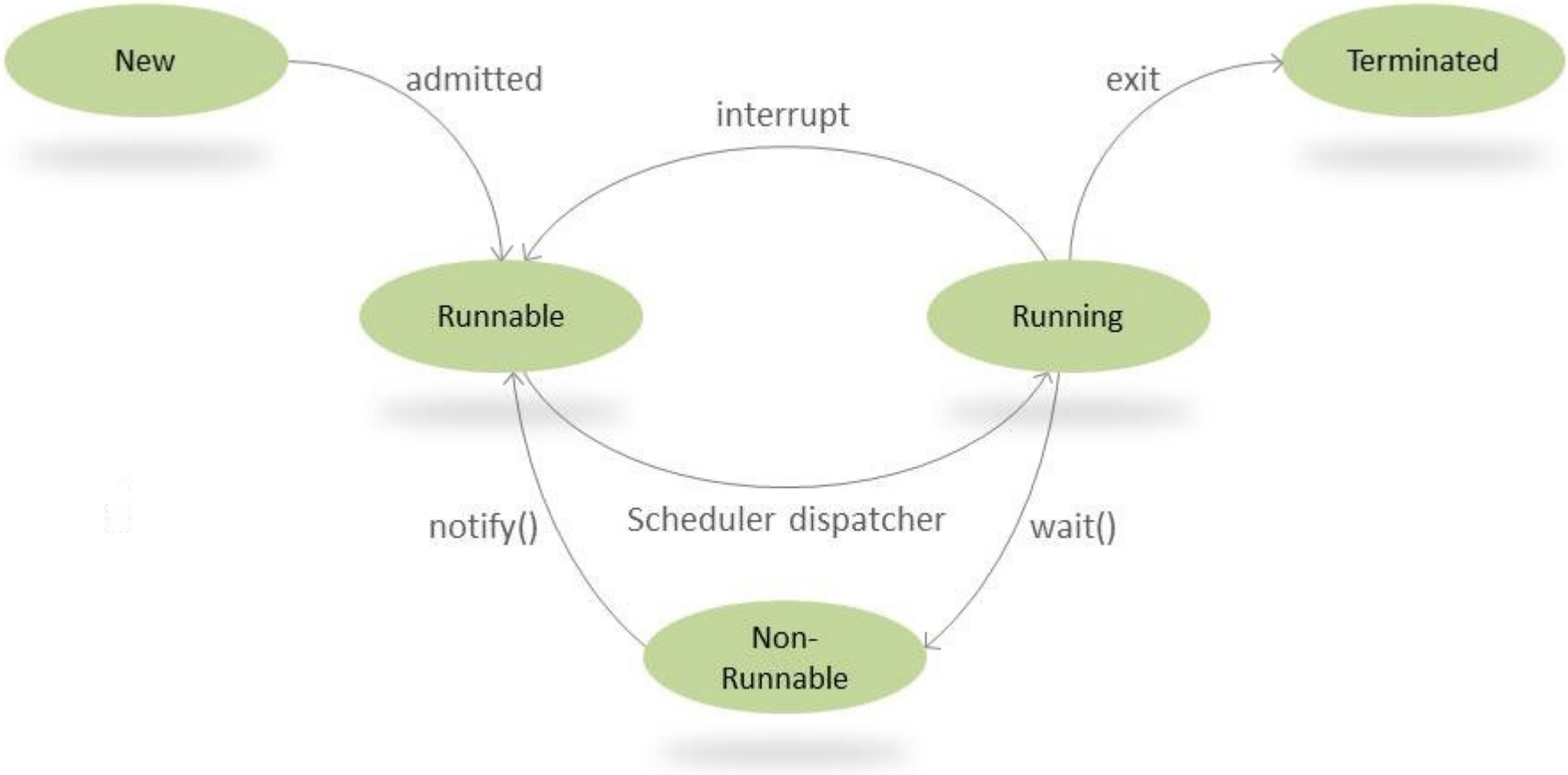
```
class MyRunnable implements Runnable
{
    private volatile boolean stopped = false;
    public synchronized void doStop()
    {
        stopped = true;
    }

    private synchronized boolean isStopped()
    {
        return stopped;
    }

    @Override
    public void run()
    {
        while( !isStopped() )
        {
            System.out.println( "Wątek działa" );

            try {  Thread.sleep(1L * 1000L); }
            catch (InterruptedException e) { e.printStackTrace(); }
        }
    }
}
```

Stany wątku



Źródło: https://www.baeldung.com/java-wait-notify

Współdziałanie wątków - przykład

```
class Message
{
    private String msg;

    public Message( String msg )
    {
        this.msg = msg;
    }

    public String getMsg()
    {
        return msg;
    }

    public void setMsg( String msg )
    {
        this.msg = msg;
    }
}
```

Współdziałanie wątków - przykład

```
class MessageReciver implements Runnable
{
    private Message msg;

    public MessageReciver( Message msg ) { this.msg = msg; }

    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        synchronized( msg )
        {
            try
            {
                System.out.printf( "[%s] oczekiwanie na komunikat, czas:%d\n",
                                   name, System.currentTimeMillis());
                msg.wait();
            }
            catch(InterruptedException e){ e.printStackTrace(); }
            System.out.printf( "[%s] komunikat otrzymany, czas:%d\n",
                               name, System.currentTimeMillis());
            System.out.printf( "[%s] otrzymal: <%s>\n", name, msg.getMsg() );
        }
    }
}
```

Współdziałanie wątków - przykład

```
class MessageSender implements Runnable
{
    private Message msg;

    public MessageSender( Message msg ) { this.msg = msg; }

    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.printf( "[%s] uruchomiony\n", name );
        try
        {
            Thread.sleep( 1000 );
            synchronized( msg )
            {
                msg.setMsg( "Treść komunikatu" );
                msg.notifyAll();
            }
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

Współdziałanie wątków - przykład

```
public static void main( String[] args )
{
    Message msg = new Message( "" );

    MessageReciver first = new MessageReciver( msg );
    new Thread( first, "Pierwszy odbiorca" ).start();

    MessageReciver second = new MessageReciver(msg);
    new Thread( second, "Drugi odbiorca" ).start();

    MessageSender sender = new MessageSender(msg);
    new Thread( sender, "Nadawca" ).start();

    System.out.println( "[Main] Nadawanie uruchomione" );
}
```

Metody wait, notify and notifyAll

Metody te może można wywołać z wątku dla każdego obiektu posiadającego *monitor*, w przeciwnym przypadku zostanie zgłoszony wyjątek *java.lang.IllegalMonitorStateException*.

Metoda *wait* może być wywołana z bloku synchronizowanego (słowo kluczowe *synchronized*), funkcja ta zwalnia blokadę na monitorze.

- *wait*() : przejście w stan oczekiwania, wątek oczekuje na wywołanie przez inny wątek metody *notify*() lub *notyfAll*().
- *wait*(*long timeout*) : przejście w stan oczekiwania, po upłynięciu czasu timeout wątek będzie uaktywniony automatycznie. Przed osiągnięciem określonego czasu wątek może zostać uaktywniony wywołaniem *notify*() or *notifyAll*().
- *wait*(*long timeout*, *int nanos*) : działanie jak wyżej, inna sygnatura, pozwala na precyzyjniejsze określenie czasu w nanosekundach: 1000000**timeout* + *nanos*.