Javan säikeet ja kriittisen vaiheen kontrollointi niiden käytön yhteydessä

1 Johdanto

Usein on luontevaa valita matalan tason ohjelmointikieli matalan tason ohjelmoinnille, mutta Javassa rinnakkaisuus on rakennettu sisälle. Se on ollut mukana alusta asti ja sille on olemassa kattava tuki. Näistä tärkeimmät ovat abstrakti luokka Thread ja rajapinta Runnable. Rinnakkaisuuden hallintaan monisäikeisille ohjelmille löytyy Javasta myös tehokasta valmista kalustoa *java.util.concurrent* -paketista

2 Säikeiden käyttö

Johdannossa kuvattiin lyhyesti Javassa olevan *Thread*-luokka ja *Runnable*-rajapinta. Näistä kahdesta yleisemmin on käytössä *Runnable*, sillä se sallii luokan perivän muita luokkia.

2.1 Threadin hyödyntäminen

Thread-luokka toteuttaa rajapintaluokan *Runnable*. Uusi säie konstruoidaan antamalla uudelle säikeelle parametrina *Runnable*-rajapintaluokan toteuttava luokka.

```
Runnable r = new AjettavaLuokka();
Thread säie = new Thread(r);
säie.start();
```

Mahdollista on myös korvata *Thread*-luokkan *run()*-metodi, kuten alla olevassa esimerkissä on tehty. Metodi *run()* määrittelee sen, mitä säikeen halutaan tekevän käynnistyttyään. Uusi säie saadaan käyttöön luomalla ensin uusi olio ja määräämällä **start()**. Tärkeää on käyttää *start()*-metodia säikeen käynnistämiseen, eikä esim metodia *run()*, sillä vasta *start()* käynnistää erillisen säikeen suorituksen. Olio voidaan myös käskeä nukkumaan komennolla **Thread.sleep(aikaMillisekunteina), jolloin säie ikään kuin nukkuu määritellyn ajan verran. Tällöin on myös otettava kiinni tarkistettu poikkeus *InterruptedException***. Toimitus voi häiriintyä, jos jokin muu säie kutsuu nukkuvan säikeen **interrupt()**-metodia.

Useiden säikeiden käyttö vaatii ennen pitkää väistämättä niiden keskinäistä synkronointia. Siihen Javassa on *synchronized*-avainsana, joka kohdistuu aina tiettyyn olioon tai metodiin. Avainsanalle määritellään suluissa kohde (olio), jota seuraa lohko, jonka sisälle päästään vain kun säie saa olion haltuunsa, ja näitä säikeitä voi olla vain yksi kerrallaan. Olio vapautuu tämän jälkeen. Avainsana *synchronized* voidaan kirjoittaa myös metodin eteen jolloin se tekee vastaavan eli metodia pääsee suorittamaan vain yksi säie kerrallaan.

Synchronized-lohkon olio voi olla itse määritelty, joka annetaan parametrina luokalle, jota halutaan synkronoida. Nyt ennen kriittisen vaiheen aloittamista avataan synkronointilohko.

Esimerkkikoodissa 100 säiettä kasvattavat kaikki samaa muuttujaa. Kaikki lisäävät sen arvoon 100. Rinnakkaisuudesta aiheutuvat ongelma on ratkaistu sykronoidulla oliolla.

2.1.1 Esimerkki: Tulostaja.java

```
public class Tulostaja extends Thread {
      protected Object synkronointiolio;
      protected String tulostaja;
      private static int luku = 0;
```

```
	Tulostaja(String tulostaja, Object olio) {
		this.tulostaja = tulostaja;
		this.synkronointiolio = olio;
	}

	//peritty metodi Thread-luokalta, säikeen "ydin"
	public void run() {

		for(int i = 1; i <= 100; i++) {
			synchronized(synkronointiolio) {
				int temp = luku;
				luku = temp + 1;
				System.out.println(tulostaja+" tulostaa: "+luku);
			}
			try {
				Thread.sleep(100);
			}
			catch(InterruptedException ie) {
			// Tähän jotain
			}
		}
	}

	public static void main(String args[]) {
		Object synkro = new Object();
		for(int i = 1; i <= 100; i++) {
			Tulostaja counter = new Tulostaja(Integer.toString(i),
synkro);
			counter.start();
		}
	}
}
```

2.2 Runnable-rajapinnan hyödyntäminen

Runnable-rajanpintaa käytettäessä on Threadin tapaan määriteltävä run()-metodi, mutta luokan määrittely muuttuu hieman. Tutustutaan esimerkkiin Opiskelija, joka on perinyt Henkilo-luokan.

2.2.1 Esimerkki: Henkilo.java

```
public class Henkilo {
	private String nimi;
	protected int ika=0;
	protected int elinika;

	public Henkilo(String n) {
		this.nimi = n;
		this.elinika = (int)(Math.random()*100);
	}


	public void tervehdi() {
		System.out.println("Terve, olen "+nimi+","+ika+" vuotta");
	}
}
```

2.2.2 Esimerkki: Opiskelija.java

```
public class Opiskelija extends Henkilo implements Runnable {
	public Opiskelija(String n) {
		super(n); // Käytetään yliluokan konstruktoria
	}
```

```
	public void run() {
		for (;ika <= elinika; ika++) {
			tervehdi();
				if (ika < 25)
					System.out.println("Ei enää!");
				else
					System.out.println("Mielenkiintoista! Kerro toki
lisää");
		}
	}
}
```

2.2.3 Esimerkki: Paaohjelma.java

```
public class Paaohjelma {

	public static void main(String args[]) {
		Opiskelija eeva = new Opiskelija("Eeva");
		Thread eevasaie = new Thread(eeva);
		eevasaie.start();
	  }
}
```

3 Valmiita apuvälineitä

Olemme käsitelleet jo joitakin metodeja, joita voidaan käyttää java-säikeitä käsitellessä. Lisäksi on olemassa muutama toiminto, joihin on tarjottu valmis toteutus javan valmiissa kalustossa. Esittelemme alla sekä jo mainitut metodit, että muutaman uuden:

- **run()** - Mitä tehdään, kun säie on käynnistynyt
- **start()** - Käynnistää säikeen, eli kutsuu run():ia
- **sleep(long millis)** - Nukuttaa säikeen määräajaksi. Säie ei menetä sen hallussa olevia monitoreita tänä aikana.
- **yield()** - Keskeyttää säikeen ja päästää muut säikeet suorittamaan.
- **join()** - Säie jatkaa suoritusta vasta kun määritelty säie on saanut suorituksensa päättymään. Thread -luokan API

Yllä olevien metodien lisäksi java.lang.Objectilla on muutama metodi, joista voi olla hyötyä:

- **wait()** - Säie pysähtyy odottamaan oliota
- **notify()** - Yksi oliota odottavista säikeistä herätetään
- **notifyAll()** - Kaikki oliota odottavat säikeet herätetään

4 Keskeytykset

Tutustuimme ohimennen esimerkkiin, jossa säi käskettiin suspend-tilaan pieneksi hetkeksi (Esimerkki Tulostaja.java). Käytimme try-catch:ia selviytyäksemme mahdollisista virheistä, mutta toinen tapa on tarkistaa itse tilanne poikkeuksien varalta.

Oletetaan, että koodi joutuu tekemään paljon työtä, joka ei kutsu metodeja, jotka saattaisivat heittää tarkistettuja poikkeuksia. Koodissa voidaan tällöin kysyä onko keskeytyksiä tullut:

```
if (Thread.interrupted() {
	throw new InterruptedException();
}
```

Kun säie keskeytetään, sille asetetaan keskeytystila, joka kertoo, että säie on keskeytetty. Mikäli

tästä halutaan päästä eroon, voidaan käyttää edeltävää metodia. Toisaalta, jos halutaan vain tarkistaa keskeytyksen tila, voidaan käyttää metodia Thread.isInterrupted, joka jättää keskeytystilan ennalleen.

5 Säikeiden toteutus käytännössä

5.1 Mitä tapahtuu käyttöjärjestelmätasolla?

Käyttöjärjestelmä voi tukea säikeitä kahdella tavalla: Ohjelmassa voi olla käyttäjäsäikeitä, ja käyttöjärjestelmässä voi pyöriä ydinsäikeitä. Ensimmäiselle on olemassa laaja säiekirjasto, joka hoitaa säikeiden ajastuksen niin, että käyttöjärjestelmän ytimen ei tarvitse puuttua asiaan. Käyttöjärjestelmän ydin tukee suoraan ydinsäikeitä, eli ydin luo säikeet, ajastaa ja hallitsee niitä. Javassa tämä ei ole aivan yhtä kahtiajakoista, sillä esimerkiksi Windowsissa yhtä Java-säiettä vastaa yksi ydinsäie, mutta UNIX:issa Javan säikeitä kuvataan joukoksi ydinsäikeitä. Java-säikeitä on kahdenlaisia (käyttäjä- ja demonisäikeet) ja niillä on myös prioriteetit.

5.2 Javan toteutus

Jokaisella olion ilmentymällä ja luokalla on javassa siihen mahdollisesti assosioitunut monitori. Sanavalinta "mahdollisesti" tulee siitä, että mikäli mitään synkronointityökaluja ei käytä, monitoria ei koskaan allokoida - se on kuitenkin olemassa.

Synkronoidun metodin suorituksen aikana säie varaa monitorin kyseisen metodin oliolle, tai jos metodi on staattinen, monitori varataan metodin luokalle. Jos puolestaan toinen säie suorittaa synkronoitua metodia, ensimmäinen joutuu odottamaan, kunnes jälkimmäinen vapauttaa monitorin (joko lopettamalla metodin suorituksen tai käyttämällä wait()-kutsua).

Säie kutsuu synkronoitua metodia olion sisällä halutessaan eksplisiittisesti päästä käsiksi sen monitoriin. Vapauttaakseen monitorin hetkellisesti, säie kutsuu wait()-funktiota. Koska säikeen on täytynyt ensin päästä käsiksi olion monitoriin, wait()-metodia on mahdollista kutsua vain synkronoidun metodin sisältä. Wait()-metodin käyttö tähän tapaan mahdollistaa rendezvousin toisen säikeen kanssa tietyllä synkronointihetkellä.

6 java.util.concurrent*

Java tarjoaa edellisten lisäksi myös muita valmiita työkaluja monisäikeisyyden ja kriittisen vaiheen hallintaan. Tämän paketin luokat ja rajapinnat antavat paljon laajemmat mahdollisuudet rinnakkaisuuden hallintaan.

Seuraava esimerkki on alkupään esimerkki toteutettuna semaforilla.

Esimerkki 6.1 TulostajaSemaphore.java

```
import java.util.concurrent.Semaphore;

public class TulostajaSemaphore extends Thread {
	private static int luku = 0;
	private static Semaphore sema = new Semaphore(0);
	private static Semaphore mutex = new Semaphore(1);

	//peritty metodi Thread-luokalta, säikeen "ydin"
	public void run() {

		for(int i = 1; i <= 100; i++) {
			try {
				mutex.acquire();
				//Kriittinen vaihe
				int temp = luku;
				luku = temp + 1;
```

```
                        //Kriittinen vaihe päättyy
                        mutex.release();
                        Thread.sleep(10);
                  }
                  catch(InterruptedException ie) {
                        System.out.println("Keskeytetty");
                  }
            }
            if(luku == 10000) sema.release();
      }

      public static void main(String args[]) throws InterruptedException {
            TulostajaSemaphore counter = null;
                  for(int i = 1; i <= 100; i++) {
                        counter = new TulostajaSemaphore();
                        counter.start();
                  }
                  sema.acquire();
                  System.out.println("Luvun arvoksi tuli: " + luku);
      }
}
```

Luennoilta tuttu tapaus, jossa mehiläiset ruokkivat loukkuun jäänyttä karhua. Tässä on käytetty halkaistuja semaforia.

Esimerkki 6.2 Hunajapurkki.java

```
import java.util.concurrent.Semaphore;

public class Hunajapurkki {

      public int portions = 0;
      public final int portionsMAX = 50;

      public Semaphore karhuNukkuu = new Semaphore(0);
      public Semaphore ampppariPurkilla = new Semaphore(1);

      public long filltimes = 10;
      public int amppareita = 10;

      public static void main(String args[]) throws InterruptedException {

            Hunajapurkki honeypot = new Hunajapurkki();
            Karhu teddybear = new Karhu(honeypot);
            Ampiainen bee;

            teddybear.start(); // Nalle nukkumaan

            //Ampparit matkaan
            for(int i = 1; i <= honeypot.amppareita; i++) {
                  bee = new Ampiainen(honeypot, i);
                  bee.start();
            }

            teddybear.join(); // Ei mennä eteenpäin jos nallu ei ole saanut
tarpeeksi hunajaa.
            System.out.println("Nallua on nyt syötetty tarpeeksi.");
      }
}

class Ampiainen extends Thread {
      Hunajapurkki purkki;
      int ampppariID;
```

```
	Ampiainen(Hunajapurkki purkki, int ampppariID) {
		this.purkki = purkki;
		this.ampppariID = ampppariID;
	}

	public void run()  {
		while(true) {
			try {
				sleep(100);
				purkki.ampppariPurkilla.acquire();

				if(purkki.filltimes <= 0)
					break;

				purkki.portions++;
				System.out.println("Amppari: " + ampppariID + " Purkissa
hunajaa: " + purkki.portions + ".");

				if (purkki.portions == purkki.portionsMAX)
					purkki.karhuNukkuu.release();
				else
					purkki.ampppariPurkilla.release();
			}
			catch(InterruptedException ie) {}

		}
	}
}

class Karhu extends Thread {
	private Hunajapurkki purkki;

	Karhu(Hunajapurkki purkki) {
		this.purkki = purkki;
	}

	public void run() {
		while(purkki.filltimes != 0)
			try {
				purkki.karhuNukkuu.acquire();
				purkki.filltimes--;
				eatHoney();
				sleep(500); // Nukutaan hiukan
				if(purkki.filltimes == 0) {
					// Karhu on syönyt tarpeeksi, joten päästetään
kaikki ampparit vapaalle.
					purkki.ampppariPurkilla.release(purkki.amppareita);
				 }
				purkki.ampppariPurkilla.release();
			}
			catch(InterruptedException ie) {}
	}

	private void eatHoney() {
		System.out.println("Nalle syö hunajaa " + purkki.filltimes + "
kertaa jäljellä.");
		purkki.portions = 0;
	}
}
```

7 Yhteneväisyydet ja eroavaisuudet kurssilla esitettyyn perustapaukseen

Käytyämme kurssimateriaalin perusteellisesti läpi huomaamme, että säikeistä puhutaan niissä hyvin vähän. Koemme, että esityksemme ei niinkään eroa esitetystä kuvauksesta, vaan pikemminkin laajentaa ja tarkentaa sitä. Paneuduimme käytännön ohjelmointiin ja säikeiden hyödyntämiseen nimenomaan javassa tehtävänannon mukaisesti, mikä todennäköisesti hieman erosi kurssin alun lyhyehkön selostuksen lähestymistavasta. Tekstimme sopinee ennen kaikkea ihmiselle, joka haluaa ottaa selvää säikeistä kurssimateriaalia syvemmin ja käytännönläheisemmin.

8 Lähteet

- http://www.tol.oulu.fi/~avesanen/Rinn_Ohjelm/Luennot/RinnakkaisJava.html
- http://javala.cs.tut.fi/show.do?category=saikeet
- http://www.kuuskeri.com/nil/lintula/distributed/java/presentation/index/v3dcmnt.htm
- http://www.cs.helsinki.fi/u/kerola/rio/Java/thread_guides/s06-kahilakoski-leino/markolasse_pr1_java.pdf
- http://www.javaworld.com/javaworld/jw-04-1996/jw-04-synch.html
- http://java.sun.com/developer/technicalArticles/J2SE/concurrency/index.html
- http://java.sun.com/javase/6/docs/api/