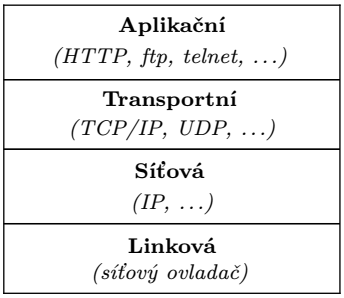
Obsah

1 Úvod do sítí

Počítače připojené do Internetu komunikují s ostatními pomocí protokolů TCP/IP. Jestliže programujeme síťové aplikace v Javě, jedná se většinou o nejvyšší vrstvu 4úrovňového modelu – vrstvu *aplikační*.



Obrázek 1: *Čtyřúrovňový model protokolů TCP/IP*

Rozhraní dvou nejvyšších úrovní modelu (aplikační a transportní vrstvy) řeší balík tříd `java.net`. Pokud chceme programovat síťové aplikace v Javě, musíme nejprve pochopit rozdíl mezi protokoly TCP a UDP.

1.1 TCP a UDP

Pokud chtějí dvě aplikace komunikovat, musí nejdříve navázat *spojení*. Teprve po jeho navázání mohou posílat data. Protokol TCP zaručuje, že data vyslaná jednou aplikací dojdou druhé aplikaci v tom samém pořadí, v jakém byla odeslána, a zároveň spolehlivě (žádná se neztratí). K aplikacím, které vyžadují spolehlivý přenos, patří např. http, ftp nebo telnet.

Naproti tomu protokol UDP nezaručuje ani spolehlivost ani pořadí přicházejících dat. V podstatě se do sítě vyšlou pouze nezávislé *datagramy*. Nenavazuje se ani spojení mezi dvěma aplikacemi.

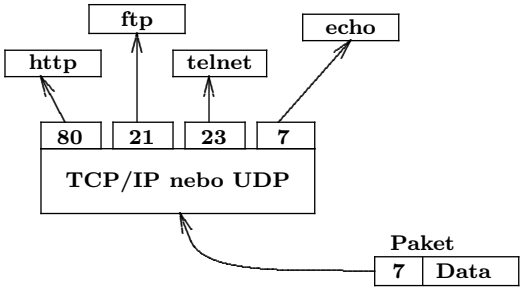
1.2 Porty

Předpokládejme, že počítač má pouze jedno fyzické spojení se sítí. Všechna data tedy musí přicházet právě tudy. Ale data mohou být určena různým aplikacím spuštěným současně na jednom počítači. Proto se pro určení cílové aplikace se používají *porty*.

Port je 16bitové číslo, které jednoznačně určuje, které aplikaci jsou data určena. Ve spojovaných službách (TCP) se navazuje spojení tak, že se vytvoří *socket* a spojí se s jedním konkrétním *číslem portu*. To proto, aby se mohlo toto číslo zaregistrovat a už žádná jiná aplikace ho nemohla používat. V nespojovaných službách je *číslo portu* přímo uloženo v *datagramu*. Žádné spojení se nenavazuje.

Číslo portu může být od 0 do 65535, ale porty 0–1023 jsou rezervovány pro tzv. *well-known porty* (všeobecně známé porty). Jedná se o čísla portů standardních síťových aplikací.

V Javě se pro komunikaci pomocí spojovaných služeb (protokolem TCP) využívají třídy `URL`, `URLConnection`, `Socket` a `ServerSocket`, pro komunikaci prostřednictvím nespojovaných služeb (protokol UDP) třídy `DatagramPacket` a `DatagramServer`.



Obrázek 2: *Rozpoznání aplikace podle čísla portu*

2 Práce s URL

Balík `java.net` obsahuje třídu `URL`. Tato třída reprezentuje adresu (*Uniform Resource Locator*) nějakého prostředku ve World Wide Web. Každé URL se skládá ze čtyř částí, přičemž pouze dvě první jsou povinné, ostatní se dosazují implicitně:

- identifikátor protokolu – `ftp, http, file, news ...`
- jméno stroje – *DNS* jméno
- číslo portu – standardní číslo *http* serveru je 80
- jméno souboru – standardně `index.html`

2.1 Vytvoření URL

Třída `URL` poskytuje několik možností vytvoření URL. Nejjednodušší cesta pro vytvoření URL objektu je ze stringu reprezentujícího pro člověka srozumitelnou formu URL adresy. V Java programu tedy vytvoříme objekt takto:

```
URL zcu = new URL( "http://www.zcu.cz" );
```

Nebo jeho jednotlivé složky odděleně:

```
URL zcu = new URL( "http", "www.zcu.cz", "/index.html" );
URL zcu = new URL( "http", "www.zcu.cz", 80, "/index.html" );
```

Podobně můžeme použít relativní adresu:

```
URL zcu_cz = new URL( zcu, "/index-cz.html" );
```

Vytvořením objektu třídu `URL` jsme vyrobili pouze lokální objekt, pro navázání spojení musíme tento objekt spojit s URL, které reprezentuje (viz. kapitola 2.4).

Všechny tvary konstruktorů mohou při chybném zadání některého parametru vyvolat výjimku `MalformedURLException`, proto je nutné vytvářet objekty `URL` v bloku `try/catch`.

```
try {
    URL myURL = new URL(. . .);
} catch ( MalformedURLException e ) {
    . . .
    // reakce na výjimku
    . . .
}
```

2.2 Získání atributů URL

Třída `URL` nabízí několik metod pro dotazování se na aktuální stav URL:

- `String getProtocol()` – vrací řetězec obsahující URL protokol
- `String getHost()` – vrací URL adresu hostitele
- `int getPort()` – vrací URL číslo portu
- `String getFile()` – vrací jméno souboru

2.3 Čtení přímo z URL

Po úspěšném vytvoření objektu třídy `URL` můžeme z tohoto URL číst data. K tomu slouží metoda `openStream()`[1], která vrací standardní *stream* pro čtení `java.io.InputStream`, a proto pak můžeme použít její metody (např. `readLine()`). Následující program ukazuje, jak jednoduché je čtení z URL:

```
// OpenStreamTest.java
// Cteni primo z URL

import java.net.*;
import java.io.*;

class OpenStreamTest {
  public static void main( String[] args )
  {
    try {
      URL katka = new URL( "http://www.lobzy.cz/" );
      DataInputStream dis = new DataInputStream( katka.openStream() );
      String inputLine;

      while ( ( inputLine = dis.readLine() ) != null ) {
      System.out.println( inputLine );
      }
      dis.close();
    } catch ( MalformedURLException me ) {
      System.out.println( "MalformedURLException: " + me );
    } catch ( IOException ioe ) {
      System.out.println( "IOException: " + ioe );
    }
  }
}
```

Program otevře vstupní *stream* a zobrazí jeho obsah na displeji. Po spuštění bychom měli vidět zdrojový soubor html dokumentu na zadané adrese.

[1] Tato metoda je jenom zkrácené volání `openConnection().getInputStream()`.

2.4 Připojení k URL

Jakmile jsme úspěšně vytvořili objekt třídy `URL`, můžeme zavolat metodu `openConnection()` k vytvoření spojení. Tato metoda vytvoří objekt třídy `URLConnection`, inicializuje ho a připojí se na toto URL po síti. Jestliže se některá část připojení nepovede, metoda `openConnection()` vyvolá výjimku `IOException`[2].

```
try {
    URL myURL = new URL(. . .);
    myURL.openConnection();
} catch ( MalformedURLException e ) {   // new URL chybne
    . . .
} catch ( IOException e ) {             // openConnection() chybne
    . . .
}
```

Když jsme se teď úspěšně připojili k URL, můžeme použít objekt třídy `URLConnection` mj. ke čtení a zápisu z/do spojení.

2.5 Čtení z `URLConnection`

Třída `URLConnection` obsahuje mnoho metod umožňujících komunikaci s URL přes síť. Tato třída se soustředí na protokol http, mnoho jejich metod funguje pouze pro http spojení.

Následující program dělá to samé, co jeho předchůdce čtoucí přímo z URL (viz. kapitola 2.3). Tento program však explicitně otevírá spojení s URL[3].

```
// ConnectionTest.java
// Vyuziva tridu URLConnection pro cteni URL

import java.net.*;
import java.io.*;

class ConnectionTest {
  public static void main( String[] args )
  {
    try {
      URL katka = new URL("http://www.lobzy.cz/");
      URLConnection katkaConnection = katka.openConnection();
      DataInputStream dis = new DataInputStream(katkaConnection.getInputStream());
      String inputLine;

      while ( ( inputLine = dis.readLine() ) != null ) {
      System.out.println( inputLine );
      }
      dis.close();
    } catch ( MalformedURLException me ) {
      System.out.println( "MalformedURLException: " + me);
    } catch ( IOException ioe ) {
      System.out.println( "IOException: " + ioe );
    }
  }
}
```

[2] Výjimku `IOException` vrací všechny metody pracující se *streamy*.

[3] Čtení z URL pomocí třídy `URLConnection` je výhodnější, protože objekt této třídy můžeme současně použít v jiném tasku třeba pro zápis.

2.6 Zápis do `URLConnection`

Mnoho HTML dokumentů obsahuje formuláře, tj. textové položky a jiné GUI objekty, kterými se zadávají data pro server. Po zadání prohlížeč předá data serveru. Na druhém konci sítě speciální cgi-bin programy data zpracují a pošlou zpět odpověď, většinou ve tvaru HTML dokumentu.

Program v Javě mohou interaktivně pracovat s cgi-bin programy na straně serveru. Může tak učinit následovně:

1. Vytvořit URL (viz. kapitola 2.1).
2. Otevřít spojení (connection) se serverem (viz. kapitola 2.4).
3. Vzít z tohoto spojení výstupní *stream*, který je připojen ke standardnímu vstupnímu *streamu* cgi-bin programu na serveru.

   ```
   PrintStream outStream = new PrintStream( connection.getOutputStream() );
   ```

4. Zapsat do *streamu* data standardními metodami, např. `outStream.println()`.
5. Zavřít *stream* (`outStream.close()`).

3 Java Sockety

Třídy `URL` a `URLConnection` jsou vytvořeny na poměrně vysokém stupni abstrakce a jsou použitelné pouze pro specifický případ komunikace s WWW serverem. Některé jiné aplikace ale mohou chtít využívat i nižších síťových úrovní např. aplikace typu klient/server.

V těchto aplikacích server poskytuje nějaké služby, např. vyhledávání v databázi, a klient posílá dotazy a čeká na odpovědi. Komunikace mezi nimi musí být spolehlivá (žádná data se nesmí ztratit a musí dojít ve správném pořadí).

Server většinou poslouchá na specifickém portu a čeká na požadavky od klientů o navázání spojení. Při požadavku na vytvoření spojení si klient vytvoří lokální číslo portu a spojí ho se *socketem*. Klient potom komunikuje se serverem tak, že čte a zapisuje do *socketu*.

A podobně také server při příchodu požadavku vytvoří nové číslo portu (protože musí poslouchat ostatní klienty na stále stejném čísle portu) a spojí ho se svým *socketem*. Potom i server komunikuje s klientem pomocí čtení a zápisu z/do *socketu*. *Socket* je tedy jeden konec dvoustranného komunikačního spojení mezi dvěma programy běžícími na síti.

Balík `java.net` nabízí dvě třídy implementující klientskou resp. serverovskou stranu spojení. Třída `Socket` implementuje klientskou stranu a třída `SocketServer` serverovskou.

3.1 Klient – třída `Socket`

Tato třída implementuje klientskou část socketů. Jednoduchá syntaxe ji předurčuje pro vytváření klientských programů komunikujících se standardními síťovými službami např. echo, telnet, ftp apod.

Otevření socketu

Socket vytvoříme[4] zavoláním konstruktoru třídy, přičemž můžeme zadat jméno nebo IP adresu stroje a samozřejmě číslo portu, se kterým chceme socket spojit.

```
Socket mySocket = new Socket( "katka", 7 );
```

Po skončení práce je třeba socket uzavřít.

```
mySocket.close();
```

Získání parametrů socketu

Třída `Socket` implementuje několik metod, kterými lze získat informace o socketu:

- `InetAddress getInetAdress()` – IP adresa stroje, se kterou je spojen socket
- `int getLocalPort()` – lokální číslo portu, se kterým je spojen socket
- `int getPort()` – vzdálené číslo portu (na protilehlém stroji)

Čtení a zápis z/do socketu

Pro čtení a zápis z/do socketu (a tím pro komunikaci se serverem) jsou ve třídě `Socket` dvě metody:

- `InputStream getInputStream()` – vrátí vstupní *stream* pro čtení ze socketu
- `OutputStream getOutputStream()` – vrátí výstupní *stream* pro zápis do socketu

Standardní klientský program potom může vypadat následovně:

1. Otevření socketu (viz. kapitola 3.1).
2. Otevření vstupního a výstupního *streamu*.
3. Čtení a zápis z/do *streamu* podle implementovaného protokolu.
4. Uzavření *streamů*.
5. Uzavření socketu[5].

[4] Vytvoření socketu je opět třeba provádět v bloku `try/catch`, protože konstruktor vyvolává výjimky `IOException` a `UnknownHostException`.

[5] Pořadí uzavření je třeba dodržet, nelze uzavřít socket s otevřeným *streamem*.

3.2 Server – třída `ServerSocket`

Vytvoření serveru

Program serveru musí čekat na požadavky klientů na specifickém portu. Jeho číslo musíme zadat při jeho vytváření[6]:

```
try {
  ServerSocket sSocket = new ServerSocket( 5555 );
} catch { IOException e ) {
  // nemůže se připojit na port 5555
}
```

A samozřejmě musíme socket na konci programu uzavřít (jako poslední):

```
sSocket.close();
```

Navázání spojení

Třída `ServerSocket` poskytuje metodu `accept()`, která vrací odkaz na třídu `Socket`. Tato metoda zablokuje program serveru, dokud nepřijde požadavek od klienta o navázání spojení:

```
Socket clientSocket;
try {
  clientSocket = sSocket.accept();
catch ( IOException e ) {
  // accept skončil chybou
}
```

Pokud `accept()` skončí, vrátí socket, který je spojen s nějakým volným lokálním číslem portu. Server pak komunikuje s klientem pomocí tohoto socketu (viz. kapitola 3.1). Pak může čekat na tom samém čísle portu na ostatní klienty.

Komunikace s klienty potom probíhá pomocí metod třídy `Socket`.

Asynchronní obsluha klientů

Jak je vidět, metoda `accept()` by zablokovala celý serverový program po celou dobu čekání na požadavek od klienta. Protože ale požadavky mohou (a pravděpodobně také budou) přicházet asynchronně (nezávisle na sobě), je třeba zařídit, aby server reagoval na všechny také nezávisle. Toto chování lze zajistit použitím *threadů* v našem programu serveru – jeden *thread*, jeden klient:

```
while ( true ) {
  navázání spojení
  vytvoření threadu pro komunikaci s klientem
}
```

Thread potom čte a zapisuje do otevřeného socketu nezávisle na ostatních.

[6]Konstruktor třídy `ServerSocket` opět vyvolává výjimku `IOException`.

Tento program komunikuje se standardní službou echo:

```
// EchoTest.java
// Otestovani standardni sluzby echo (port 7)

import java.io.*;
import java.net.*;

public class EchoTest {
  public static void main( String[] args )
  {
    Socket echoSocket = null;
    DataOutputStream os = null;
    DataInputStream  is = null;
    DataInputStream  stdIn = new DataInputStream( System.in );

    try {
      echoSocket = new Socket( InetAddress.getLocalHost(), 7 );
      os = new DataOutputStream( echoSocket.getOutputStream() );
      is = new DataInputStream( echoSocket.getInputStream() );
    } catch ( UnknownHostException e ) {
      System.err.println( "Don't know about host: " +
                    echoSocket.getInetAddress() );
    } catch ( IOException e ) {
      System.err.println( "Couldn't get I/O for the connection to: " +
                    echoSocket.getInetAddress() );
    }

    if ( echoSocket != null && os != null && is != null ) {
      try {
      String userInput;

      while ( ( userInput = stdIn.readLine() ) != null ) {
        os.writeBytes( userInput );
        os.writeByte( '\n' );
        System.out.println( "echo: " + is.readLine() );
      }
      os.close();
      is.close();
      echoSocket.close();
      } catch ( IOException e ) {
      System.err.println( "I/O failed on the connection to: " +
                     echoSocket.getInetAddress() );
      }
    }
  }
}
```

4 Java Datagramy

V předcházejících kapitolách mělo zásadní význam pořadí, v jakém data přicházejí: když čteme z URL, data musí být přijata ve stejném pořadí, v jakém byla odeslána, jinak dostaneme nesmyslný html dokument, poškozený soubor či jinou nesmyslnou informaci.

Mechanismus, který zajišťuje správné pořadí dat typicky používá pakety. Toto zajištění pořadí však není zadarmo. Vyžaduje speciální verifikaci pořadí paketů, reakci na ztrátu nebo poškození paketu apod. Pro mnohé aplikace má bezpečná komunikace poskytovaná protokolem TCP zásadní význam. Pro některé však nehraje důležitou roli nebo je dokonce nežádoucí. Např. příkaz `ping` testuje komunikaci mezi dvěma počítači v síti, ale ve skutečnosti chce vědět o ztracených paketech, aby zjistil, jak dobré je spojení.

Služby, které nepotřebují nebo nechtějí mít perfektní a spolehlivý kanál, používají ke komuni-

kaci *datagramy*. Klient i server posílá zcela nezávislé pakety bez existence oboustranného kanálu. Datagramový paket obsahuje celou adresu zdroje i cíle cesty a jeho doručení ani nepoškozenost nejsou zaručeny.

Balík `java.net` obsahuje dvě třídy používající datagramový model komunikace: `DatagramPacket` a `DatagramSocket`. Tyto dvě třídy implementují systémově nezávislou komunikaci protokolem UDP.

4.1 Vytvoření paketu

Dříve než pošleme nějaký paket po síti, musíme vytvořit jeho obsah. Pro to nám slouží třída `DatagramPacket`. Objekty této třídy slouží jako „vyrovnávací paměť“ na jeden paket, nezáleží na tom, jestli přijímaný nebo vysílaný. V konstruktoru třídy ale musíme rozlišit, o jaký druh paketu se bude jednat.

Pokud chceme paket přijmout, vymezíme pouze část paměti (nezajímá nás, z jakého portu bude paket přijat):

```
byte[] buff = new byte[256];
DatagramPacket packet = new DatagramPacket( buff, 256 );
```

Když má být paket naopak odeslán, musíme už při jeho vytváření specifikovat, na jakou adresu a jaký port má být poslán (každý paket nese tuto informaci samostatně). Tím zároveň specifikujeme, jaký proces na protilehlé straně si tento paket odebere ze sítě:

```
byte[] buff = new byte[256];
DatagramPacket packet = new DatagramPacket( buff, 256, InetAddress.getLocalHost(), 7 );
```

Tímto postupem pouze vymezíme lokální paměť pro příjem nebo vysílání, popř. ji naplníme odpovídajícími hodnotami, **nic se do sítě nevysílá!**

4.2 Získání informací z paketu

Třída `DatagramPacket` obsahuje několik metod, jimiž se můžeme dotazovat na stav přijatého nebo vysílaného paketu:

- `InetAddress getAddress()` – vrátí adresu stroje, kam má být paket poslán, nebo odkud byl přijat
- `int getPort()` – vrátí číslo portu vzdáleného stroje, kam má být paket poslán, nebo odkud byl přijat
- `int getLength()` – vrátí délku přijatého paketu, nebo paketu připraveného k odeslání
- `byte[] getData()` – vrátí obsah přijatého paketu, nebo paketu připraveného k odeslání

4.3 Posílání a příjem paketů

Jak bylo řečeno výše, třída `DatagramPacket` slouží pouze jako vyrovnávací paměť pro příjem nebo vyslání jednoho paketu. Pro vlastní síťovou komunikaci obsahuje balík `java.net` další třídu `DatagramSocket`.

Tato třída se chová podobně jako třída `Socket`, tzn., že vytvářený socket musíme nejdříve spojit s nějakým lokálním číslem portu. Buď můžeme nechat systém, aby nám poskytl první volné:

```
DatagramSocket socket = new DatagramSocket();
```

Nebo můžeme sami specifikovat, jaké číslo chceme:

```
DatagramSocket socket = new DatagramSocket( 7 );
```

Samotná komunikace je pak velmi jednoduchá, třída poskytuje pouze dvě metody:

- `void receive( DatagramPacket p )` – čeká na příjem dat ze socketu. Po návratu z této metody je objekt třídy `DatagramPacket` nejen naplněn přijatými daty, ale obsahuje také IP adresu a port odesílatele.

  Tato metoda zablokuje provádění programu, dokud se příjem neuskuteční. Položka `length` objektu `p` obsahuje délku přijatých dat. Jestliže jsou data delší než je délka vyrovnávací paměti, paket je **zkrácen**.
- `void send( DatagramPacket p )` – pošle paket do socketu. Paket již obsahuje informaci, na jakou adresu a jaký port má být poslán (viz. kapitola 4.1).

Na konci programu musíme samozřejmě socket zase zavřít:

```
socket.close();
```

Následující příklad ukazuje použití datagramů pro komunikaci se serverem:

```
import java.io.*;
import java.net.*;
import java.awt.*;

public class EchoTest extends Frame {
  TextField enter;
  TextArea display;
  Panel enterPanel;
  Label enterLabel;

  DatagramPacket sendPacket, receivePacket;
  DatagramSocket sendSocket, receiveSocket;

  public EchoTest()
  {
    super( "EchoTest" );
    enterPanel = new Panel();
    enterLabel = new Label( "Enter message:" );
    enter = new TextField( 20 );
    enterPanel.add( enterLabel );
    enterPanel.add( enter );
    add( "North", enterPanel );
    display = new TextArea( 20, 10 );
    add( "Center", display );
    resize( 400, 300 );
    show();

    try {
      sendSocket = new DatagramSocket();
      receiveSocket = new DatagramSocket( 7 );
```

```
    }
    catch ( SocketException e ) {
      e.printStackTrace();
      System.exit( 1 );
    }
  }

  public void waitForPackets()
  {
    while ( true ) {
      try {
        byte array[] = new byte[ 100 ];
        receivePacket = new DatagramPacket( array, array.length );
        receiveSocket.receive( receivePacket );
        byte data[] = receivePacket.getData();
        String received = new String( data, 0 );
        display.appendText( "echo:" + received );
      }
      catch ( IOException e ) {
        display.appendText( e.toString() + "\n" );
        e.printStackTrace();
      }
    }
  }

  public boolean handleEvent( Event e )
  {
    if ( e.id == Event.WINDOW_DESTROY ) {
      hide();
      dispose();
      System.exit( 0 );
    }
    return super.handleEvent( e );
  }

  public boolean action( Event event, Object o )
  {
    try {
      String s = o.toString();
      byte data[] = new byte[ 100 ];
      s.getBytes( 0, s.length(), data, 0 );
      sendPacket = new DatagramPacket( data, s.length(), InetAddress.getLocalHost(), 7 );
      sendSocket.send( sendPacket );
    }
    catch ( IOException exception ) {
      display.appendText( exception.toString() + "\n" );
      exception.printStackTrace();
    }
    return true;
  }


  public static void main( String args[] )
  {
    EchoTest c = new EchoTest();

    c.waitForPackets();
  }
}
```

5 Java Thready

Thread je sekvenční posloupnost instrukcí definovaná počátkem a koncem, v každém časovém okamžiku během vykonávání *threadu* je vykonávána právě jedna instrukce. *Thread* ale není program,

nemůže běžet sám o sobě, ale běží uvnitř programu (procesu). Proces může obsahovat několik *threadů*, které běží paralelně[7].

Na samostatném *threadu* není nic zvláštního, to splňuje každý program. Pravým jádrem věci je použití několika paralelně běžících *threadů*, kdy každý vykonává něco jiného, v jednom programu. Například prohlížeč, kde posouváme stránku zatímco se natahuje obrázek a je přehráván zvukový soubor. Jiným příkladem mohou být třeba paralelně běžící algoritmy.

Thready bývají někdy nazývány *lightweight procesem*. *Thread* a proces jsou si podobné v tom, že jde o samostatný sekvenční tok řízení. *Thread* ale běží pouze v kontextu programu a může využívat pouze jeho prostředky a jeho prostředí. Má svůj vlastní kontext (zásobník, čítač instrukcí), v němž jsou prováděny jeho instrukce.

5.1 Podpora Javy pro thready

Java podporuje thready:

- na úrovni Java Runtime (interpret Javy) – jádro je multithreaded
- thready pro podporu runtime:
  - `hlavní thread interpretu Javy`
  - `Clock thread`
  - `Idle thread`
  - `Garbage Collection thread`
- na úrovni knihoven – umožňuje použití threadů v native kódu
- na úrovni jazyka – třídy `Thread`, `ThreadGroup` a interface `Runnable`

Java thready (dále jen thready) jsou implementovány pomocí třídy `Thread`, která je součástí balíku `java.lang`. Třída `Thread` implementuje systémově nezávislou definici threadu. Aktuální realizace běhu threadu je dána systémově závislou implementací. Pro většinu programátorských potřeb není důležitá a lze ji ignorovat a programovat pomocí API.

5.2 Tělo threadu

Činnost threadu (sekvenční činnost, kterou vykonává) je určena v těle threadu v metodě `run()`. V těle threadu je často používán cyklus (např. thread pro animaci zobrazuje v cyklu jednotlivé obrázky). Někdy je thread (metoda `run()`) použit pro vykonání operací náročných na čas (např. stažení a přehrání zvuku).

Tělo threadu lze vytvořit jednou z následujících možností:

1. Děděním třídy `Thread` (balík `java.lang`) a přepsáním metody `run()`. Thread pak spustíme následovně:

```
myThreadClass p = new myThreadClass();
p.start();
```

[7] *Thread* je samostatný sekvenční tok řízení (instrukcí) uvnitř programu.

2. Vytvořením objektu `Thread`, jemuž při volání `new` zadáme jako parametr objekt, který implementuje interface `Runnable` (také balík `java.lang`). Metoda `run()` tohoto objektu potom tvoří tělo threadu. Tento druh threadu se spouští následovně:

```
myRunnableClass p = new myRunnableClass();
new Thread(p).start();
```

Pro každou z možností existují dobré důvody pro její zvolení. Obecně lze říci, že pokud je naše třída odděděna od jiné třídy (častý případ appletů), měla by mít implementován interface `Runnable`.

```
import java.awt.Graphics;
import java.util.Date;
import java.applet.Applet;

public class Clock extends Applet implements Runnable {

    Thread clockThread = null;

    public void start() {
        if ( clockThread == null ) {
            clockThread = new Thread( this, "Clock" );
            clockThread.start();
        }
    }
    public void run() {
        while ( Thread.currentThread() == clockThread ) {
            repaint();
            try {
                clockThread.sleep( 1000 );
            } catch ( InterruptedException e ) {
            }
        }
    }
    public void paint( Graphics g ) {
        Date now = new Date();
        g.drawString( now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds(), 5, 10 );
    }
    public void stop() {
        clockThread = null;
    }
}
```

5.3 Stavy threadu

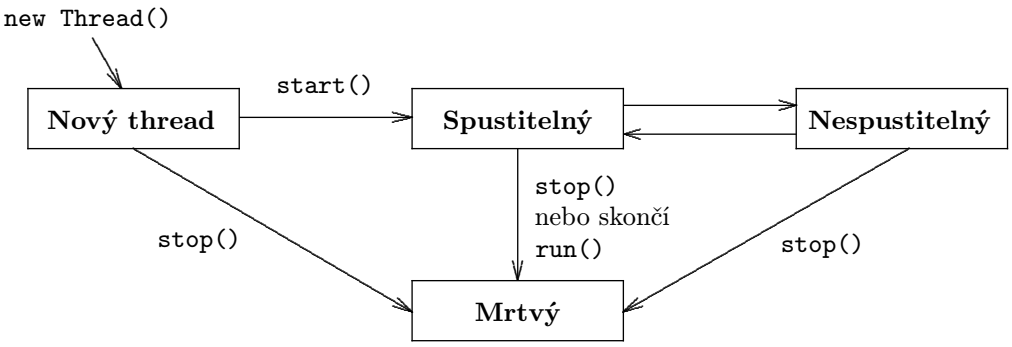
Následující diagram ilustruje různé stavy, kterými může thread procházet během svého života (v každém okamžiku se nachází právě v jednom z nich) a které metody způsobí přechod do jiného stavu.

Tento diagram není úplným diagramem konečného automatu, ale spíše přehledem hlavních stavů threadu.

Nový thread

Následující příkaz vytvoří nový thread, ale nespustí ho (proto se thread nachází ve stavu označeném „Nový thread“):

```
Thread myThread = new MyThreadClass();
```



Obrázek 3: *Stavy threadu*

Pokud se thread nachází ve stavu „Nový thread“, je to v podstatě prázdný objekt `Thread`, nebyly pro něj dosud alokovány žádné systémové prostředky. Proto, když se thread nachází v tomto stavu, lze jej pouze spustit nebo zastavit. Volání jiné metody než `start()` nebo `stop()` nemá v tomto stavu smysl a způsobí výjimku `IllegalThreadStateException`.

Spustitelný (Runnable)

Následující řádky vytvoří nový thread a spustí jej:

```
Thread myThread = new MyThreadClass();
myThread.start();
```

Metoda `start()` vytvoří systémové prostředky nezbytné pro běh threadu, naplánuje jej na běžící a zavolá jeho metodu `run()`. V tomto okamžiku se thread nachází ve stavu „Spustitelný“. Stav se nazývá „Spustitelný“ místo „Běžící“, protože thread v tomto stavu nemusí v každém okamžiku právě běžet (pokud má počítač pouze jedem procesor, není možné, aby více threadů ve stavu „Spustitelný“ běželo zároveň). Proto Java runtime systém musí implementovat plánování threadů, což umožní sdílení procesoru více thready najednou (další informace viz. kapitola 5.4).

Přesto pro většinu účelů lze prostě uvažovat stav „Spustitelný“ jako stav „Běžící“. Pokud thread běží (je ve stavu „Spustitelný“ a je aktuálním threadem) jsou příkazy v jeho metodě `run()` vykonávány sekvenčně.

Nespustitelný (Not runnable)

Thread vstoupí do stavu „Nespustitelný“, pokud nastane jedna z následujících událostí:

- někdo vyvolá jeho metodu `suspend()`
- někdo vyvolá jeho metodu `sleep()`
- thread použije svoji metodu `wait()`, aby čekal na condition proměnnou (*sdílenou proměnnou*) viz. kapitola 5.7
- thread je blokován na I/O

Například čtvrtá řádka v následující ukázce kódu

```
Thread myThread = new MyThreadClass();
myThread.start();
try {
  myThread.sleep( 10000 );
} catch ( InterruptedException e ) {
}
```

převede `myThread` do stavu „Nespustitelný“ na 10 sekund (10 000 milisekund). Během těchto 10 sekund, i když je procesor volný, `myThread` neběží. Po 10 sekundách se `myThread` přesune znova do stavu „Spustitelný“. Pokud je nyní procesor volný, `myThread` bude spuštěn.

Pro každou možnost „vstupu“ do stavu „Nespustitelný“ uvedenou v seznamu výše, je zvláštní a odlišná úniková metoda, která vrací thread do stavu „Spustitelný“. Například, pokud byl thread „uspán“ metodou `sleep()`, pak musí uplynout daný počet milisekund než thread opět přejde do stavu „Spustitelný“. Volání metody `resume()` na spící thread nemá žádný efekt.

Následující řádky udávají „Únikovou cestu“ pro každý vstup do stavu „Nespustitelný“:

- po vyvolání metody `sleep()` musí uplynout daný počet milisekund
- po vyvolání metody `suspend()` musí někdo zavolat jeho metodu `resume()`
- pokud thread čeká na condition proměnnou, musí se jí objekt, který danou proměnnou vlastní, vzdát pomocí volání metody `notify()` nebo `notifyAll()`
- pokud je thread blokován I/O operací, pak musí daná operace skončit

Mrtvý (Dead)

Thread může přejít do stavu „Mrtvý“ dvěma způsoby:

- z přirozených důvodů
- je ukončen

Thread přejde do stavu „Mrtvý“ z přirozených důvodů, když metoda `run()` normálně skončí. Například cyklus `while` v teto metodě je konečný cyklus – proběhne 100krát a pak skončí.

```
public void run() {
  int i = 0;
  while ( i < 100 ) {
    i++;
    System.out.println( "i = " + i );
  }
}
```

Thread s touto metodou `run()` přirozeně skončí, jakmile je cyklus (a tím i metoda `run()`) ukončen.

Thread lze kdykoli ukončit zavoláním jeho metody `stop()`. Tato ukázka kódu

```
Thread myThread = new MyThreadClass();
myThread.start();
try {
  Thread.currentThread().sleep( 10000 );
} catch ( InterruptedException e ) {
}
myThread.stop();
```

vytvoří a spustí `myThread`, „uspí“ aktuální thread na 10 sekund. Když je uspaný thread vzbuzen, poslední řádka ukončí `myThread`.

Metoda `stop()` zašle threadu výjimku `ThreadDeath`. Proto pokud ukončujeme thread tímto způsobem, je ukončen asynchronně.

Applety často používají metodu `stop()` k ukončení všech svých threadů, když prohlížeč, který applet spustil, chce ukončit svoji činnost.

Výjimka IllegalThreadStateException

Runtime systém vyvolá tuto výjimku, když zavoláme metodu threadu, jehož stav nedovoluje vyvolání této metody. Například je tato výjimka vyvolána, pokud zavoláte metodu `suspend()` threadu, který není ve stavu „Spustitelný“.

Metoda `isAlive()`

Třída `Thread` obsahuje metodu `isAlive()`. Metoda vrací hodnotu `true`, pokud byl thread spuštěn a nebyl zastaven. Proto pokud metoda vrátí hodnotu `false` víme, že thread je buď ve stavu „Nový thread“ nebo „Mrtvý“. Pokud metoda `isAlive()` vrátí hodnotu `true` víme, že se thread nachází ve stavu „Spustitelný“ nebo „Nespustitelný“. Stavy „Nový thread“ a „Mrtvý“ nebo „Spustitelný“ a „Nespustitelný“ nelze rozlišit.

5.4 Priorita threadu

Model threadu říká, že běží paralelně. V praxi to však obvykle není možné, proto v každém okamžiku běží pouze jeden thread a jednotlivé thready jsou přepínány a simulují tak paralelní běh. Běh několika threadů na jednom procesoru a způsob jejich přepínání se nazývá *plánování*. Java runtime systém implementuje velmi jednoduchý deterministický plánovací algoritmus založený na *statické prioritě*. Tento algoritmus „plánuje“ thready na základě jejich priority vzhledem k ostatním threadům ve stavu „Spustitelný“.

Když je vytvořen nový thread, zdědí svoji prioritu od threadu, který jej vytvořil. Prioritu threadu je pak možno kdykoliv změnit metodou `setPriority()`. Priorita threadu leží v intervalu mezi `MIN_PRIORITY` a `MAX_PRIORITY` (konstanty definované ve třídě `Thread`). Kdykoli je více threadů připraveno připraveno k vykonávání, runtime systém volí thread ve stavu „Spustitelný“ s nejvyšší prioritou. Pouze pokud tento thread skončí nebo přejde z nějakého důvodu do stavu „Nespustitelný“, je spuštěn thread s nižší prioritou. Pokud čeká více threadů se stejnou prioritou, plánovač je spouští cyklicky.

Plánovací algoritmus je také *preemptivní*. Jestliže se kdykoliv během vykonávání threadů s nižší prioritou dostane thread s vyšší prioritou do stavu „Spustitelný“, runtime systém ho okamžitě naplánuje.

5.5 Typy threadů

Java podporuje dva typy threadů:

- normální thread
- *daemon* thread

Jakýkoli thread může být *daemon* thread. *Daemon* thready obvykle poskytují služby ostatní threadům nebo objektům běžícím ve stejném procesu jako *daemon* thread. Metoda `run()` *daemon* threadu je obvykle nekonečný cyklus, který čeká na žádost o službu.

Typ threadu *daemon* je určen voláním metody `setDaemon()` s parametrem typu boolean s hodnotou `true`. Zjistit typ threadu lze pomocí metody daného threadu `isDaemon()`.

Pokud v procesu zbývají pouze *daemon* thready, interpret proces ukončí. Důvodem je to, že pokud zbyly pouze *daemon* thready, nemají komu poskytovat služby.

5.6 Skupiny threadů

Každý thread je členem *skupiny threadů*. Skupiny threadů poskytují mechanismus pro sdružování více threadů do jednoho objektu a pro manipulaci se skupinou těchto threadů místo s každým zvlášť (např. spustit nebo suspendovat všechny thready ve skupině použitím jediné metody). Skupiny threadů jsou implementovány třídou `ThreadGroup` z balíku `java.lang`.

Runtime systém thread zařadí do skupiny threadů během jeho vytvoření. Při jeho vytváření můžeme nechat runtime systém, aby thread zařadil do některé standardní skupiny, nebo výslovně nastavit jeho skupinu. Thread je po vytvoření permanentním členem skupiny, do které byl zařazen, a nelze ho již přeřadit do jiné.

Standardní skupina threadů

Třída `Thread` podporuje několik konstruktorů, které nepotřebují zadávat argument určující skupinu. Pokud vytvoříme nový thread pomocí jednoho z nich, runtime systém ho automaticky zařadí do stejné skupiny, v které je thread vytvářející thread nový (skupina nazývaná *aktuální skupina threadu*).

Při spuštění aplikace runtime systém vytvoří objekt `ThreadGroup` zvaný „main“. Takže pokud není určeno jinak, všechny nové thready, které vytvoříme, se stanou členy skupiny „main“[8].

Vytvoření threadu a dané skupině

Jak bylo uvedeno výše, nelze thready přesouvat mezi skupinami. Proto pokud chceme vytvořit thread v jiné skupině než standardní, musí být skupina zadána explicitně při vytváření. Třída `Thread` poskytuje tři konstruktory, které umožňují nastavení skupiny nového threadu:

```
public Thread( ThreadGroup group, Runnable target );
public Thread( ThreadGroup group, String name );
public Thread( ThreadGroup group, Runnable target, String name );
```

Každý z těchto konstruktorů vytvoří nový thread, inicializuje ho na základě parametru `Runnable` a `String` a zařadí nový thread do specifikované skupiny. Například následující ukázka kódu vytvoří skupinu threadů (objekt `ThreadGroup`) nazvanou `myThreadGroup` a potom v této skupině vytvoří thread pojmenovaný `myThread`:

[8] Pokud vytváříme thread v appletu, jeho skupina nemusí být „main“, záleží to na použitém prohlížeči.

```
ThreadGroup myThreadGroup = new ThreadGroup( "My Group of Threads" );
Thread myThread = new Thread( myThreadGroup, "a thread for my group" );
```

Parametr `ThreadGroup` předávaný Thread konstruktoru nemusí nutně udávat skupinu, kterou jsme vytvořili. Může to být skupina vytvořená runtime systémem nebo skupina vytvořená aplikací, která spustila applet.

Zjištění skupiny threadu

Zjistit, do které skupiny daný thread patří, lze voláním jeho metody `getThreadGroup()`:

```
skupina = myThread.getThreadGroup();
```

Jakmile zjistíme skupinu, do které thread náleží, můžeme ji požádat o informace, jako například které další thready patří do skupiny, můžeme s thready skupiny manipulovat (suspend, resume, stop...) pomocí volání jedné metody.

5.7 Synchronizace threadů

Java poskytuje pro synchronizaci threadů dva nástroje:

- reentrantní monitory
- metody `notify()` a `wait()`

Monitory

Jazyk Java a runtime systém podporují synchronizaci threadů použitím *monitorů*. Monitory poskytují řešení pro sdílení objektů a synchronizaci přístupu ke sdílenému objektu. Sdílený objekt bývá nazýván condition proměnná (*sdílená proměnná, condition variable*). Java tak pomocí monitorů umožňuje synchronizaci threadů a přístup ke sdílené proměnné. Monitory zabraňují dvěma threadům v současném přístupu ke stejné proměnné.

Obecně monitor je svázán s určitým datovým objektem (condition proměnná) a funguje jako zámek pro tato data. Když thread vlastní monitor pro nějaký datový objekt, ostatním threadům není umožněn přístup k datovému objektu, je uzamčen.

Segment kódu v programu, který umožňuje odděleným, paralelním threadům přístup ke stejným datovým objektům, se nazývá *kritická sekce*. V Javě jsou kritické sekce v programu označeny klíčovým slovem `synchronized`.

Obecně jsou kritickými sekcemi v Java programech metody. Je možné označit malé části kódu (blok) slovem `synchronized`. To však porušuje objektově orientovaná paradigmata a vede k matoucímu kódu, který je obtížně laditelný. Pro většinu případů je nejlepší použít `synchronized` na úrovni metod.

V Javě je s každým objektem, který obsahuje synchronizovanou metodu, spojen unikátní monitor. Kdykoli při běhu programu vstoupí do synchronizované metody, thread, který zavolal tuto metodu, získá monitor pro objekt, jehož metoda je volána. Nyní ostatní thready nemohou zavolat synchronizovanou metodu téhož objektu, dokud není monitor uvolněn[9] (jsou zařazeny do fronty threadů čekajících na monitor a nachází se ve stavu „Nespustitelný“ viz. kapitola 5.3).

Získání monitoru je prováděno automaticky a atomicky runtime systémem.

Metody `notify()` a `wait()`

Metoda `wait()` je používána ve spojení s metodou `notify()` pro koordinaci běhu více threadů používajících stejné zdroje[10] (proměnné, objekty). Tyto metody jsou v balíku `java.lang.Object`.

Metoda `notify()`

Metoda `notify()` uvolní monitor a vzbudí jeden thread, který na něj čeká. Čekající thread požádá o monitor a provede svoji činnost.

Pokud na monitor čeká více threadů, runtime systém zvolí jeden z nich (není řečeno který).

Třída `Object` poskytuje další metodu `notifyAll()`, která vzbudí všechny thready čekající na tentýž monitor. V tomto případě vzbuzené thready „soutěží“ o získání monitoru. Jeden z nich ho získá a ostatní thready jsou znovu uvedeny do stavu „Nespustitelný“.

Metoda `wait()`

Metoda `wait()` způsobí, že současný thread čeká (možná navždy), dokud ho jiný thread neuvědomí metodou `notify()` o změně stavu condition proměnné (uvolnění monitoru, který je s ní svázaný).

Kromě metody `wait()`, kdy thread čeká na uvolnění monitoru donekonečna, třída `Object` poskytuje další dvě verze této metody:

`wait( long timeout )`
: Čeká na uvědomění (notify), tedy uvolnění monitoru, nebo pokračuje až vyprší daný časový limit udávaný v milisekundách.

`wait( long timeout, int nanos )`
: Čeká na uvědomění (notify), tedy uvolnění monitoru, nebo pokračuje až vyprší daný časový limit udávaný v milisekundách plus nanosekundách.

[9] Java monitory jsou reentrantní tzn., že tentýž thread může volat synchronizovanou metodu objektu, jehož monitor již vlastní, takže získá monitor znova. Tato vlastnost eliminuje možnost deadlocku threadu při žádosti o monitor, který již vlastní.

[10] Metody `notify()` a `wait()` lze volat pouze ze synchronizované metody.

Monitory a metody `notify()` a `wait()`

Potenciálním problémem monitoru je případ, kdy jeden thread získá monitor vstupem do synchronizované metody, zavolá metodu `wait()` a čeká až ho druhý thread uvědomí metodou `notify()`. Ten však tuto metodu volá uvnitř své synchronizované metody, tedy než ji spustí musí požádat o monitor. Ten však vlastní první thread.

Řešení tohoto problému je jednoduché. Jakmile thread zavolá metodu `wait()`, je monitor uvolněn atomicky a po návratu z metody `wait()` si jej znovu přivlastní. To umožňuje čekajícímu objektu (threadu) získat monitor a provést kód synchronizované metody (tím pádem zavolat metodu `notify()` a uvolnit tak první thread).