1 Trådar

1.1 Runnable och Thread

Kommentarer

Se javakoden.

- I övningen är **ShoutThread** hårdkodad att använda just **ShoutRunnable**. Det typiska förfarandet brukar annars vara att skicka över din **Runnable** i konstruktor-anropet till **Thread**:

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

- Att endast använda sig av **Thread** för att åstadkomma parallel exekvering (d.v.s. låta klassen ärva av **Thread**) kan vara motiverat i små exempel och för klasser interna i ett paket. Men i de flesta fall är det rekommenderat att istället placera koden i en **Runnable**. Det finns flera anledningar:
  - Det innebär en lösare koppling mellan koden och själva körningen.
  - Om klassen ärver av **Thread** kan den inte ärva av någonting annat.
  - Körningen av koden kan väljas flexibelt, **Thread** är inte det enda alternativet. *Executor Framework*[1] erbjuder t.ex. hantering och körning av flera trådar asynkront.

1.2 Time of check/Time of use (TOCTOU)

- TOCTOU-problemet kan ej uppstå för strikt icke-muterbara klasser. Då hela deras tillstånd initieras vid instantiering (konstruktoranropet) finns det ingenting som kan förändras mellan "check" och "use". Strikt icke-muterbara klasser är per automatik trådsäkra. Notera att strikt icke-muterbarhet innebär att de helt saknar referenser till andra muterbara objekt.

1.2.1 TOCTOU I

Se javakoden.

- En angripare kan köra två trådar, i den ena tråden skapas en ny **Period**, medan den andra ligger på lur, även den med tillgång till argumenten *start* och *end*. Efter att konstruktorn i den första tråden kört testet, och konstaterat att $start \leq end$, kan den andra tråden bryta detta villkor innan konstruktorn tar kopior av *start* och *end*.

[1] `http://download.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html`

1.3 Deadlock

Kommentarer

Se javakoden.

Den önskade utskriften är:

```
Emil says: I got a question from Pelle
Pelle says: I got an answer from Emil
Pelle says: I got a question from Emil
Emil says: I got an answer from Pelle
```

- Observera något som kanske är självklart, men lätt att blanda ihop när man börjar studera parallela program: En tråd är inte knuten till ett visst objekt. Körningen startar visserligen i en viss metod i ett objekt (t.ex. **run()** i ett **Runnable**-objekt), men den kan sedan fortsätta inuti ett annat objekts metoder genom anrop av dem.

- Det första problemet bygger på att det inte finns någon kontroll över vad de olika trådarna får göra. Utskrifternas inbördes ordning beror på hur schemaläggningen av trådarnas körning ser ut.

- Tråden som startas i objektet "Pelle" tar monitorn till objektet "Emil" (genom att köra en metod där i deklarerad **synchronized**), och samtidigt sker det motsatta: tråden som startas i objektet "Emil" tar monitorn till objektet "Pelle". Innan de släpper sina egna monitorer vill båda trådarna köra synkroniserade metoder i motsatt objekt, men eftersom dessa monitorer är upptagna av den andre tråden får vi en *deadly embrace*, d.v.s. deadlock.

- I det generella fallet är det bevisat att det inte går att förutsäga om deadlock kommer att kunna inträffa vid ett framtida tillfälle bland ett antal resurser och parallela program som efterfrågar dessa resurser[2]. Om situationen tillåter att vissa begränsningar införs går det att förhindra att deadlock överhuvudtaget kan uppstå. Men dessa begränsningar lämpar sig sällan för operativsystem i persondatorer.

- Allmänt är det riskabelt att anropa metoder på andra objekt inne i ett synkroniserat block; att de metoderna inte har egna lås är svårt eller omöjligt att garantera. Då vi i lösningen bara låser under tiden vi sover och sedan släpper låset innan vi anropar `p.getAnAnswerFrom(this)` kommer objektet `p` kunna anropa `this` tillbaks, utan deadlock.

[2] http://en.wikipedia.org/wiki/Deadlock

1.4 Ofullständig klassinvariant & race conditions

Kommentarer

Se javakoden.

- För svar till uppgift 1–4, se javakoden inklusive kommentarer.
- Observera hur kod som uppfyller krav på både funktionalitet och klassinvarianter ändå kan ha problem när parallel exekvering införs. Parallelism kräver ofta att man funderar igenom olika typer av körningssituationer för att upptäcka brister och säkerhetshål.
- För att knyta en bit kod (en hel metod eller ett antal satser) till en viss monitor deklareras koden **synchronized**. Detta innebär som bekant att endast en tråd i taget kan köra någon av alla kodpartier som är knutna till motsvarande monitor. Men observera att deklaration av **synchronized** också är ett krav om koden påverkar variabler som flera olika trådar skriver till och läser från. För att garantera att uppdateringarna kommer i rätt ordning måste all kod som använder variablerna deklareras **synchronized**.

2 Innerklasser och anonyma klasser

2.1 Det kan vara fint med få klasser...

Kommentarer

Se javakoden.

- Metoden **stringPrefixComparator()** kan inte returnera en singleton-instans, utan den måste skapa en ny komparator vid varje anrop eftersom varje sådan komparator är beroende av ett parametervärde (**prefixLength**).
- Observera att parametern (**prefixLength**) till metoden **stringPrefixComparator()** måste deklareras **final** eftersom den används i den anonyma klassen.

2.2 Statiska och icke-statiska Innerklasser

Kommentarer

Se javakoden.

- Klassen **Game** kan vara en statisk (**static**) klass eftersom den inte behöver ha tillgång till något tillstånd (instansvariabler) i objekt av den omgivande klassen.

3 I/O

3.1 Heltalsförflyttning

Kommentarer

Se javakoden.