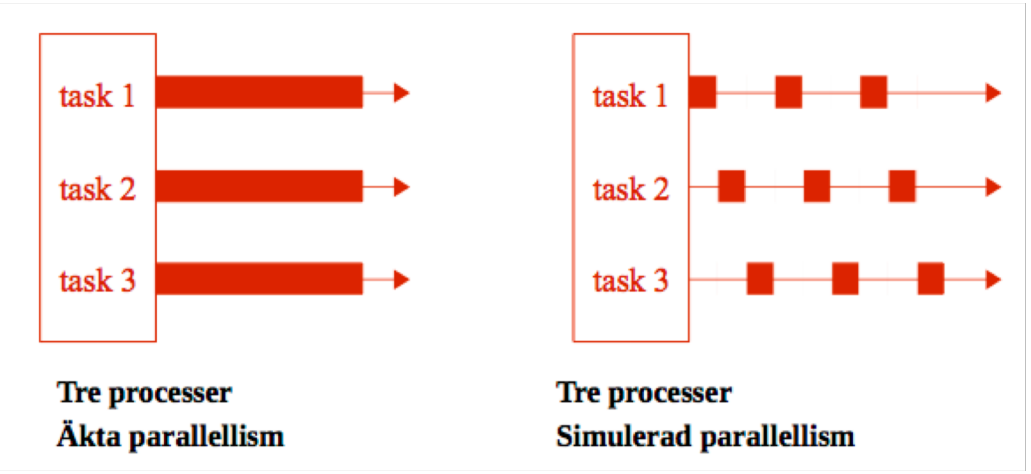
Trådar och trådsäkerhet

Objekt-orienterad programmering och design

Alex Gerdes, 2018

Aktiva objekt

- Det är välkänt från vardagslivet att saker händer samtidigt.
- Det är viktigt att det underliggande systemet stöder parallellism så att man kan modellera verkligheten på ett bra sätt.
- Det är onaturligt att modellera parallellism som ett sekventiellt förlopp. I datorer med en processor *simuleras parallellismen*.

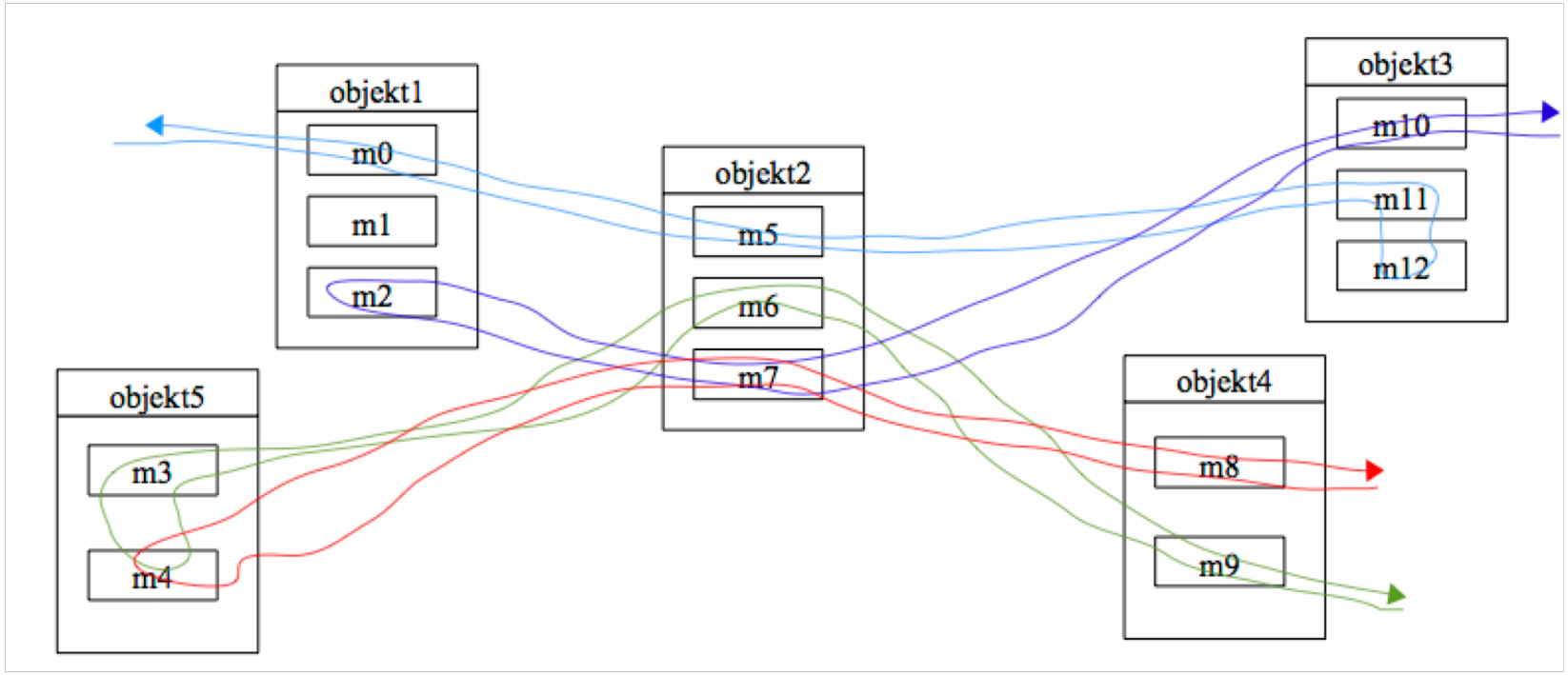


Motivering

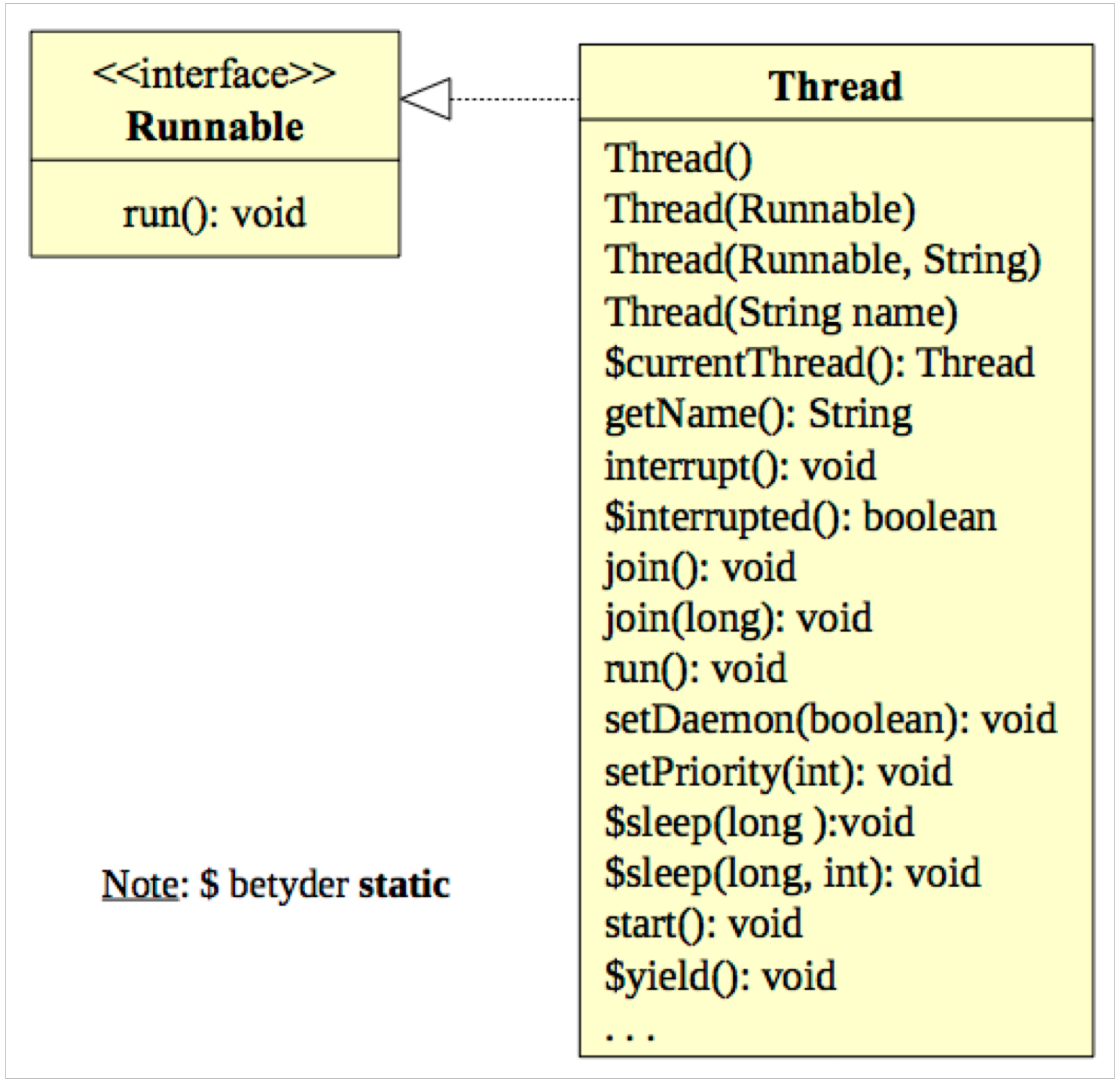
- Många program måste kunna hålla på med flera saker samtidigt:
  - bokningssystem av olika slag
  - en webbserver som måste kunna leverera flera webbsidor samtidigt
  - en grafisk applikation som ritar samtidigt som den arbetar med nästa bild
  - ett grafiskt användargränssnitt som utför beräkningar och samtidigt är redo att ha dialog med användaren
- Denna typ av program kallas samverkande program (concurrent programs).
- Vi skall här endast ge en introduktion till programmering med samverkande program, för mer djupgående studier hänvisas till kursen Parallell programmering.

Trådar

- I Java beskriver man aktiva objekt (och parallellism) med hjälp av standardklassen `Thread`. Aktiviteter som pågår samtidigt kallas därför i Java för *trådar*.



Klassen `Thread`



Trådar

- I ett parallellt program beskrivs varje aktivitet som en instans av klassen `Thread`. Det tråden skall utföra definieras genom att överskugga metoden `run()`.

```
public class SupporterThread extends Thread {
  private String team;

  public SupporterThread(String team) {
    this.team = team;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.print(team + " ");
    }
  }
}
```

Trådar

- För att starta exekveringen av en tråd anropar man metoden `start()`!
- Anropas `run()` direkt skapas ingen tråd.

```
public class WorldCup {
  public static void main(String[] args) {
    Thread kallaFan = new SupporterThread("Kalla");
    Thread wengFan = new SupporterThread("Weng");

    kallaFan.start();
    wengFan.start();
  }
}
```

- Quiz: vad blir utskriften?

Icke-deterministiskt beteende

- Sekventiella program säges vara *deterministiska* vilket betyder att:
  - Vilka operationer som utförs i programmet är en konsekvens av föregående operationer
  - Det är förutsägbart i vilken ordning operationerna i ett sekventiellt program kommer att utföras
  - Om ett program körs flera gånger med samma indata kommer samma resultat att erhållas varje gång
- Program med parallellism uppvisar ett *icke-deterministiskt beteende*:
  - I vilken ordning operationerna sker mellan de olika programflödena vet vi inte
  - Två programkörningar med samma indata kan ge olika resultat

Att programmera med trådar

- Mycket komplicerad, bara använd det när det behövs
- Svårt att felsöka/avlusa
- Inte garanterad snabbara än ett sekventiell program (växla mellan trådar kostar tid)
- Vi introducera bara bas kunskap, en fortsättningskurs behövs

Metoden `sleep()`

- En tråd kan temporärt avbryta sin exekvering med metoden `sleep()`, som är en *statisk metod* i klassen `Thread`.
- Metoden `sleep()` finns i följande två versioner:

```
public static sleep(long mills) throws InterruptedException
public static sleep(long mills, int nanos) throws InterruptedException
```

- Tiden som man vill avbryta trådens exekvering anges alltså i millisekunder respektive millisekunder och nanosekunder.
- Eftersom `sleep()` kan kasta en kontrollerad exceptionell händelse måste anropet av metoden ligga i ett `try-catch`-block.

Exempel

- På nästa slide har vi skrivit om föregående exempel på så sätt att vi låter tråden ”sova” med ett slumpmässigt tidsintervall mellan 0 och 1 sekund mellan varje utskrift.
- Vi nyttjar också att klassen `Thread` har en konstruktor

  ```
  public Thread(String name)
  ```

  som ger tråden namnet `name`, samt att detta namn kan avläsas med metoden `getName()`.

Exempel

```
public class SupporterThread extends Thread {
  public SupporterThread(String team) {
    super(team);
  }

  public void run() {
    for (int i = 1; i <= 10; i++) {
      System.out.println(i + " " + getName());
      try {
        Thread.sleep((int)(Math.random() * 1000));
      } catch (InterruptedException e) {}
    }
  }
}
```

Exempel

- Vi skriver ett testprogram som innehåller tre objekt av typen `SupporterThread`:

```
public class WorldCup {
  public static void main(String[] args) {
    Thread kallaFan = new SupporterThread("Kalla");
    Thread wengFan  = new SupporterThread("Weng");
    Thread holigan  = new SupporterThread("Ut med domare!");

    kallaFan.start();
    wengFan.start();
    holigan.start();
  }
}
```

En körning kan se ut enligt nedan:

1 Weng
1 Kalla
1 Ut med domare!
2 Kalla
2 Weng
3 Weng
4 Weng
2 Ut med domare!
3 Ut med domare!
…

Interfacet `Runnable`

- Java tillåter inte multipla implementationsarv, därför är det omöjligt för en klass som är subklass till `Thread` att ärva från någon annan klass. Aktiva objekt har många gånger behov av att ärva från andra klasser.
- Lösningen är att använda interfacet `Runnable`:

```
public interface Runnable {
  void run();
}
```

- Interfacet `Runnable`
  - har endast en metod `run()`
  - varje klass som implementerar `Runnable` måste implementera metoden `run()`
  - metoden `run()` i klassen som implementerar `Runnable` är utgångspunkten för exekveringen av en tråd.

Interfacet `Runnable`

- Ett `Runnable`-objekt abstraherar begreppet *ett jobb* (t.ex. ett recept).
- Ett `Thread`-objekt abstraherar begreppet *en arbetare* (t.ex. en kock).
- Hur kan vi associera ett jobb med en arbetare?
- Genom att klassen `Thread` har en konstruktor

```
public Thread(Runnable target)
```

- Konstruktorn skapar ett `Thread`-objekt som när det startas exekverar metoden `run()` i objekt `target`. Exempel:

```
Runnable recipe = . . .;
Thread chef = new Thread(recipe);
chef.start();
```

Exempel `Runnable`

- Vi skriver en klass `Writer` som har två instansvariabler `text` och `interval`. När ett objekt av klassen `Writer` exekveras skall texten `text` skrivas ut gång på gång med ett tidsintervall av `interval` sekunder mellan utskrifterna.

Exempel `Runnable`

```
public class Writer implements Runnable {
  private String text;
  private long interval;

  public Writer(String text, long time) {
    this.text = text;
    interval = time * 1000;
  }

  public void run() {
    while (!Thread.interrupted()) {
      try {
        Thread.sleep(interval);
      } catch (InterruptedException e) {
        break;
      }
      System.out.print(text + " ");
    }
  }
}
```

Exempel `Runnable`

- För att demonstrera hur flera objekt av klassen `Writer` kan exekvera parallellt ges nedan ett program som under en minut skriver ut ”Kalla" var 3:e sekund, ”Weng" var 4:e sekund och "Ut med domaren!" var 5:e sekund.

Exempel `Runnable`

```
public class TestWriter {
  public static void main(String[] arg) {
    Thread kallaFan = new Thread(new Writer("Kalla", 3));
    Thread wengFan  = new Thread(new Writer("Weng", 4));
    Thread huligan  = new Thread(new Writer("Ut med domaren!", 5));

    kallaFan.start();
    wengFan.start();
    huligan.start();

    try {
      Thread.sleep(60000);
      kallaFan.interrupt();
      wengFan.interrupt();
      huligan.interrupt();
    } catch (InterruptedException e) {
    }
  }
}
```

Avbryt trådarna efter 60 sekunder

Exempel `Runnable`

```
public class TestWriter2 {
  public static void main(String[] arg) {
    Thread kallaFan = new Thread(new Writer("Kalla", 3));
    Thread wengFan  = new Thread(new Writer("Weng", 4));
    Thread holigan  = new Thread(new Writer("Ut med domaren!", 5));
    kallaFan.setDaemon(true);
    wengFan.setDaemon(true);
    holigan.setDaemon(true);
    kallaFan.start();
    wengFan.start();
    holigan.start();
    try {
      Thread.sleep(60000);
    } catch (InterruptedException e) {
    }
  }
}
```

Sätt till demon-trådar. Dödas av JVM när det inte finns andra icke demon-trådar som exekverar.

Vänta på att tråden skall avsluta

- Vi utökar vår tidigare klass `WorldCup` enligt:

```
public class WorldCup {
  public static void main(String[] args) {
    Thread kallaFan = new SupporterThread("Kalla");
    Thread wengFan  = new SupporterThread("Weng");
    Thread holigan  = new SupporterThread("Ut med domare!");

    kallaFan.start();
    wengFan.start();
    holigan.start();

    System.out.println("Storma plan!");
  }
}
```

- Quiz: När stormas planen?

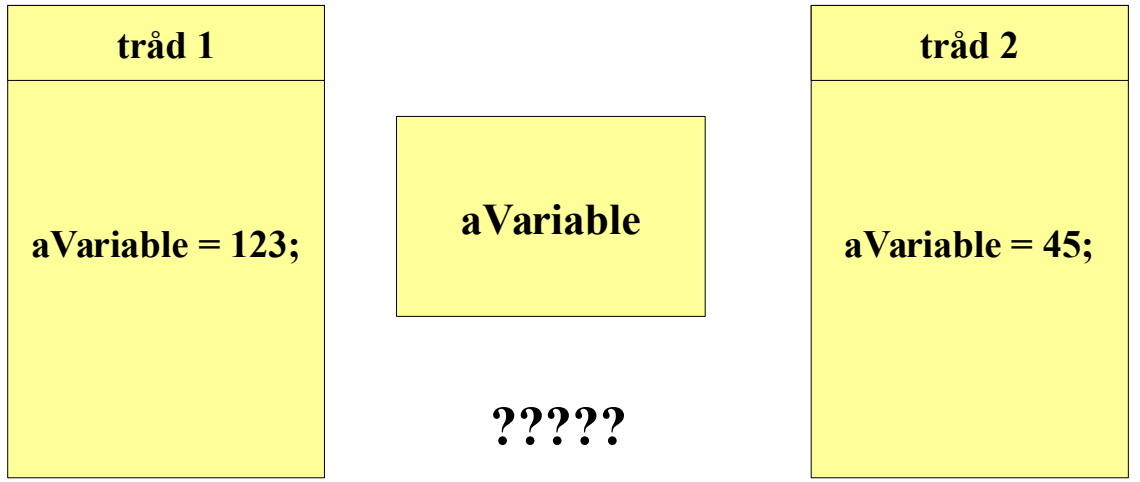
Vänta på att tråden skall avsluta

- För att vänta på att en tråd avslutas används metoden `join()`:

```
public class WorldCup {
  public static void main(String[] args) {
    Thread kallaFan = new SupporterThread("Kalla");
    Thread wengFan  = new SupporterThread("Weng");

    kallaFan.start();
    wengFan.start();

    try {
      kallaFan.join();
      wengFan.join();
    } catch (InterruptedException e) {}

    System.out.println("Storma plan!");
  }
}
```

Trådsäkerhet

- Då ett objekt modifieras kan det inta ett antal tillfälliga tillstånd som inte är konsistenta (d.v.s. ogiltiga).
- Om en tråd avbryts under en modifikation av ett objekt, så kan det lämna objektet i ett ogiltigt tillstånd.
- En klass säges vara trådsäker om den garanterar konsistens för sina objekt, även om det finns multipla trådar närvarande.



Kritiska sektioner

- Kodsegment som har access till samma objekt från olika separata trådar utgör en s.k. *kritisk sektion* (eng: *critical section*), och måste synkroniseras på så sätt att endast en tråd i taget får tillgång till objektet, annars kan objektet hamna i ett *inkonsistent tillstånd*.

Kritiska sektioner exempel

- Antag att vi har en klass för att handha bankkonton enligt nedan:

```
public class Account {
  private double balance;
  // ...
  public boolean withdraw(double amount) {
    if (amount <= balance) {
      double newBalance = balance - amount;
      balance = newBalance;
      return true;
    } else
      return false;
  }

  public void deposite(double amount) {
    balance = balance + amount;
  }
  // ...
}
```

Inkonsistent tillstånd

- Antag vidare att balansen på ett konto är 100000 kronor och att två uttag görs samtidigt vardera på 100000 kronor. Följande kan hända:

Balans	Uttag 1	Uttag 2
100000	amount<= balance	
100000		amount<= balance
100000	newBalance = balance - amount	
100000		newBalance = balance - amount
0	balance = newBalance	
0		balance = newBalance
0	**return true**	
0		**return true**

- Ett inkonsistent tillstånd på kontot har inträffat! Balansen borde vara -100000 men är 0!
- Vi har fått ***race condition***. En tråd försöker läsa data medan en annan tråd uppdaterar densamma.

Synkronisering

- För att kontot inte skall hamna i ett inkonsistent tillstånd måste metoden withdraw *synkroniseras*, dvs endast en tråd i taget skall kunna få tillgång till metoden.
- Java's lösning för att åstadkomma synkronisering:
  - varje objekt definieras som en *monitor*, vilken har ett singulärt lås
  - för att en tråd skall få tillträde till en kritisk sektion måste tråden först få tillgång till låset
  - att ha tillgång till låset innebär att ingen annan tråd kan ha tillgång till låset
  - låset återlämnas när tråden lämnar den kritiska sektionen.
- Allt detta görs automatiskt med konstruktionen `synchronized`.
- Programmerarens uppgift är att identifiera de kritiska sektionerna.

Synkroniserad version av `withdraw`

```
public class Account {
  private double balance;
  // ...
  public synchronized boolean withdraw(double amount) {
    if (amount <= balance) {
      double newBalance = balance - amount;
      balance = newBalance;
      return true;
    } else
      return false;
  }

  public void deposite(double amount) {
    balance = balance + amount;
  }
  // ...
}
```

Konsistent tillstånd

- Om vi nu återigen antar att balansen på kontot är 100000 kronor och att två uttag görs samtidigt på 100000 kronor händer följande:

Balans	**Uttag 1**	**Uttag 2**
100000	amount<= balance	
100000	newBalance = balance - amount	
0	balance = newBalance	
0	**return true**	
0		amount<= balance
0		**return false**

Ömsesidig uteslutning

- För att göra klassen `Account` trådsäker räcker det inte att enbart metoden `withdraw` är synkroniserad, utan även metoden `deposite` måste synkroniseras.
- Detta på grund av att båda metoderna konkurrerar om att förändra tillståndet på variabeln `balance`.

Ömsesidg uteslutning

```
public class Account {
  private double balance;
  // ...
  public synchronized boolean withdraw(double amount) {
    if (amount <= balance) {
      double newBalance = balance - amount;
      balance = newBalance;
      return true;
    } else
      return false;
  }

  public synchronized void deposite(double amount) {
    balance = balance + amount;
  }
  // ...
}
```

ac.withdraw stänger ute ytterligare en ac.withdraw
ac.withdraw stänger ute ac.deposit
ac.deposit stänger ute ytterligare en ac.deposit
ac.deposit stänger ute ac.withdraw
men ac1.withdraw stänger inte ute ac2.withdraw, etc.

Låset ägs av tråden

- Låset ägs av tråden, vilket förhindrar att en tråd blir blockerad av ett lås som tråden redan har:

```
public class ChainExample {
  public synchronized void methodA() {
    //do something methodB();
    //do something more
  }
  public synchronized void methodB() {
    //do something
  }
}
```

- I och med att tråden är i besittning av låset när metoden `methodA` börjar exekvera och låset släpps först när exekveringen av metoden är klar är tråden i besittning av låset när den synkroniserade metoden `methodB` anropas.

Synkronisering av satser

- Om alla metoder är synkroniserade i ett objekt kan endast en tråd åt gången använda objektet.
- För lite synkronisering
  - konflikter mellan trådar
  - inkonsistens på grund av tillgång till samma resurs (race condition)
- För mycket synkronisering
  - trådarna får vänta på varandra, ingen parallellism

Synkronisering av satser

- Det är möjligt att synkronisera enskilda satser med konstruktionen:

```
synchronized (obj) {
  <statements>
}
```

  där `obj` är en godtycklig objektreferens.
- För att att kunna exekvera satserna i `<statements>` måste tråden äga låset till objektet `obj`.

```
public static void aMethod(int[] n) {
  // ...
  synchronized(n) {
    for (int = 0; i < n.length; i = i + 1) {
      if(n[i] > 100) { save(n[i]);
      }
    }
  }
  // ...
}
```

Synkronisering av satser

- Om satser som berör en primitiv variabel skall synkroniseras, associeras den primitiva variabeln med ett *lås-objekt*.

```
public class PrimaryTypeSynchronization {
  private int valueA;
  private double valueB;
  private Object lockA = new Object();
  private Object lockB = new Object();
  public void setA(int value) {
    synchronized(lockA) {
      valueA = value;
    }
  }
  public double getA() {
    synchronized(lockA) {
      return valueA;
    }
  }
  public void setB(double value) {
    synchronized(lockB) {
      valueB = value;
    }
  }
  public double getB() {
    synchronized(lockB) {
      return valueB;
    }
  }
}
```

Samarbete mellan trådar

- Synkronisering handlar om *uteslutning* inte samarbete.
- Samarbete innebär:
  - vänta: när en tråd inte kan fortsätta, så låt andra trådar fortsätta
  - meddela: väck upp sovande trådar om det händer något som dessa kanske väntar på
- Metoden `wait()`
  - suspenderar tråden och öppnar låset för objektet
  - andra trådar får möjlighet att exekvera synkroniserade metoder
  - tråden väcks när någon tråd anropar `notifyAll` för objektet
- Metoden `notifyAll()`
  - signalerar till andra trådar som väntar på att objektet skall vakna
- Anm: Det finns även en metod `notify()`, men den bör inte användas.

Live code

- `DivideAndConquer`

What’s next

Block 7-2:
Sammanfattning och tentameninfo