Vícevláknové programování

Běžci na dálkové tratě musí občas řešit dilema, jež jim vznikne, když na jeden týden připadnou dva velké závody a oni se musí rozhodnout, kterého z nich se účastní. Možná si přejí, aby existoval způsob, jak se rozdělit, takže by se jednou polovinou mohli zúčastnit jednoho závodu a druhou druhého. To však není možné – pokud ovšem není běžcem program v jazyce Java. Dvě části programu v jazyce Java totiž mohou běžet současně, a to pomocí vícevláknového zpracování. V této kapitole se budeme vícevláknovému zpracování věnovat a dozvíme se, jak současně spustit dvě části programu.

Paralelní zpracování úloh

Při paralelním zpracování úloh dochází k provádění dvou a více úloh současně. Téměř všechny operační systémy dovolují paralelní zpracování úloh pomocí jedné ze dvou následujících technik: paralelního zpracování na bázi procesů a paralelního zpracování na bázi vláken.

Paralelní zpracování na bázi procesů se stará o současné provádění více programů. Programátoři se na program odvolávají jako na *proces*. Proto lze říci, že paralelní zpracování na bázi procesů je paralelní zpracování na bázi programů.

Paralelní zpracování na bázi vláken znamená, že program provádí dvě úlohy současně. Například textový procesor může provádět kontrolu správnosti slov v dokumentu, zatímco my tento dokument píšeme. To je příklad vícevláknového paralelního zpracování.

Dobrým způsobem, jak si zapamatovat rozdíl mezi procesovým a vícevláknovým paralelním zpracováním, je představit si, že procesové zpracování pracuje s více programy a vícevláknové s několika částmi jednoho programu. Cílem paralelního zpracování je využít čas nečinnosti procesoru. Dívejme se na procesor jako na motor auta. Jeho motor běží bez ohledu na to, zda auto jede nebo stojí. Naším cílem je udržet auto v pohybu, jak jen to jde, abychom ujeli na 1 litr benzínu co nejvíce kilometrů. Motor běžící naprázdno jen spotřebovává benzín.

Stejným způsobem lze nahlížet i na procesor v počítači. Chceme, aby během svých taktů zpracovával instrukce a data, a ne aby čekal na něco, co má zpracovat. Takt procesoru je něco, co lze přirovnat ke spuštěnému motoru.

Možná se to zdá neuvěřitelné, ale v mnoha stolních počítačích procesor většinu svého času nic nedělá. Řekněme, že používáme textový procesor, na kterém píšeme dopis. Převážnou dobu našeho psaní procesor čeká na to, až zadáme znak z klávesnice nebo posuneme myší. Paralelní zpracování úloh je navrženo k tomu, aby využívalo zlomky sekund

mezi jednotlivými úhozy do klávesnice ke zpracování instrukcí z jiného programu nebo z další části stejného programu.

Snaha o zefektivnění využití procesoru není pro aplikace běžící na stolních počítačích až tak důležitá, protože málokdo z nás potřebuje současně spouštět více aplikací nebo provádět více částí stejného programu. Nicméně počítače provozované v síťovém prostředí, jako třeba ty, které zpracovávají transakce pro mnoho dalších počítačů, potřebují naplnit čekací čas procesoru produktivní činností.

Režie

Pro operační systém znamená řízení paralelního zpracování práci navíc. Programátoři tuto nadbytečnou práci nazývají *režií*, protože zdroje uvnitř počítače se používají k řízení operací paralelního zpracování místo toho, aby je využívaly programy pro zpracování instrukcí a dat.

Paralelní zpracování na bázi procesů má větší režii než vícevláknové zpracování. V paralelním zpracování procesů totiž každý proces potřebuje v paměti svůj vlastní adresový prostor. Operační systém dále potřebuje pro přepnutí z jednoho procesu na druhý poměrně velké množství času procesoru. Přepínání mezi procesy nazýváme *přepínání kontextu*. Znamená to, že každý proces (program) představuje kontext. Navíc jsou potřeba další zdroje pro vzájemnou komunikaci mezi procesy.

V porovnání s tím sdílejí vlákna v paralelním zpracování vláken v paměti stejný adresový prostor, protože pocházejí ze stejného programu. To má následný vliv i na přepínání kontextu, protože přepnutí z jedné části programu na druhou se děje v rámci jednoho adresového prostoru. Podobně i komunikace mezi jednotlivými částmi programu probíhá uvnitř stejné paměťové oblasti.

Vlákna

Vlákno je běžící část programu. Paralelní zpracování úloh na bázi vláken pracuje s vlákny běžícími současně (tj. více částí programu, které běží paralelně). Každé vlákno představuje jinou cestu provádění.

Vraťme se ještě jednou k programu na zpracování textu, abychom si na něm ukázali použití vláken. Zajímají nás dvě části textového procesoru: první je ta část programu, která zajišťuje přijímání znaků z klávesnice, ukládá je do paměti a zobrazuje na obrazovce. Druhá část programu ověřuje pravopis. Každá část je vlákno běžící nezávisle na tom druhém, a to i přes to, že obě jsou součástí stejného programu. Zatímco jedno vlákno přijímá a zpracovává znaky zadávané na klávesnici, druhé vlákno spí. To znamená, že druhé vlákno má přestávku až do té doby, než se procesor dostane do čekacího stavu. Procesor se mezi jednotlivými úhozy obvykle nachází v čekacím stavu. V tuto chvíli se probudí vlákno kontrolující pravopis a pokračuje v kontrole správnosti slov v dokumentu. Vlákno se opět uspí ve chvíli, kdy na klávesnici napíšeme další znak.

Na rozdíl od paralelního zpracování procesů, kde přepínání mezi programy zajišťuje operační systém, se prováděcí (run-time) prostředí Javy stará o řízení vláken. Vlákna se zpracovávají asynchronně. To znamená, že jedno vlákno čeká, zatímco druhé pokračuje v běhu.

Vlákno se může nacházet ve čtyřech stavech:

- **Běžící** Vlákno se provádí.
- **Pozastavené** Provádění vlákna je pozastaveno a může se obnovit od místa, ve kterém bylo zastaveno.
- **Blokované** Ke zdroji nelze přistoupit, protože jej používá jiné vlákno.
- **Ukončené** Provádění bylo zastaveno a nelze jej obnovit.

Všechna vlákna si nejsou rovna. Některá jsou důležitější než jiná a je jim dána vyšší priorita pro přístup ke zdrojům, jakým je například prováděcí čas procesoru. Každému vláknu se přidělí priorita, podle které se určuje, kdy se má přepnout provádění z jednoho vlákna na druhé. Říkáme tomu *přepínání kontextu*.

Priorita vlákna je relativní vůči prioritě ostatních vláken. Jinými slovy, priorita vlákna nemá význam, jestliže se jedná o jediné běžící vlákno. Pokud neexistuje další souběžně prováděné vlákno, běží vlákno s nižší prioritou stejně rychle jako vlákno s vyšší prioritou.

Priorita vlákna se používá ve chvíli, kdy se aplikují pravidla pro přepínání kontextu. Pravidla jsou tato:

- Vlákno může dát dobrovolně přednost jinému vláknu. Jestliže k tomu dojde, je řízení předáno vláknu s nejvyšší prioritou.
- Vlákno s vyšší prioritou může přerušit vykonávání vlákna s nižší prioritou. Chod vlákna s nižší prioritou je přerušen bez ohledu na to, co vlákno právě provádí. Programátoři tomu říkají *preemptivní paralelní zpracování*.
- Vlákna se stejnou prioritou se zpracovávají na základě pravidel operačního systému, která se používají pro spouštění programu. Například Windows používají *dávkování času*, což znamená, že každému vláknu s vysokou prioritou se přidělí několika milisekundový časový interval prováděcího času procesoru. Provádění se pak mezi vlákny s vysokou prioritou cyklicky střídá. V systému Solaris musí první vlákno s vysokou prioritou dát po nějaké době dobrovolně přednost jinému vláknu s vysokou prioritou. Pokud to neudělá, musí druhé vlákno čekat až na ukončení běhu prvního vlákna.

Synchronizace

Vícevláknové zpracování probíhá *asynchronně*, což znamená, že vlákno probíhá nezávisle na ostatních vláknech. Tímto způsobem není vlákno závislé na provádění ostatních vláken. Jinak tomu je u procesů běžících synchronně. Ty jsou závislé jeden na druhém. To znamená, že jeden proces čeká na ukončení dalšího a teprve poté může běžet.

Někdy je však i provádění vlákna závislé na chodu jiného vlákna. Řekněme, že máme dvě vlákna – jedno obsluhuje zadávání přihlašovacích informací a druhé má na starosti ověření přihlašovacího jména a hesla uživatele. Přihlašovací vlákno musí čekat na dokončení zpracování ověřovacím vláknem, a teprve pak může uživateli sdělit, zda bylo jeho přihlášení úspěšné. Z toho plyne, že obě vlákna musí běžet synchronně a nikoliv asynchronně.

Java dovoluje synchronizovat vlákna definováním synchronizační metody. Vlákno, které je uvnitř synchronizační metody, zabraňuje všem ostatním vláknům volat další synchronizační metodu nad stejným objektem. Více si povíme později v této kapitole.

Třída Thread a rozhraní Runnable

Vlákna vytváříme za pomoci třídy `Thread` a rozhraní `Runnable`. To znamená, že naši třídu musíme odvodit od třídy `Thread` nebo implementovat rozhraní `Runnable`. Třída `Thread` definuje dvě metody, které se používají ke správě vláken. V tabulce 10.1 nalezneme obvykle používané metody třídy `Thread`. Jejich použití si ukážeme při vysvětlování ilustračních příkladů této kapitoly.

Tabulka 10.1 Často používané metody definované ve třídě `Thread`

Metoda	Popis
`getName()`	Vrací název vlákna.
`getPriority()`	Vrací prioritu vlákna.
`isAlive()`	Zjišťuje, zda vlákno běží.
`join()`	Pozastaví provádění až do ukončení vlákna.
`run()`	Vstupní bod do vlákna.
`sleep()`	Pozastaví vlákno. Metoda dovoluje specifikovat časový interval, po který je ! vlákno pozastaveno.
`start()`	Spustí vlákno.

Hlavní vlákno

Každý program v jazyce Java má jedno vlákno. A to dokonce i tehdy, když žádné nevytvoříme. Toto vlákno se nazývá *hlavní*, protože se jedná o vlákno vytvořené a prováděné při spuštění programu. Z hlavního vlákna vznikají další vlákna, která vytvoříme. Těmto vláknům se říká *dceřiná* vlákna. Hlavní vlákno je vždy posledním vláknem, které ukončuje své provádění. Je to proto, že obvykle hlavní vlákno potřebuje uvolnit zdroje používané programem, jako například síťová spojení.

Programátor může hlavní vlákno řídit tak, že nejprve vytvoří objekt třídy `Thread` a potom použije jeho členské metody k řízení hlavního vlákna. Objekt třídy Thread vytvoříme zavoláním metody `currentThread()`. Metoda vrací odkaz na vlákno. Odkaz pak použijeme k řízení hlavního vlákna stejně, jako kdyby se jednalo o jakékoliv jiné vlákno. O tom si budeme povídat v dalších částech kapitoly.

Nyní zkusíme vytvořit odkaz na hlavní vlákno a potom změnit jeho jméno z „main“ na „Vlákno Demo“. Následující program ukazuje, jak to lze udělat. Podívejme se, co se vypíše na obrazovku po spuštění programu:

```
Aktuální vlákno: Thread[main,5,main]
Přejmenované vlákno: Thread[Vlákno Demo,5,main]
```

Zde je výpis programu:

```
class Demo {
  public static void main(String arg[]) {
```

```
        Thread v = Thread.currentThread();
        System.out.println("Aktuální vlákno: " + v);
        v.setName("Vlákno Demo");
        System.out.println("Přejmenované vlákno: " + v);
    }
}
```

Jak jsme si před chvílí řekli, při spuštění programu se vytvoří automaticky vlákno. Cílem příkladu je deklarovat odkaz na vlákno a potom do něj přiřadit odkaz na hlavní vlákno. To vše se děje v prvním příkazu metody `main()`.

Odkaz deklarujeme tak, že uvedeme jméno třídy a název proměnné, do které se bude odkaz ukládat. Tomu odpovídá další řádek:

```
Thread v
```

Dále potřebujeme získat odkaz na hlavní vlákno. K tomu zavoláme členskou metodu `currentThread()` třídy `Thread`.

```
Thread.currentThread()
```

Odkaz vrácený metodou `currentThread()` nakonec přiřadíme do proměnné definované na začátku příkazu. Informace o vlákně vypíšeme na obrazovku:

```
Thread[main,5,main]
```

Údaje uvedené uvnitř hranatých závorek nám sdělují něco o vlákně. První slovo, *main*, představuje název vlákna. Číslo 5 je priorita vlákna. Hodnota 5 znamená normální prioritu. Rozsah priority je v rozmezí 1 až 10, kde 1 je nejnižší a 10 nejvyšší priorita. Posledním slovem, opět main, je jméno skupiny vláken, do které vlákno přísluší. Skupina vláken je datová struktura používaná k řízení stavu souboru vláken. Skupinami vláken se však nemusíme zabývat, protože prováděcí prostředí Javy za nás vše obstará.

Metoda `setName()` se volá proto, abychom si ukázali, že máme kontrolu nad hlavním vláknem programu. Metoda `setName()` je členskou metodou třídy `Thread` a používá se ke změně jména vlákna. V tomto případě jsme metodu použili ke změně jména hlavního vlákna z „main“ na „Vlákno Demo“. A zase vypíšeme informace související s vláknem na důkaz toho, že došlo ke změně jeho jména. Výsledek je tento:

Přejmenované vlákno: Thread[Vlákno Demo,5,main]

Vytvoření vlastního vlákna

Už víme, že program je hlavní vlákno a další části programu mohou být také samostatnými vlákny. Část programu označíme jako vlákno tím, že vytvoříme nové vlákno. Nejjednodušším způsobem, jak toho dosáhnout, je implementovat rozhraní `Runnable`. Implementování rozhraní `Runnable` představuje alternativu k odvození třídy od třídy `Thread`.

Rozhraní popisuje jednu nebo více členských metod, které musíme ve třídě definovat, aby vyhovovala danému rozhraní. Metody jsou popsány jménem metody, seznamem argumentů a návratovou hodnotou.

Rozhraní `Runnable` popisuje metody, které třídy potřebují, aby mohly vytvářet vlákna a komunikovat s nimi. K tomu, abychom mohli použít rozhraní `Runnable` ve své vlastní třídě, v ní musíme definovat metody popsané tímto rozhraním.

Naštěstí je potřeba definovat pouze jednu metodu popsanou rozhraním `Runnable`. Tou metodou je metoda `run()`. Metoda `run()` musí být veřejná a není potřeba, aby měla seznam argumentů nebo návratovou hodnotu.

Obsah metody `run()` tvoří ta část programu, ze které chceme vytvořit nové vlákno. Příkazy vně této metody jsou součástí hlavního vlákna. Ve chvíli, kdy spustíme nové vlákno, což se naučíme za okamžik, začnou hlavní a nové vlákno běžet současně. Nové vlákno bude ukončeno, jakmile se ukončí běh metody `run()`. Řízení se poté vrátí do příkazu, který metodu `run()` zavolal.

Při implementaci rozhraní `Runnable` se musí zavolat níže uvedený konstruktor třídy `Thread`. Konstruktor vyžaduje dva argumenty. Prvním je instance třídy, která implementuje rozhraní `Runnable` a říká konstruktoru, kde se bude nové vlákno provádět. Druhý argument představuje jméno nového vlákna. Formát konstruktoru je následující:

```
Thread (Runnable trida, String jmeno)
```

Konstruktor nové vlákno vytvoří, ale nespustí je. Nové vlákno spustíme explicitně tím, že zavoláme metodu `start()`. Metoda `start()` volá metodu `run()`, kterou jsme definovali v programu. Metoda `start()` také nemá žádný seznam argumentů ani nevrací žádnou hodnotu.

Následující příklad ilustruje vytvoření a spuštění nového vlákna. Podívejme se, co program po svém spuštění vypíše na obrazovku:

```
Dceřiné vlákno spuštěno
Hlavní vlákno spuštěno
Dceřiné vlákno ukončeno
Hlavní vlákno ukončeno
```

Poznámka: Výstup z programu může vypadat i odlišně. To záleží na interpretu jazyka Java, který použijeme ke spuštění programu. Některé interprety tohoto jazyka ukončí běh hlavního vlákna dříve než běh vlákna dceřiného, zatímco jiné interprety ukončují běh dceřiného vlákna dříve hlavním vláknem. Proto může být výstup programu i v jiném pořadí, než v jakém ho uvádíme v této knize.

```
class MojeVlakno implements Runnable {
  Thread v;
  MojeVlakno () {
    v = new Thread(this,"Moje vlákno");
    v.start();
  }
  public void run() {
    System.out.println("Dceřiné vlákno spuštěno");
    System.out.println("Dceřiné vlákno ukončeno");
  }
}
class Demo {
  public static void main (String arg[]) {
    new MojeVlakno();
    System.out.println("Hlavní vlákno spuštěno");
    System.out.println("Hlavní vlákno ukončeno");
  }
}
```

Program začíná definicí třídy nazvané `MojeVlakno`, která implementuje rozhraní `Runnable`. Proto jsme použili klíčové slovo `implements`. Na dalším řádku jsme deklarovali odkaz na vlákno. Po něm následuje definice konstruktoru třídy. Konstruktor volá konstruktor třídy `Thread`. Vzhledem k tomu, že implementujeme rozhraní `Runnable`, potřebujeme do konstruktoru předat odkaz na instanci třídy, která bude nové vlákno spouštět, a jméno nového vlákna. Všimněme si, že jako odkaz na instanci třídy používáme klíčové slovo `this`. Klíčové slovo this představuje odkaz na aktuální instanci třídy.

Konstruktor navrací odkaz na nové vlákno. Odkaz přiřadíme do proměnné deklarované na prvním řádku v definici třídy `MojeVlakno`. Odkaz použijeme k volání metody `start()`. Jistě si vzpomenete, že metoda `start()` volá metodu `run()`.

V další části definujeme metodu `run()`. Příkazy uvnitř této metody se stanou částí programu, která se provádí při vykonávání vlákna. Metoda obsahuje jen dva zobrazovací příkazy. Později v této kapitole metodu `run()` rozšíříme o zajímavější příkazy.

Dále definujeme třídu programu. Tato třída explicitně spustí nové vlákno tím, že vytvoří instanci třídy `MojeVlakno`. Děje se to pomocí operátoru `new` a volání konstruktoru třídy `MojeVlakno`.

V poslední části program vypíše na obrazovku dva řádky a je ukončen.

Vytvoření vlákna pomocí klíčového slova `extends`

Dědění od třídy `Thread` je další možný způsob vytvoření nového vlákna v programu. Jak jsme si říkali v 8. kapitole, třídu odvodíme od jiné třídy použitím slova `extends` v definici nové třídy. Když pak deklarujeme instanci této nové třídy, budeme mít zajištěn přístup i ke členům třídy `Thread`. Kdykoliv odvozujeme třídu od třídy `Thread`, musíme překrýt metodu `run()`.

V dalším příkladu definujeme třídu `MojeVlakno`. Třídu odvozujeme od třídy `Thread`. Konstruktor třídy `MojeVlakno` volá konstruktor třídy `Thread` pomocí klíčového slova `super` a předává do něj název nového vlákna, kterým je v našem případě jméno „Moje vlákno“. Poté voláme metodu `start()`, čímž vlákno aktivujeme.

Metoda `start()` volá metodu `run()` třídy `MojeVlakno`. Jak poznáme z kódu, metodu `run()` jsme překryli, takže nyní na obrazovku vypisuje dva řádky, které informují o spuštění a ukončení dceřiného vlákna. Nezapomínejme, že příkazy v těle metody `run()` vytvářejí tu část programu, která poběží v novém vláknu. Ve skutečném programu budou asi smysluplnější příkazy, než jaké máme v našem příkladě.

Nové vlákno jsme deklarovali uvnitř metody `main()` třídy Demo. Tato třída je programovou třídou aplikace. Po spuštění nového vlákna se zobrazí dvě zprávy, které indikují stav hlavního vlákna.

```
class MojeVlakno extends Thread {
  MojeVlakno () {
    super("Moje vlákno");
    start();
  }
```

```
    public void run() {
      System.out.println("Dceřiné vlákno spuštěno");
      System.out.println("Dceřiné vlákno ukončeno");
    }
  }
  class Demo {
    public static void main (String arg[]) {
      new MojeVlakno();
      System.out.println("Hlavní vlákno spuštěno");
      System.out.println("Hlavní vlákno ukončeno");
    }
  }
```

Poznámka: Při vytváření nového vlákna bychom se měli řídit následujícím pravidlem. Je-li metoda `run()` jedinou metodou, kterou budeme ve třídě `Thread` překrývat, měli bychom implementovat rozhraní `Runnable`. Odvozovat třídu od třídy `Thread` bychom měli jen v případě, že budeme překrývat více metod definovaných v této třídě.

Použití více vláken v programu

Není nic neobvyklého, když program spouští více instancí jednoho vlákna. Příkladem může být třeba souběžný tisk více dokumentů. Programátoři tomu říkají *otevření* vlákna. Otevřít se dá libovolný počet vláken. Nejprve musíme definovat vlastní třídu, která buď implementuje rozhraní `Runnable` a nebo se odvodí od třídy `Thread`. Poté deklarujeme instance této třídy. Každá instance je nové vlákno.

Podívejme se na to, jak to udělat. V následujícím příkladě jsme definovali třídu `MojeVlakno`, která implementuje rozhraní `Runnable`. Konstruktor třídy `MojeVlakno` přijímá jeden argument, kterým je řetězec nesoucí jméno nového vlákna. Nové vlákno vytvoříme uvnitř konstruktoru tak, že zavoláme konstruktor třídy `Thread` a předáme mu odkaz na objekt definující vlákno a název vlákna. Nezapomeňme, že klíčové slovo `this` je odkazem na aktuální objekt. Dále voláme metodu `start()`, která zase volá metodu `run()`.

Metoda `run()` je ve třídě `MojeVlakno` překryta. Při volání metody se stanou dvě věci. Za prvé se na obrazovce zobrazí jméno vlákna. Za druhé se vlákno na dvě sekundy pozastaví, a to pomocí volání metody `sleep()`. Metoda `sleep()` je definována ve třídě `Thread` a akceptuje jeden nebo dva argumenty. Prvním argumentem je počet milisekund, na který se má vlákno zastavit. Druhým parametrem je počet mikrosekund, na který se má vlákno zastavit. V našem příkladě nás zajímají jen milisekundy, takže druhý argument neuvádíme (2 000 milisekund = 2 sekundy). Po prodlevě vypíšeme na obrazovku dalším příkazem informaci o ukončení vlákna.

Metoda `main()`třídy `Demo` deklaruje čtyři instance stejného vlákna. Děláme to tak, že voláme konstruktor třídy `MojeVlakno` a předáváme do něj název vlákna. Každá instance je brána jako samostatné vlákno. Běh hlavního vlákna se následně pozastaví na 10 sekund. Opět k tomu používáme metodu `sleep()`. Během této doby ostatní vlákna pokračují v chodu. Když se hlavní vlákno probudí, napíše zprávu o tom, že jeho činnost je ukončena.

A takto vypadá výpis obrazovky po spuštění programu:

```
Vlákno: 1
Vlákno: 2
Vlákno: 3
Vlákno: 4
Ukončení vlákna: 1
Ukončení vlákna: 2
Ukončení vlákna: 3
Ukončení vlákna: 4
Ukončení vlákna: hlavní vlákno.
```

A zde je kód programu:

```
class MojeVlakno implements Runnable {
  String JmenoVlakna;
  Thread vlakno;
  MojeVlakno (String threadName) {
    JmenoVlakna = threadName ;
    vlakno = new Thread (this, JmenoVlakna);
    vlakno.start();
  }
  public void run() {
    try {
      System.out.println("Vlákno: " + JmenoVlakna);
      Thread.sleep(2000);
    } catch (InterruptedException v) {
      System.out.println("Výjimka: vlákno " + JmenoVlakna + " přeruše-
no");
    }
    System.out.println("Ukončení vlákna: " + JmenoVlakna);
  }
}
class Demo {
  public static void main (String arg[]) {
    new MojeVlakno("1");
    new MojeVlakno("2");
    new MojeVlakno("3");
    new MojeVlakno("4");
    try {
      Thread.sleep(10000);
    } catch (InterruptedException v) {
      System.out.println("Výjimka: Hlavní vlákno přerušeno.");
    }
    System.out.println("Ukončení vlákna: hlavní vlákno.");
  }
}
```

Použití metod `isAlive()` a `join()`

Obvykle je hlavní vlákno posledním vláknem, které se v programu ukončuje. Nicméně nemáme žádnou záruku toho, že hlavní vlákno neskončí dříve než vlákna dceřiná. V předešlém příkladě jsme uspali hlavní vlákno na tak dlouho, dokud nedošlo k ukončení dceřiných vláken. Přitom jsme však odhadovali čas, který potřebují dceřiná vlákna na dokončení zpracování. Pokud by náš odhad byl příliš krátký, dceřiné vlákno by se mohlo ukončit až poté, co bylo ukončeno hlavní vlákno. Z toho plyne, že technika využívající metodu `sleep()` není tou nejlepší, pokud chceme zaručit, že se hlavní vlákno ukončí jako poslední.

Naštěstí máme k dispozici jiné dvě techniky, které zajistí, že hlavní vlákno bude posledním ukončeným vláknem. Tyto techniky zahrnují volání metod `isAlive()` a `join()`. Obě metody jsou definovány ve třídě `Thread`.

Metoda `isAlive()` zjišťuje, zda dané vlákno stále běží. Pokud ano, vrátí metoda booleovskou hodnotu true. V opačném případě vrátí hodnotu false. Metoda `isAlive()` se dá použít ke zjištění, zda dceřiný proces stále běží. Metoda `join()` funguje jinak než metoda `isAlive()`. Metoda `join()` čeká, dokud se dceřiný proces neukončí a „nepřidá“ se k hlavnímu vláknu. Navíc můžeme u této metody specifikovat množství času, po který má čekat na ukončení dceřiného procesu.

Další ukázka ilustruje použití metod `isAlive()` a `join()` v programu. Program se téměř podobá předchozímu. Rozdíl spočívá v metodě `main()` třídy `Demo`.

Po deklarování vláken za použití konstruktoru třídy `MojeVlakno` voláme pro každé vlákno metodu `isAlive()`. Hodnotu, kterou metoda vrací, vypisujeme na obrazovku. V dalším kroku voláme pro každé vlákno metodu `join()`. Metoda způsobí, že hlavní vlákno čeká na dokončení zpracování všech dceřiných vláken, a teprve poté dojde k jeho ukončení.

A takový je výstup na obrazovku:

```
Stav vlákna: isALive
Vlákno 1: true
Vlákno 2: true
Vlákno 3: true
Vlákno 4: true
Vlákno: join
Vlákno: 1
Vlákno: 2
Vlákno: 3
Vlákno: 4
Ukončení vlákna: 1
Ukončení vlákna: 2
Ukončení vlákna: 3
Ukončení vlákna: 4
Stav vlákna: isALive
Vlákno 1: false
```

```
Vlákno 2: false
Vlákno 3: false
Vlákno 4: false
Ukončení vlákna: hlavní vlákno.
```

Výpis programu:

```
class MojeVlakno implements Runnable {
  String JmenoVlakna;
  Thread vlakno;
  MojeVlakno (String vlaknoName) {
    JmenoVlakna = vlaknoName ;
    vlakno = new Thread (this, JmenoVlakna);
    vlakno.start();
  }
  public void run() {
    try {
      System.out.println("Vlákno: " + JmenoVlakna);
      Thread.sleep(2000);
    } catch (InterruptedException v) {
      System.out.println("Výjimka: vlákno " + JmenoVlakna + " přerušeno");
    }
    System.out.println("Ukončení vlákna: " + JmenoVlakna);
  }
}
class Demo {
  public static void main (String arg[]) {
    MojeVlakno vlaknol = new MojeVlakno("1");
    MojeVlakno vlakno2 = new MojeVlakno("2");
    MojeVlakno vlakno3 = new MojeVlakno("3");
    MojeVlakno vlakno4 = new MojeVlakno("4");
    System.out.println("Stav vlákna: isALive");
    System.out.println("Vlákno 1: " + vlaknol.vlakno.isAlive());
    System.out.println("Vlákno 2: " + vlakno2.vlakno.isAlive());
    System.out.println("Vlákno 3: " + vlakno3.vlakno.isAlive());
    System.out.println("Vlákno 4: " + vlakno4.vlakno.isAlive());
    try {
      System.out.println("Vlákno: join");
      vlaknol.vlakno.join();
      vlakno2.vlakno.join();
      vlakno3.vlakno.join();
      vlakno4.vlakno.join();
    } catch (InterruptedException v) {
      System.out.println("Výjimka: Hlavní vlákno přerušeno.");
    }
    System.out.println("Stav vlákna: isALive");
    System.out.println("Vlákno 1: " + vlaknol.vlakno.isAlive());
    System.out.println("Vlákno 2: " + vlakno2.vlakno.isAlive());
```