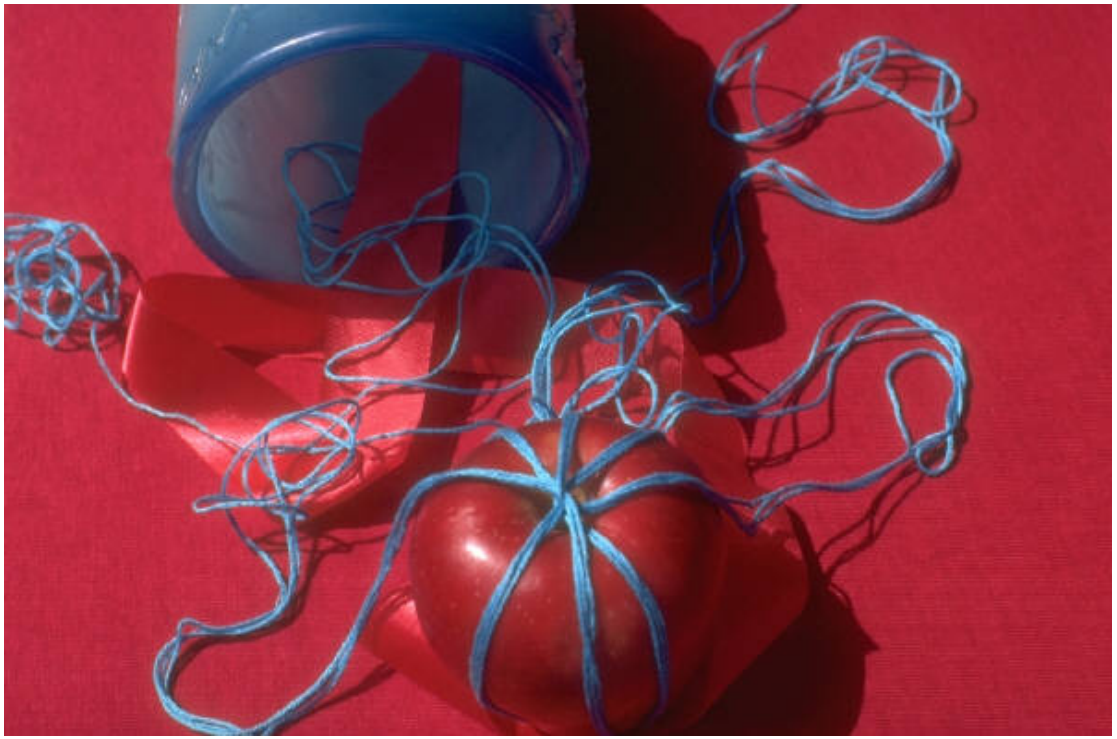
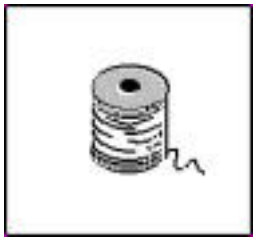
Tråde



Plan

- Trådbegrebet
- Synkronisering
- Koordinering
- Eksempel: et flertrådet spil

Trådbegrebet



En **tråd** er et sekventielt forløb i et program.

Java tillader flere tråde at eksistere på samme tid.

Tråde kan enten afvikles på en flerprocessor-maskine, eller (mere normalt) i *simuleret parallel* ved brug af *tidsdeling*.

Tråde

Fordele:

- Tillader hurtig reaktion på brugerinput
  Muliggør udvikling af *reaktive* systemer
- Gør det muligt for en server at servicere flere klienter samtidigt

Komplikationer:

- Afbrydelse af en tråd kan efterlade et objekt i en inkonsistent tilstand (*safety problem*)
- En tråd kan blokere andre tråde (*liveness problem*)

Skabelse af tråde

Tråde kan skabes og erklæres på to måder:

(1) ved nedarvning fra klassen `Thread`

(2) ved implementation af grænsefladen `Runnable`

Nedarvning fra `Thread`

```
public class MyThread extends Thread {
    public void run() {
        // the thread body
    }

    // other methods and fields
}
```

Start af en tråd:

```
new MyThread().start();
```

Eksempel

```
public class Counter1 extends Thread {
    protected int count, inc, delay;

    public Counter1(int init, int inc, int delay) {
        this.count = init; this.inc = inc; this.delay = delay;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(count + " ");
                count += inc;
                sleep(delay);
            }
        } catch (InterruptedException e) {}
    }

    public static void main(String[] args) {
        new Counter1(0, 1, 33).start();
        new Counter1(0, -1, 100).start();
    }
}
```

Kørselsresultat

```
0 0 1 -1 2 -2 3 4 5 -3 6 -4 7 -5 8 9 -6 10 11 -7 12 13 -8
14 15 -9 16 17 -10 18 19 -11 20 21 -12 22 23 -13 24 25 -14
26 27 28 -15 29 30 -16 31 32 -17 33 34 35 -18 36 37 -19 38
39 40 -20 41 42 -21 43 44 -22 45 46 47 -23 48 49 50 -24 51
52 53 -25 54 55 56 -26 57 58 59 -27 60 61 62 -28 63 64 65
-29 66 67 68 -30 69 70 71 -31 72 73 74 -32 75 76 77 -33 78
79 80 -34 81 82 -35 83 84 -36 85 86 87 -37 88 89 90 -38 91
92 93 -39 94 95 96 -40 97 98 99 -41 100 101 102 -42 103 104
...
```

Implementering af `Runnable`

```
public class MyThread extends AnotherClass
                      implements Runnable {
    public void run() {
        // the thread body
    }

    // other methods and fields
}
```

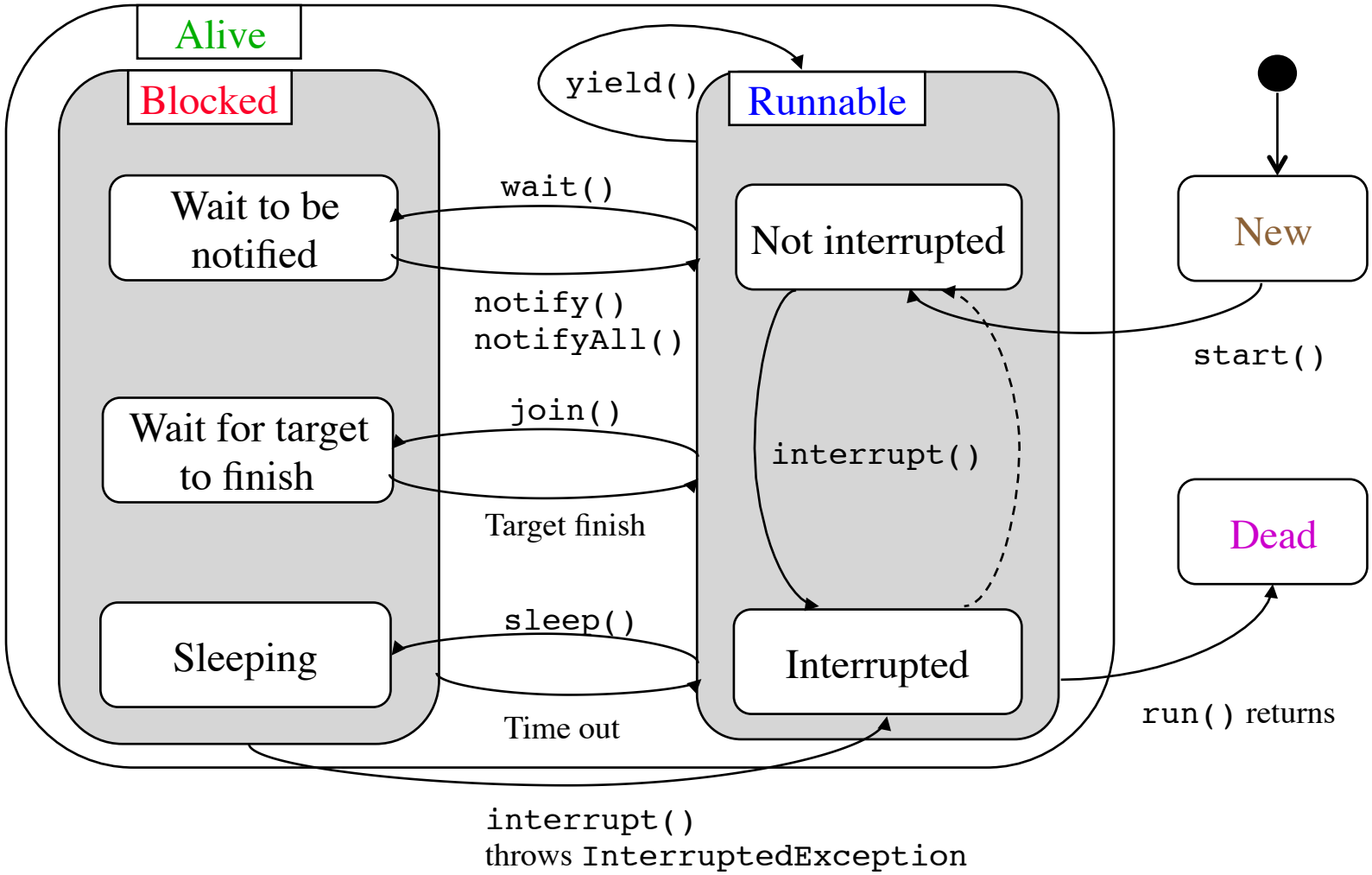
Start af en tråd:

```
new Thread(new MyThread()).start();
```

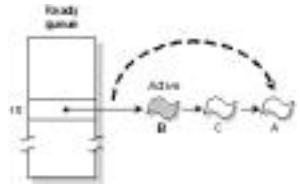
Eksempel

```
public class Counter2 implements Runnable {
    protected int count, inc, delay;

    public Counter2(int init, int inc, int delay) {
        this.count = init; this.inc = inc; this.delay = delay;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(count + " ");
                count += inc;
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {}
    }

    public static void main(String[] args) {
        new Thread(new Counter2(0, 1, 33)).start();
        new Thread(new Counter2(0, -1, 100)).start();
    }
}
```

En tråds livscyklus



Prioritering af tråde

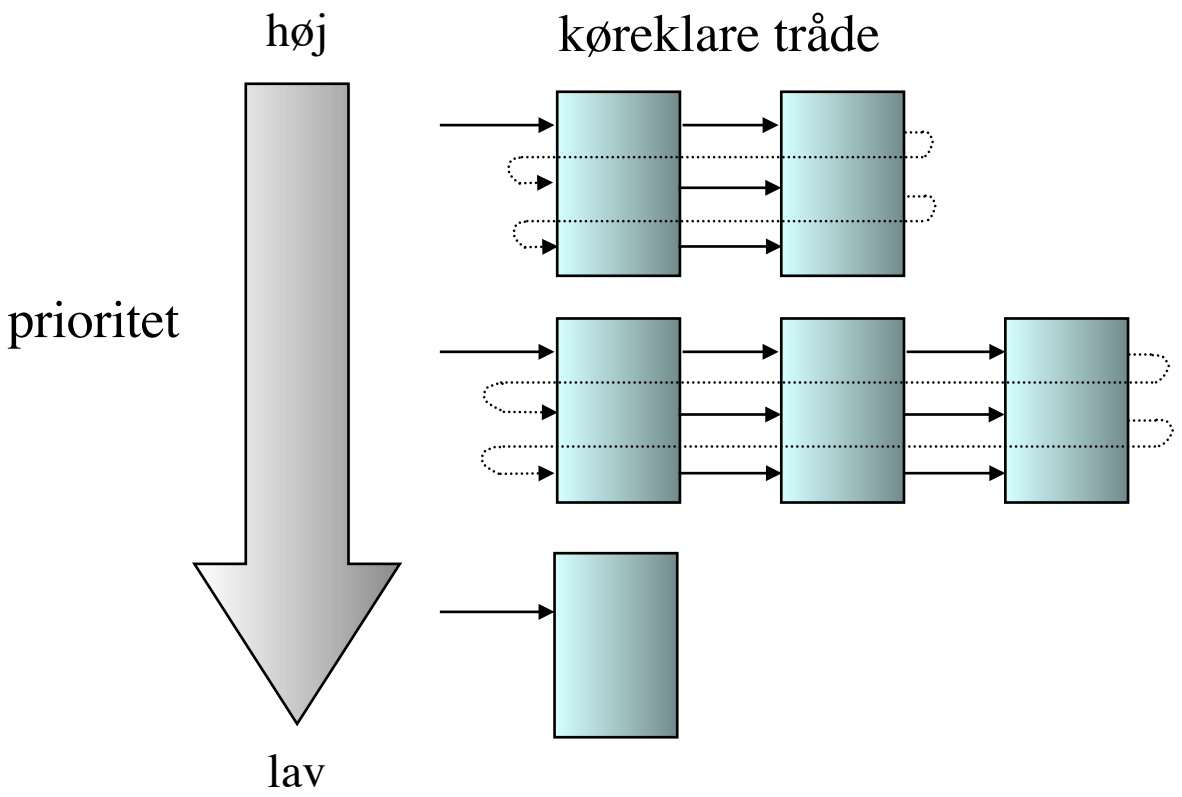


Enhver tråd er forsynet med en **prioritet** (et heltal imellem 1 og 10).

Prioriteter tildeles ved skabelsen. Som standard bliver prioriteten sat til prioriteten for den tråd, der skaber tråden. Prioriteter kan ændres dynamisk.

JVM vælger vilkårligt blandt de kørbare tråde, der har højst prioritet. Således vil tråde med en høj prioritet fortrænge (preempt) tråde med en lavere prioritet.

Afvikling af prioriterede tråde



Retningslinjer for design

Tråde, som skal reagere hurtigt på brugerinput, bør tildeles en høj prioritet.

Tråde med høj prioritet bør ikke varetage opgaver med lang udførelsestid.

Anvend kun prioriteter til at effektivisere et program. Et programs korrekthed må ikke afhænge af de involverede trådes prioriteter.

Synkronisering

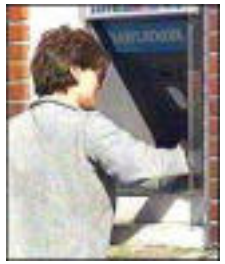
Mens et objekt bliver modificeret, kan det være i en inkonsistent tilstand.

Det er vigtigt at sikre, at andre tråde ikke tilgår objektet i denne situation.

Eksempel


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

    private long balance;
}
```

Implementeringen er korrekt ved kun én tråd; men ikke for flere tråde!



Race hazard

Balance	Withdrawal1	Widthdrawal2
1000000	`widthdraw(1000000)`	`widthdraw(1000000)`
1000000	`amount <= balance`	
1000000		`amount <= balance`
1000000	`newBalance = ...;`	
1000000		`newBalance =  ...;`
0	`balance = ...;`	
0		`balance = ...;`
0	`return true;`	
0		`return true;`

Klassen `Account` er ikke **trådsikker**

Atomiske operationer

Java garanterer, at læsning og skrivning af primitive typer - med undtagelse af `long` og `double` - sker atomisk.

Alle andre operationer må synkroniseres eksplicit for at sikre atomicitet.

Kritiske regioner

En **kritisk region** er et område af et program, som kun kan udføres af én tråd ad gangen.

Java anvender begrebet *synkronisering* til at specificere kritiske regioner i et program.

Synkronisering kan foretages på en metode eller på en blok af sætninger.

Synkronisering på metoder

```
class MyClass {
    synchronized void aMethod() {
        «do something»
    }
}
```

Hele metodekroppen er en kritisk region.

Eksempel med synkronisering


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

    private long balance;
}
```

Implementeringen er nu korrekt ved for flere tråde.

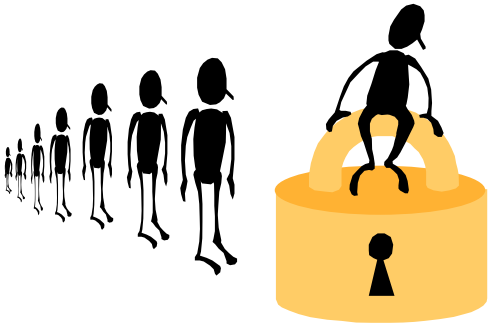
Synkronisering på sætningsblokke

```
synchronized(expr) {
    «do something»
}
```

hvor `expr` er et udtryk af referencetype.

Sætningerne i blokken udgør en kritisk region.

Låse



Synkronisering er implementeret ved at forsyne ethvert objekt med en **lås**.

En tråd må være i eksklusiv besiddelse af en lås, før den går ind i en kritisk region.

- For en synkroniseret instansmetode benyttes den lås, der er tilknyttet `this`.

- For en synkroniseret sætningsblok benyttes den lås, der er tilknyttet objektet givet ved `expr`.

Frigivelse af låse

En lås frigives, når en tråd forlader en kritisk region.

Midlertidig frigivelse kan ske ved kald af metoden `wait`.

Visualisering af låsemekanismen ved hjælp af en telefonboks

Et objekt er en telefonboks med plads til én person.
Trådene er personer, der ønsker at benytte telefonen.

Synkronisering af kollektioner

Alle Collections-implementationer er usynkroniserede.

Synkroniseret tilgang til en kollektion kan opnås ved brug af en *synkroniseret wrapper-klasse*.

Eksempel:

```
Map unsyncMap = new HashMap();
Map syncMap = Collections.synchronizedMap(unsyncMap);
```

Synkronisering af statiske metoder

Statiske metoder kan synkroniseres.

```
class C {
    synchronized static void method() { ... }
}
```

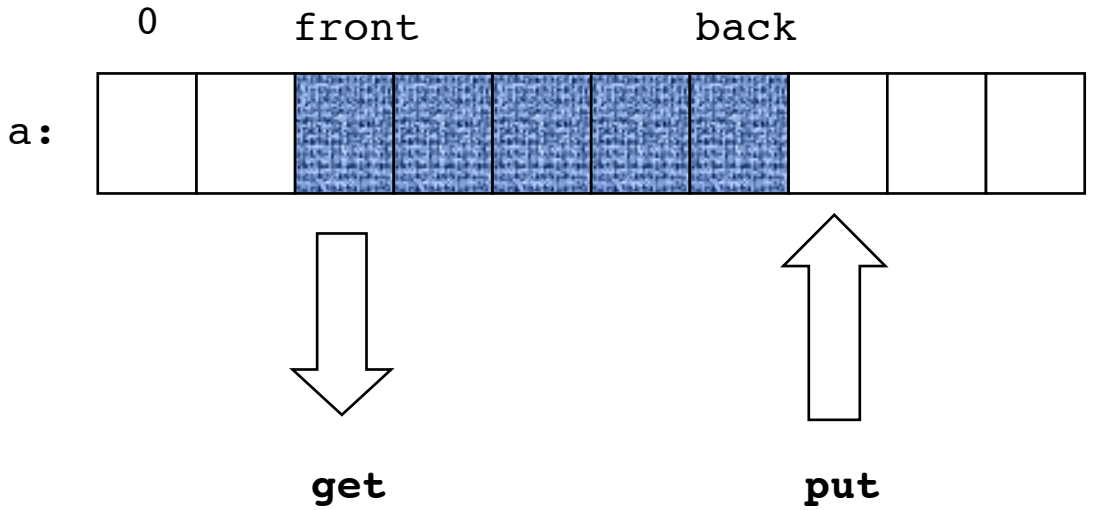
Der anvendes den lås, der er knyttet til klassens `Class`-objekt.

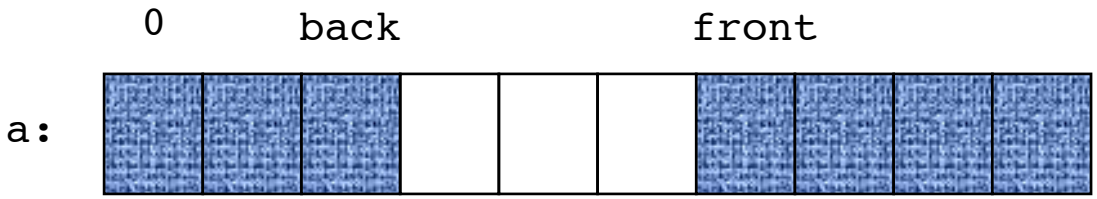
Eksempler på synkroniserede sætningsblokke:

```
synchronized(getClass()) { ... }
synchronized(C.class) { ... }
```

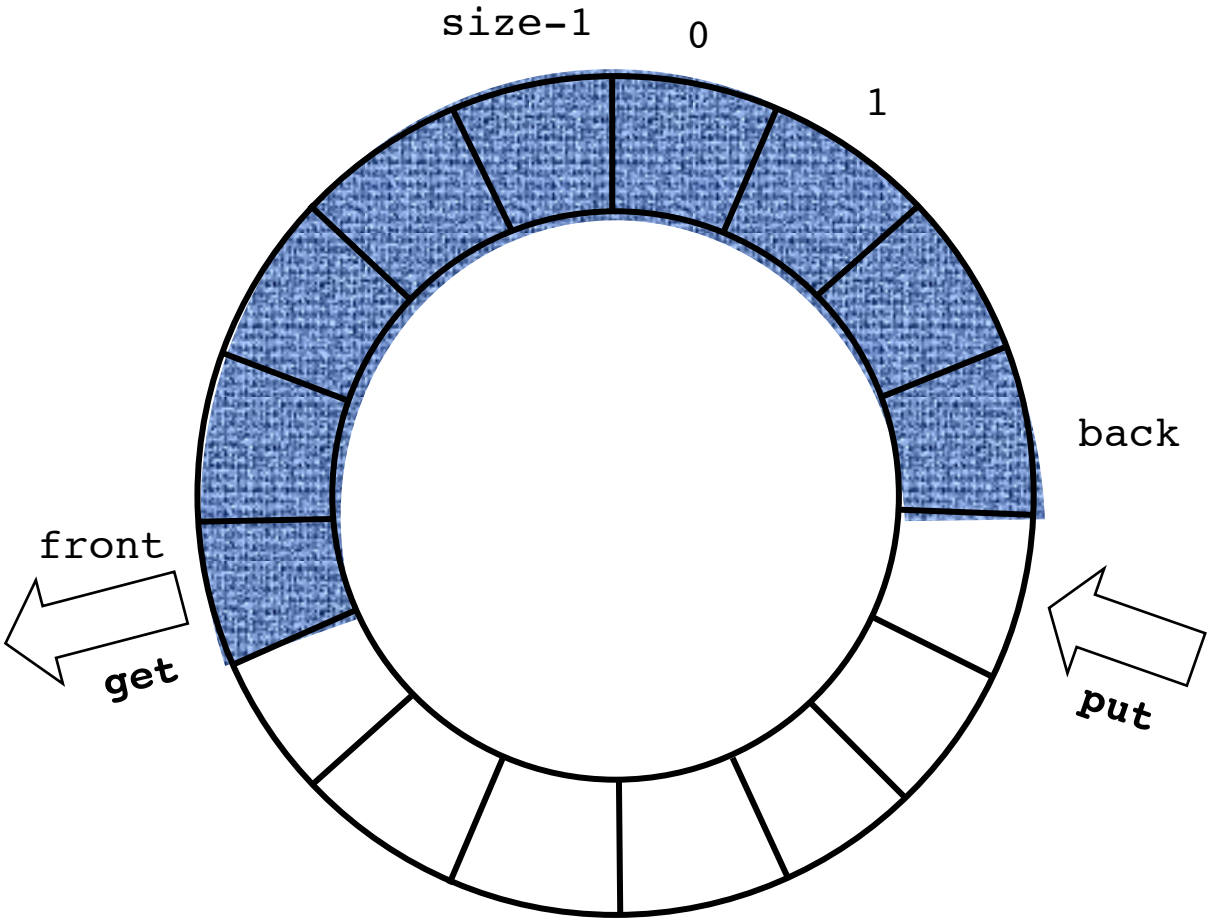
En begrænset kø

(implementeret med et cirkulært array)





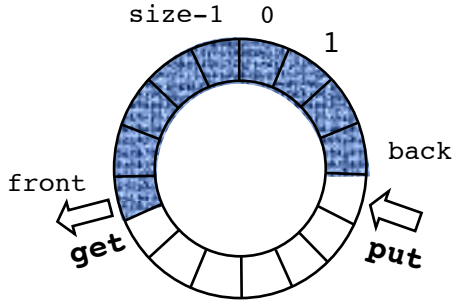
Cirkulært array



En ikke-trådsikker begrænset kø

```
public class BoundedQueue {
    protected Object[] a;
    protected int front, back, size, count;

    public BoundedQueue(int size) {
        if (size > 0) {
            this.size = size;
            a = new Object[size];
            back = size - 1;
        }
    }

    public boolean isEmpty() { return count == 0; }
    public boolean isFull() { return count == size; }
    public int getCount() { return count; }

    // put, get
}
```

fortsættes




```
public void put(Object obj) {
    if (obj != null && !isFull()) {
        back = (back + 1) % size;
        a[back] = obj;
        count++;
    }
}
```

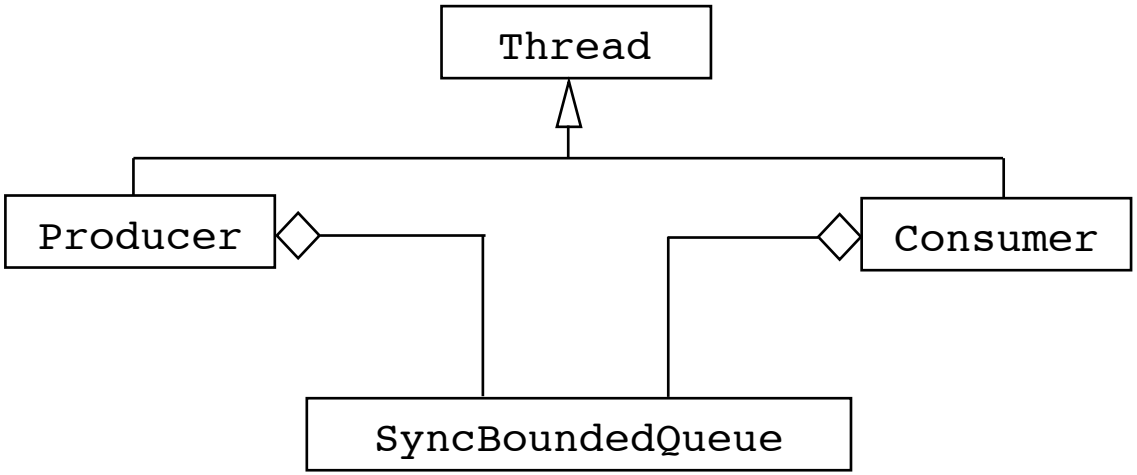
```
public Object get() {
    if (!isEmpty()) {
        Object result = a[front];
        a[front] = null;
        front = (front + 1) % size;
        count--;
        return result;
    }
    return null;
}
```

En trådsikker begrænset kø

```
public class SyncBoundedQueue extends BoundedQueue {
    public SyncBoundedQueue(int size) { super(size); }

    public synchronized boolean isEmpty() { return super.isEmpty(); }
    public synchronized boolean isFull() { return super.isFull(); }
    public synchronized int getCount() { return super.getCount(); }
    public synchronized void put(Object obj) { super.put(obj); }
    public synchronized Object get() { return super.get(); }
}
```

Eksempel på anvendelse

En trådsikker begrænset kø kan anvendes som en buffer imellem en producent og en forbruger, der begge er tråde.



Klassen `Producer`

```
public class Producer extends Thread {
    protected BoundedQueue queue;
    protected int n;

    public Producer(BoundedQueue queue, int n) {
        this.queue = queue; this.n = n;
    }

    public void run() {
        for (int i = 0; i < n; i++) {
            queue.put(new Integer(i));
            System.out.println("produce: " + i);
            try {
               sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {}
        }
    }
}
```

Klassen `Consumer`

```
public class Consumer extends Thread {
    protected BoundedQueue queue;
    protected int n;

    public Consumer(BoundedQueue queue, int n) {
        this.queue = queue; this.n = n;
    }

    public void run() {
        for (int i = 0; i < n; i++) {
            Object obj = queue.get();
            if (obj != null)
                System.out.println("\tconsume: " + obj);
            try {
                sleep((int)(Math.random() * 400));
            } catch (InterruptedException e) {}
        }
    }
}
```

Testprogram

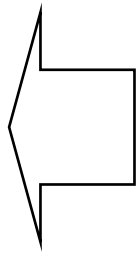
```
public static void main(String args[]) {
    BoundedQueue queue = new SyncBoundedQueue(5);
    new Producer(queue, 15).start();
    new Consumer(queue, 10).start();
}
```

Enhver applikation starter med en **brugertråd**, der udfører `main`. En applikation terminerer, når alle brugertråde er temineret, eller hvis metoden `exit` fra `System` eller `Runtime` kaldes.

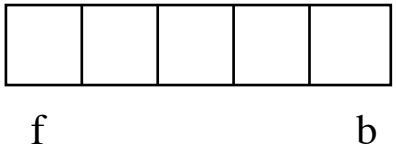
Kørselsresultat

```
produce: 0
        consume: 0
produce: 1
produce: 2
produce: 3
produce: 4
        consume: 1
produce: 5
produce: 6
produce: 7
        consume: 2
produce: 8
produce: 9
        consume: 3
```

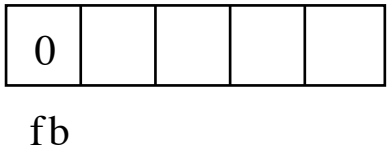
```
produce: 10
produce: 11
produce: 12
        consume: 4
produce: 13
produce: 14
        consume: 5
        consume: 6
        consume: 8
        consume: 10
        consume: 13
```



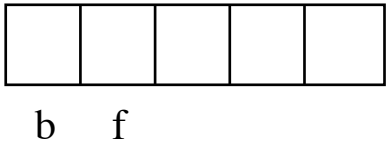
Forbrugeren kan **ikke** følge med producenten
(produkter går tabt)



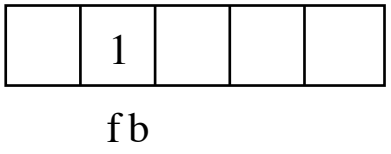
put(0)



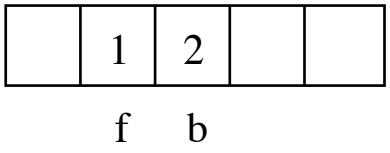
get = 0



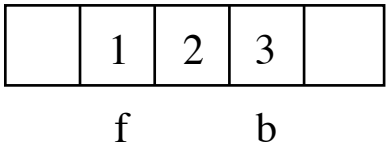
put(1)



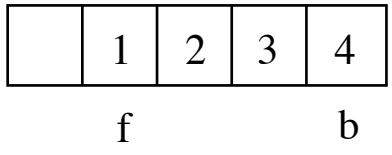
put(2)



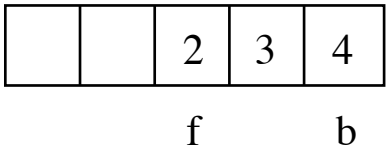
put(3)



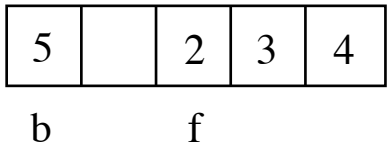
put(4)



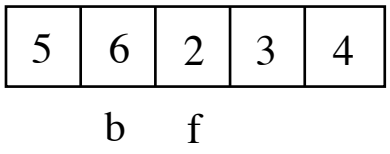
get = 1



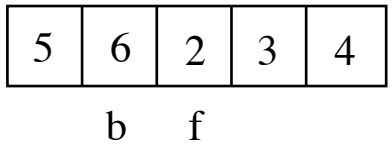
put(5)



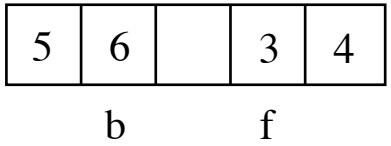
put(6)



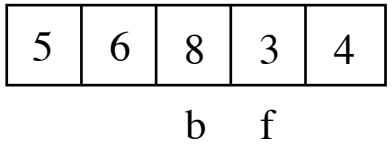
put(7)



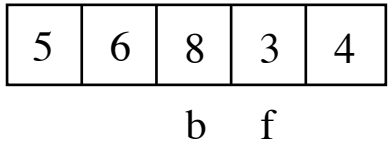
get = 2



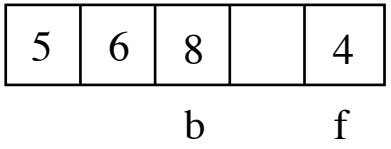
put(8)



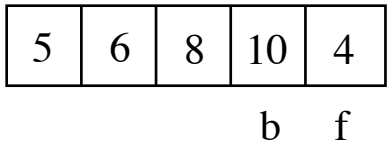
put(9)



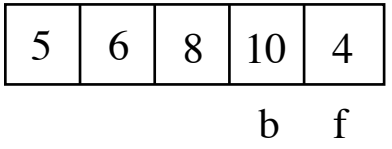
get = 3



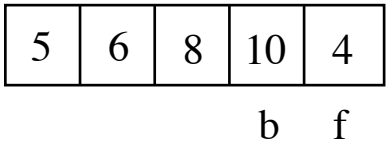
put(10)



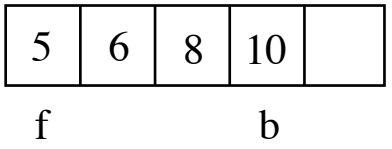
put(11)



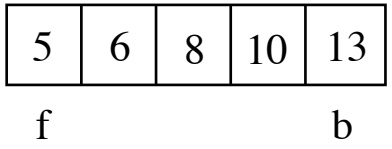
put(12)

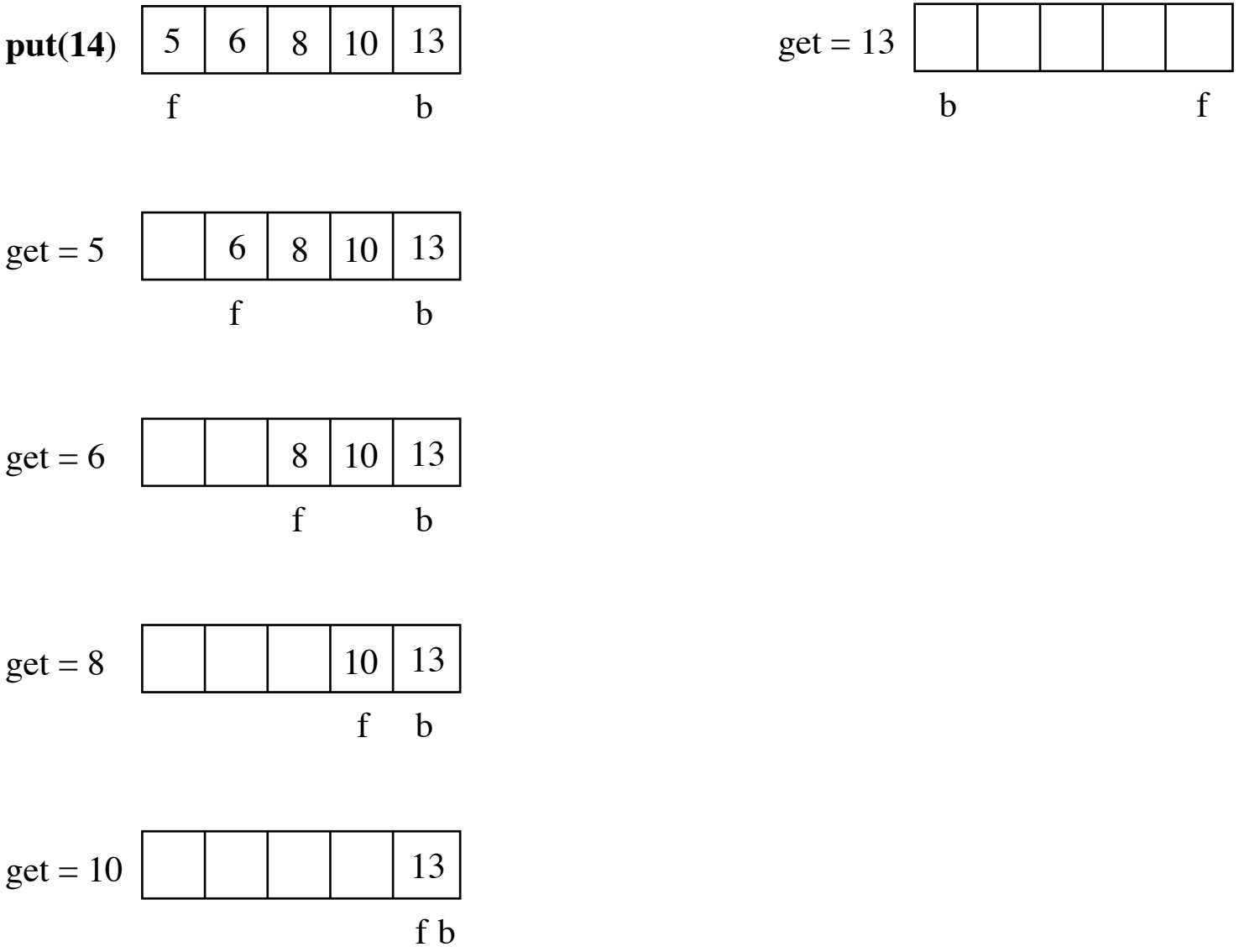


get = 4



put(13)




put(14)
5 6 8 10 13
f b
get = 5
6 8 10 13
f b
get = 6
8 10 13
f b
get = 8
10 13
f b
get = 10
13
f b
get = 13
b f

Koordinering
(samarbejde imellem tråde)

Synkronisering benyttes til gensidig udelukkelse fra kritiske regioner.

Men der også behov for, at tråde kan koordinere deres arbejde. Hertil benyttes **bevogtet suspendering** (guarded suspension).

En tråd kan bringes til at vente midlertidigt, indtil en given betingelse (guard) bliver opfyldt.

`wait`, `notify` og `notifyAll`

Bevogtet suspendering implementeres ved hjælp af metoderne `wait()`, `notify()` og `notifyAll()` fra klassen `Object`.

Metoden `wait` kaldes, når en tråd midlertidigt skal vente, enten fordi den selv ikke er i stand til at fortsætte, eller fordi andre tråde skal have lejlighed til det.

En af metoderne `notify` og `notifyAll` kaldes, når en tråd skal underrette ventende tråde om, at de gerne må fortsætte (hvis de kan).

`wait`

- Skal forekomme i en synkroniseret metode eller blok
- Bringer tråden i en blokeret tilstand, hvor den venter på et `notify`-signal
- Frigiver låsen, som den synkroniserede kode benytter
- Beslaglægger igen låsen, når `wait` returnerer
- Varianterne `wait(long millis)` og `wait(long millis, int nanos)` venter højst i et tidsrum, der er angivet af parametrene

`notify`

- Skal forekomme i en synkroniseret metode eller blok
- Vækker højst én tråd, som venter
- Varianten `notifyAll()` giver alle ventende tråde en chance for at komme videre. Det lykkes imidlertid for højst én ventende tråd

Koordinering imellem producent og forbruger

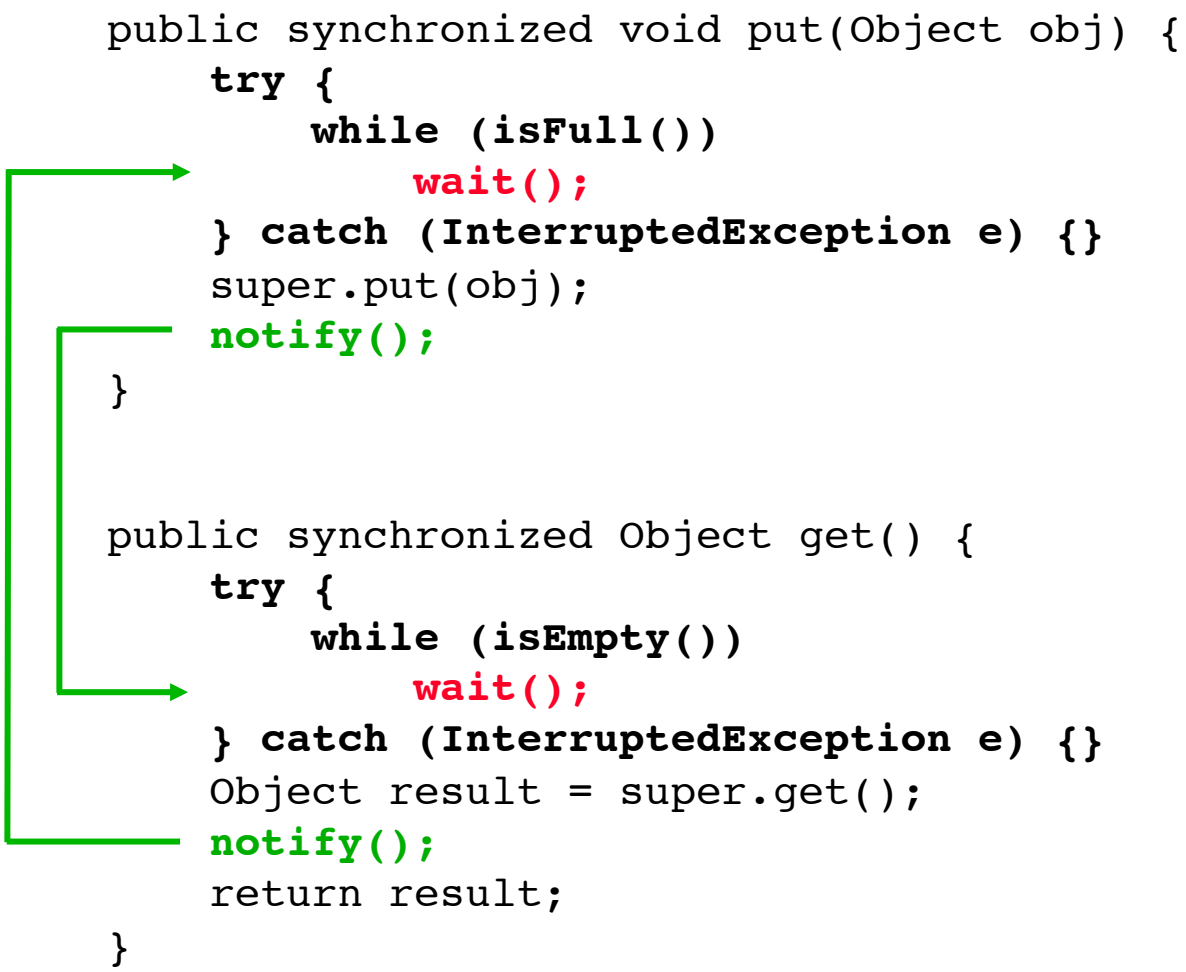
Hvis producenten forsøger at lægge et nyt element i køen, når køen er fuld, skal han vente, indtil forbrugeren har taget et element fra køen (så der bliver plads til det nye element).

Hvis forbrugeren forsøger at tage et element fra køen, når køen er tom, skal han vente, indtil producenten har lagt et element i køen.

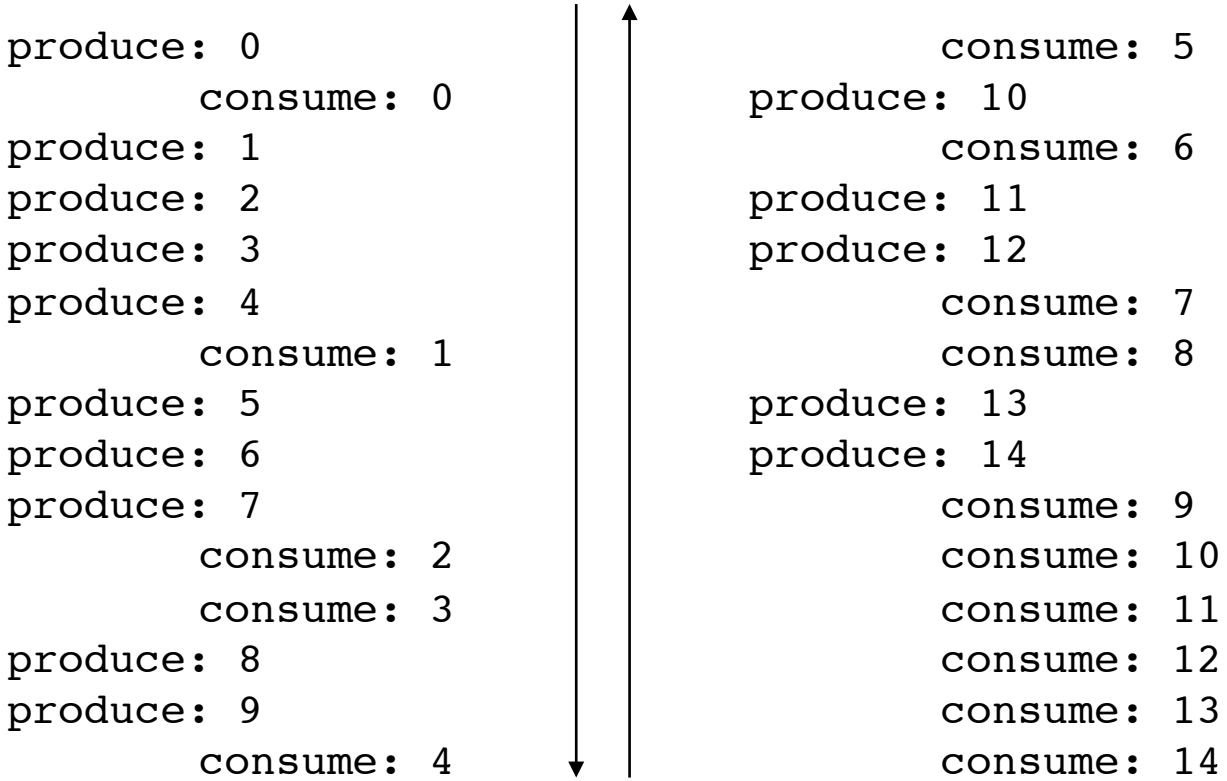
En trådsikker begrænset kø med bevogtet suspendering

```
public class BoundedQueueWithGuard extends BoundedQueue {
    public BoundedQueueWithGuard(int size) { super(size); }

    public synchronized boolean isEmpty() { return super.isEmpty(); }
    public synchronized boolean isFull() { return super.isFull(); }
    public synchronized int getCount() { return super.getCount(); }

    public synchronized void put(Object obj) { ... }
    public synchronized Object get() { ... }

    public static void main(String args[]) {
        BoundedQueueWithGuard queue = new BoundedQueueWithGuard(5);
        new Producer(queue, 15).start();
        new Consumer(queue, 10).start();
    }
}
```

fortsættes


public synchronized void put(Object obj) {
try {
while (isFull())
wait();
} catch (InterruptedException e) {}
super.put(obj);
notify();
}
public synchronized Object get() {
try {
while (isEmpty())
wait();
} catch (InterruptedException e) {}
Object result = super.get();
notify();
return result;
}

Kørselsresultat



Ingen produkter går tabt

Trådsikre køer i Java 5.0

```
package java.util.concurrent.*;

public interface BlockingQueue<E> extends java.util.Queue<E> {
    void put(E o) throws InterruptedException;
    E take() throws InterruptedException;

    boolean offer(E o, long timeout, TimeUnit unit)
        throws InterruptedException;
    E poll(long timeout, TimeUnit unit)
        throws InterruptedException;

    int remainingCapacity();
}
```

```
public class ArrayBlockingQueue<E> implements BlockingQueue<E>;
public class LinkedBlockingQueue<E> implements BlockingQueue<E>;
public class PriorityBlockingQueue<E> implements BlockingQueue<E>;
```

Retningslinjer for design

Metoden `wait` bør altid kaldes i en løkke; ikke i en `if`-sætning. At blive vækket ved `notify` er nemlig ikke ensbetydende med, at ventebetingelsen er opfyldt.

Når et synkroniseret objekt skifter tilstand, bør det normalt kalde metoden `notifyAll`. Derved får alle ventende tråde chancen for at checke, om der er sket et tilstandsskift, som muliggør deres genoptagelse.

Problemer omkring koordinering

- Udsultning (starvation)
- Dvale (dormancy)
- Baglås (deadlock)
- For tidlig terminering (premature termination)

Udsultning

En tråd får aldrig chancen for at køre.

Sker hvis der altid findes tråde med en højere prioritet, eller en tråd med samme prioritet ikke frigiver processoren (f. eks. ved kald af `sleep` eller `yield`).

Undgå “busy waiting".

```
while (x < 2)
    ;
```

Dvale

En blokeret tråd bliver aldrig kørbar.

Sker hvis en tråd, der har kaldt `wait`, ikke bliver notificeret.

Når man er i tvivl, bør `notifyAll` kaldes fremfor `notify`.

Baglås

Når to eller flere tråde blokerer for hinanden.

Kan ske, hvis flere tråde konkurrerer om de samme to eller flere ressourcer og vil være i eksklusiv besiddelse af disse på samme tid.

```
synchronized(a) {
    synchronized(b) {
        ...
   }
}
```

thread1

```
synchronized(b) {
    synchronized(a) {
        ...
   }
}
```

thread2

En klasse med risiko for baglås

```
public class DiskDrive {
    public synchronized InputStream openFile(String fileName) {
        ...
    }

    public synchronized void writeFile(String fileName,
                                       InputStream in) {
        ...
    }

    public synchronized void copy(DiskDrive destination,
                                  String fileName) {
        InputStream in = openFile(fileName);
        destination.writeFile(fileName, in);
    }
}
```

Opstået af baglås

thread1: `c.copy(d, file1)`

Kald `c.copy(...)`

Får låsen for `c`

Kald `c.openFile(...)`

Kald `d.writeFile(...)`

Kan ikke få låsen for `d`

thread2: `d.copy(c, file2)`

Kald `d.copy(...)`

Får låsen for `d`

Kald `d.openFile(...)`

Kald `c.writeFile(...)`

Kan ikke få låsen for `c`

Baglås!

Undgåelse af baglås

Køretidssystemet er hverken i stand til at opdage eller undgå baglås. Det er programmørens opgave at sikre, at der aldrig kan opstå baglås.

En ofte anvendt teknik er **ressource-ordning**:

```
synchronized(a) {
    synchronized(b) {
        ...
   }
}
```

thread1

```
synchronized(a) {
    synchronized(b) {
        ...
   }
}
```

thread2

Baglås kan ikke opstå

For tidlig terminering

Når en tråd terminerer, før den skal, og dermed hindrer, at andre tråde ikke kan komme videre.

Klassen `Thread`

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


    public static final int MIN_PRIORITY = 1;
    public static final int NORM_PRIORITY = 5;
    public static final int MAX_PRIORITY = 10;
```

fortsættes

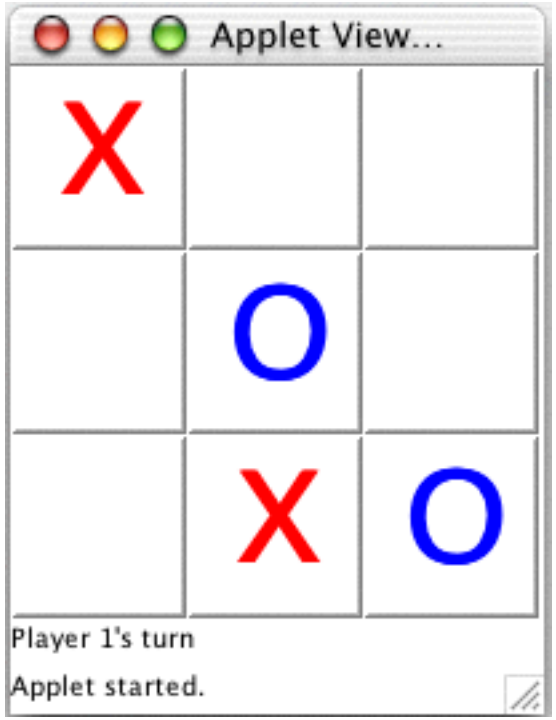
```
public static int activeCount();
public static void dumpStack();
public static boolean interrupted();
public static native void sleep(long millis)
                          throws InterruptedException;
public static native void sleep(long millis, int nanos)
                          throws InterruptedException;
public static native void yield();

public final String getName();
public final int getPriority();
public final ThreadGroup getThreadGroup();
public void interrupt();
public final native boolean isAlive();
public final native boolean isDaemon();
public final native boolean isInterrupted();
```

fortsættes

```
    public final synchronized void join()
                                        throws InterruptedException;
    public final synchronized void join(long millis)
                                        throws InterruptedException;
    public final synchronized void join(long millis, int nanos)
                                        throws InterruptedException;
    public void run();
    public final void setDaemon(boolean on);
    public final void setName(String name);
    public final void setPriority(int newPriority);
    public String toString();
}
```

Metoderne `stop`, `suspend`, `resume` og `destroy` er "deprecated" (misbilligede) - på grund af deres risiko for at forårsage baglås.

Et flertrådet kryds-og-bolle-spil
(Tic-tac-toe)



Spillere

To typer:

- En menneskespiller, der laver træk ved at klikke med musen på brættet

- En maskinspiller, der automatisk genererer træk (ikke nødvendigvis gode)

Spillet kan spilles af

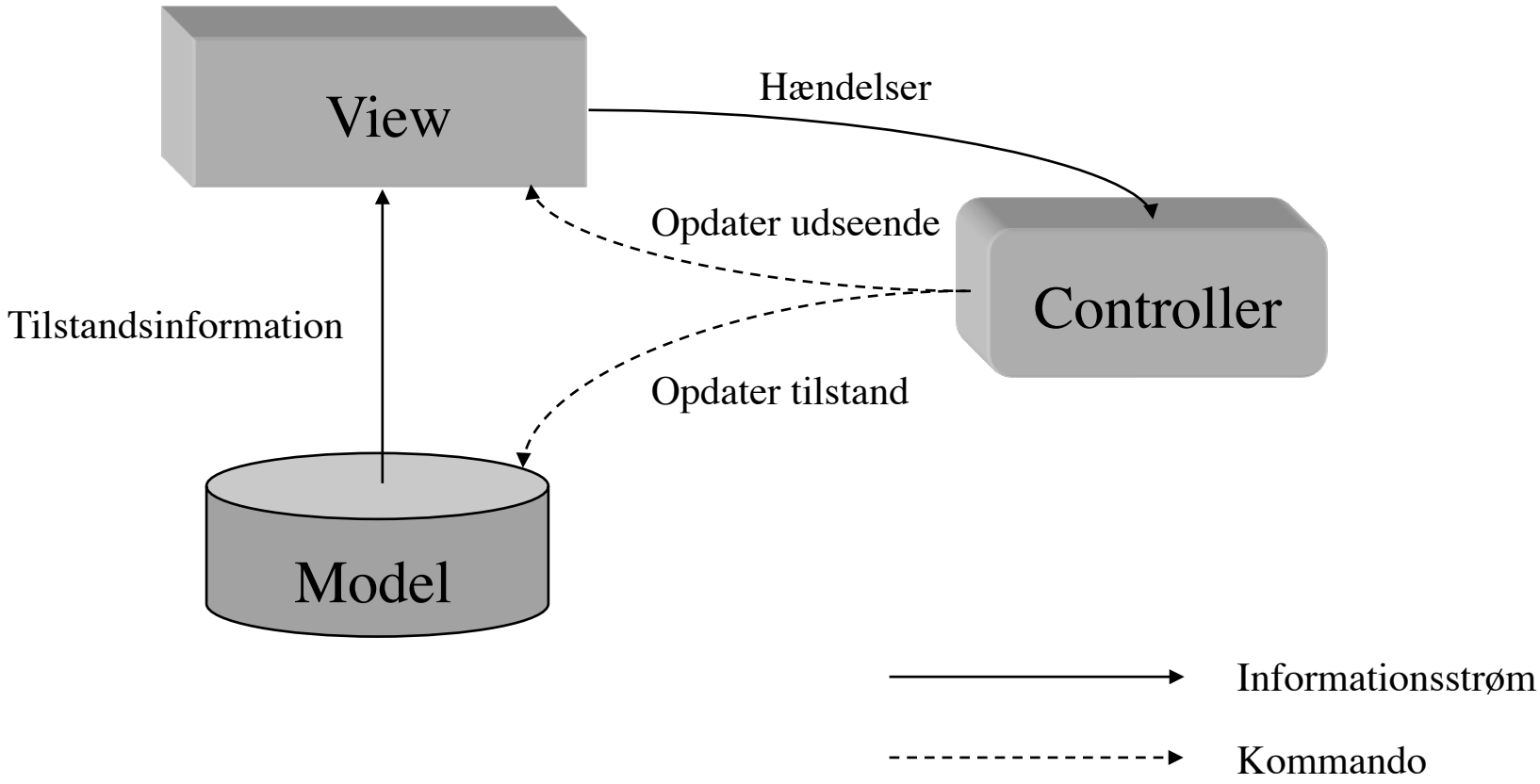
- to menneskespillere,
- to maskinspillere, eller
- en menneskespiller og en maskinspiller

Design

Programmet beskrevet i det følgende afviger fra bogens ved:

- anvendelse af Model-View-Controller-arkitekturen
- anvendelse af Swing
- en simplere implementering af den grafiske brugergrænseflade

MVC-arkitekturen



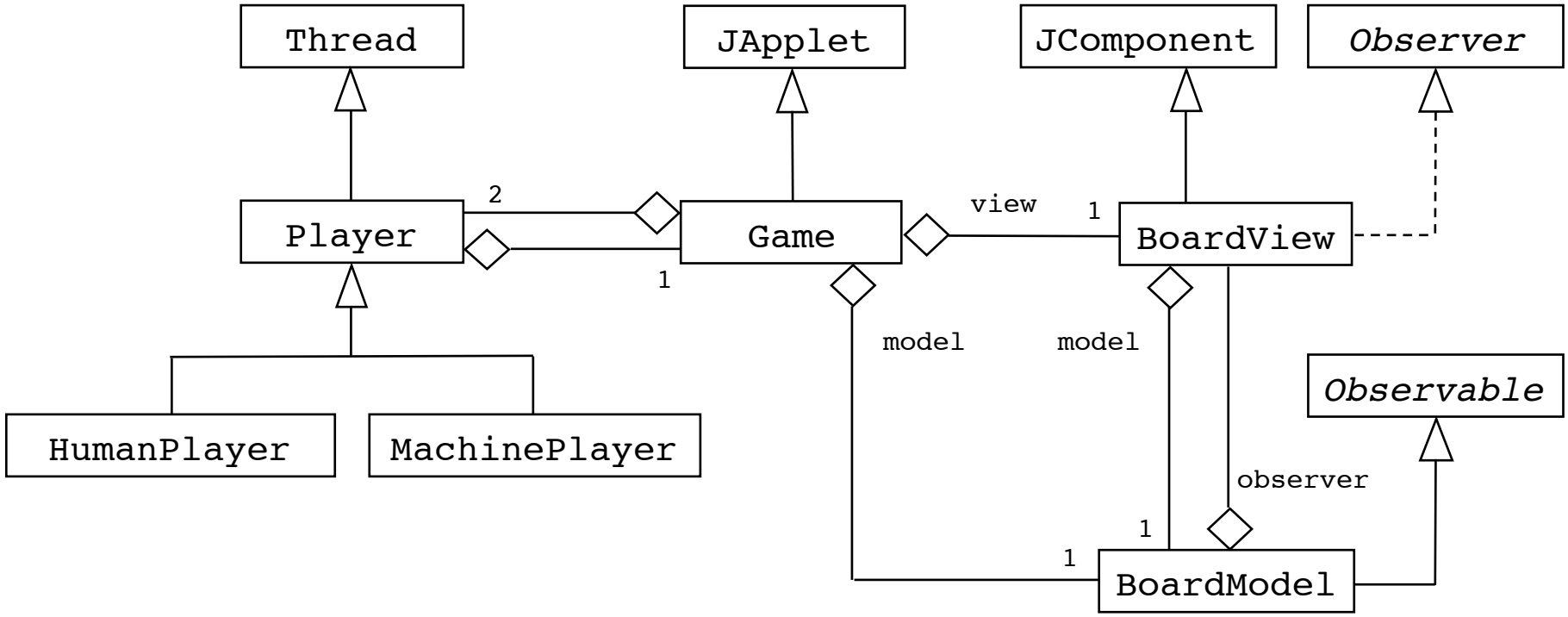
`Observer` og `Observable`

Java understøtter brugen af MVC-konceptet igennem grænsefladen `Observer` og klassen `Observable`.

Et `Observable`-objekt har en metode, `addObserver`, som benyttes til at registrere dets `Observer`-objekter.

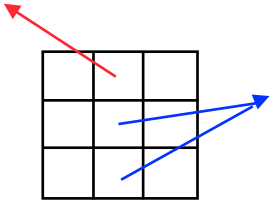
`Observer`-objekterne kan underrettes om ændringer ved kald af metoden `notifyObservers`. Disse `Observer`-objekter vil da få kaldt deres `update`-metode.

Programstruktur



Hvor er controller-delen?

Klassen `BoardModel`



```
public class BoardModel extends Observable {
    protected Player[][] board;
    protected int rows, columns;
    protected int moves, maxMoves;
    protected boolean isOver;
    protected Player winner;

    public BoardModel(int rows, int columns) {
        this.rows = rows;
        this.columns = columns;
        maxMoves = rows * columns;
        board = new Player[rows][columns];
    }

    public boolean isLegalMove(Move move) { ... }

    public boolean makeMove(Move move, Player player) { ... }

    // ... Auxiliary methods
}
```

isLegalMove

```
public class Move {
    public int row, col;
}
```

```
public boolean isLegalMove(Move move)
    return move.row >= 0 && move.row < rows &&
           move.col >= 0 && move.col < columns &&
           board[move.row][move.col] == null;
}
```

makeMove

```
public boolean makeMove(Move move, Player player) {
    if (isLegalMove(move)) {
        board[move.row][move.col] = player;
        moves++;
        checkGame(move, player);
        setChanged();
        notifyObservers(move);
        return true;
    }
    return false;
}
```

Hjælpemetoden `checkGame`

```
protected void checkGame(Move move, Player player) {
    isOver = checkRow(player, move.row) ||
             checkColumn(player, move.col) ||
             checkDiagonal1(player) ||
             checkDiagonal2(player);
    winner = isOver ? player : null;
    if (moves >= maxMoves)
        isOver = true;
}
```

checkRow og checkColumn

```
protected boolean checkRow(Player player, int row) {
    for (int i = 0; i < columns; i++)
        if (board[row][i] != player)
            return false;
    return true;
}
```

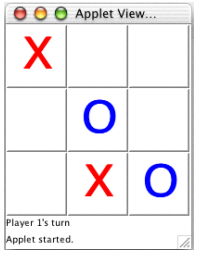
```
protected boolean checkColumn(Player player, int col) {
    for (int i = 0; i < rows; i++)
        if (board[i][col] != player)
            return false;
    return true;
}
```

checkDiagonal1 og checkDiagonal2

```
protected boolean checkDiagnal1(Player player) {
    for (int i = 0; i < rows && i < columns; i++)
        if (board[i][i] != player)
            return false;
    return true;
}
```

```
protected boolean checkDiagonal2(Player player) {
    for (int i = 0; i < rows && i < columns; i++)
        if (board[i][columns - i - 1] != player)
            return false;
    return true;
}
```

Klassen `BoardView`




```
public class BoardView extends JComponent implements Observer {
    protected BoardModel model;
    protected JButton[][] button;

    public BoardView(BoardModel model) {
        this.model = model;
        setLayout(new GridLayout(model.rows, model.columns));
        button = new JButton[model.rows][model.columns];
        for (int row = 0; row < model.rows; row++)
            for (int col = 0; col < model.columns; col++)
                add(button[row][col] = new SquareButton(row, col));
        model.addObserver(this);
    }

    public void addButtonListener(ActionListener a) { ... }

    public void update(Observable obj, Object arg) { ... }
}
```

Klassen `SquareButton`

```
public class SquareButton extends JButton {
    public SquareButton(int row, int col) {
        this.row = row;
        this.col = col;
        setBackground(Color.white);
        setFocusPainted(false);
        setBorder(BorderFactory.createRaisedBevelBorder());
        setFont(new Font("Helvetica", Font.BOLD, 48));
    }

    protected int row, col;
}
```

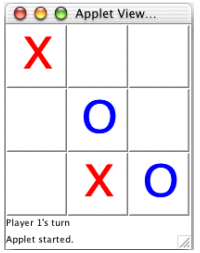
addButtonListener

```
public void addButtonListener(ActionListener a) {
    for (int row = 0; row < model.rows; row++)
        for (int col = 0; col < model.columns; col++)
            button[row][col].addActionListener(a);
}
```

update

```
public void update(Observable obj, Object arg) {
    Move move = (Move) arg;
    Player player = model.board[move.row][move.col];
    JButton b = button[move.row][move.col];
    if (player.id == 1) {
        b.setForeground(Color.red);
        b.setText("X");
    } else if (player.id == 2) {
        b.setForeground(Color.blue);
        b.setText("O");
    }
    b.repaint();
}
```

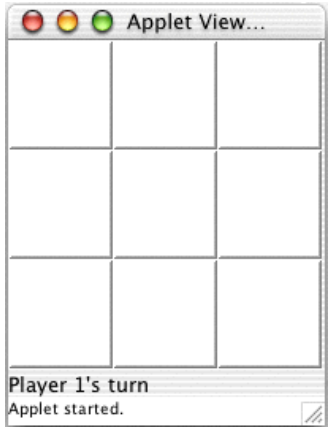
Klassen Game




```
public class Game extends JApplet {
    protected Player[] player;
    protected Player turn;
    protected BoardModel model;
    protected BoardView view;
    protected JLabel messageBar;

    public Game() { ... }

    public void init() { ... }

    public boolean makeMove(Move move) { ... }

    public Player getPlayer() { return turn; }

    public boolean isOver() { return model.isOver(); }

    public void displayMessage(String msg) {
        messageBar.setText(msg);
    }
}
```

fortsættes

Konstruktøren i `Game`




```
public Game() {
    players = new Player[2];
    model = new BoardModel(3, 3);
    view = new BoardView(model);
    messageBar = new JLabel("Game begin.");
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(view, BorderLayout.CENTER);
    getContentPane().add(messageBar, BorderLayout.SOUTH);
}
```

Eksempel på HTML-fil

```
<html>
  <head>
    <title>Tic-Tac-Toe Game</title>
  </head>
  <body>
    <h2> Human vs. Machine </h2>
    <applet archive=JavaClasses.jar
            code=Game.class
            width=200 height=225>
      <param name=type value="human-machine">
    </applet>
  </body>
</html>
```

init

```
public void init() {
    String gameType = getParameter("type");
    if ("human-human".equals(gameType)) {
        player[0] = new HumanPlayer(this, 1);
        player[1] = new HumanPlayer(this, 2);
    } else if ("machine-machine".equals(gameType)) {
        player[0] = new MachinePlayer(this, 1);
        player[1] = new MachinePlayer(this, 2);
    } else {
        player[0] = new HumanPlayer(this, 1);
        player[1] = new MachinePlayer(this, 2);
    }
    player[0].setNext(player[1]);
    player[1].setNext(player[0]);
    player[0].start();
    player[1].start();
    player[0].hasTurn();
}
```

makeMove

```
public boolean makeMove(Move move) {
    if (model.makeMove(move, turn)) {
        if (isOver()) {
            Player winner = model.winner;
            if (winner != null)
                displayMessage("Player " + winner.id + " won.");
            else
                displayMessage("It's a draw");
            for (int i = 0; i < player.length; i++)
                player[i].interrupt();
        }
        return true;
    }
    return false;
}
```

Klassen `Player`

```
abstract public class Player extends Thread {
    protected Game game;
    protected int id;
    protected Player next, turn;

    public Player(Game game, int id) {
        this.game = game;
        this.id = id;
    }

    public synchronized void setNext(Player p) { next = p; }

    abstract public Move makeMove();
    public synchronized void run() { ... }
    public synchronized void hasTurn() { ... }
}
```

fortsættes

run

```
public synchronized void run() {
    while (!game.isOver()) {
        try {
            while (turn != this)
                wait();
        } catch (InterruptedException e) {}
        if (game.isOver())
            return;
        game.displayMessage("Player " + id + "'s turn");
        while (true) {
            Move move = makeMove();
            if (game.makeMove(move))
                break;
            game.displayMessage("Illegal move!");
        }
        turn = null;
        next.hasTurn();
    }
}
```

hasTurn

```
public synchronized void hasTurn() {
   turn = game.turn = this;
   notify();
}
```

Klassen HumanPlayer

```
public class HumanPlayer extends Player implements ActionListener {
    Move move;

    public HumanPlayer(Game game, int id) {
        super(game, id);
        move = new Move();
        game.view.addButtonListener(this);
    }

     public synchronized Move makeMove() {
        try {
            wait();
        } catch (InterruptedException e) {}
        return move;
    }

    public synchronized void actionPerformed(ActionEvent event) {
        SquareButton b = (SquareButton) event.getSource();
        move.row = b.row;
        move.col = b.col
        notify();
    }
}
```

Klassen `MachinePlayer`

```
public class MachinePlayer extends Player {
    Move move;

    public MachinePlayer(Game game, int id) {
        super(game, id);
        move = new Move();
    }

    public Move makeMove() { ... }
}
```

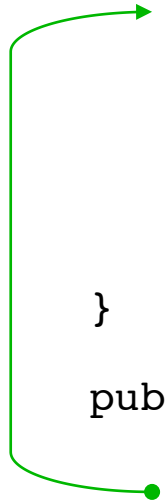
fortsættes

makeMove

```
public Move makeMove() {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {}
    int rows = game.model.rows;
    int columns = game.model.columns;
    int ncells = rows * columns;
    int i = (int) (Math.random() * ncells);
    while (true) {
        move.row = i / columns;
        move.col = i % columns;
        if (game.model.isLegalMove(move))
            break;
        i = ++i % ncells;
    }
    return move;
}
```

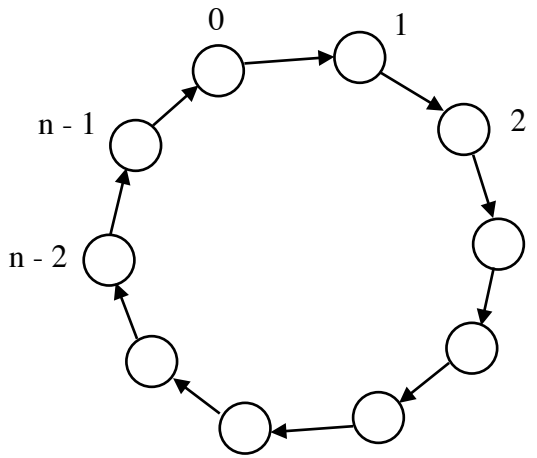
Idiom: Taking Turns

```
class Participant extends Thread {
    protected Partipant next;
    protected Partipant turn;

    public synchronized void run() {
        while (!isDone()) {
            try {
                while (turn != this)
                    wait();
            } catch (InterruptedException e) { return; }
            // perform an action or make a move
            turn = null;
            next.hasTurn();
        }
    }

    public synchronized void hasTurn() {
        turn = this;
        notify();
    }
}
```



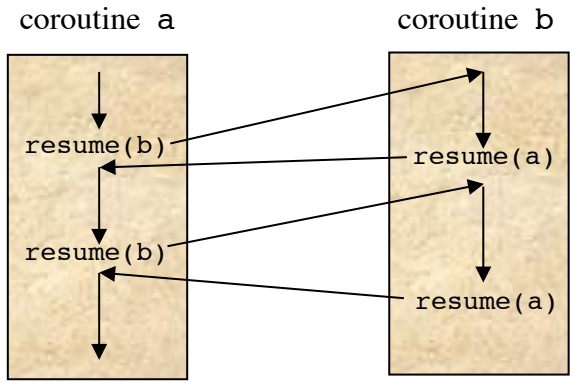
Opstart

```
Participant[] p = new Participant[n];

for (int i = 0; i < n; i++)
    p[i] = new Participant();

for (int i = 1; i < n; i++)
    p[i - 1].next = p[i];
p[n - 1].next = p[0];

for (int i = 0; i < n; i++)
    p[i].start();

p[0].hasTurn();
```



Korutiner

En **korutine** er en rutine, der midlertidigt kan standse sin udførelse.
I mellemtiden kan en anden korutine blive udført.
En standset korutine kan senere genoptage sin udførelse.



Klassen `Coroutine`

K. Helsgaun,

Discrete Event Simulation in Java

```
public class abstract Coroutine {
    protected abstract void body();

    public static void resume(Coroutine c);
    public static void call(Coroutine c);
    public static void detach();
}
```

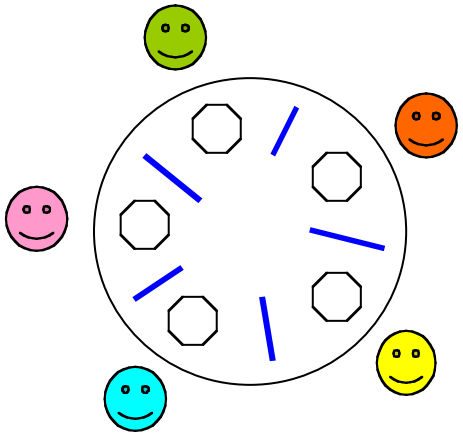
Ugeseddel 8

19. oktober - 26. oktober

- Læs kapitel 12 i lærebogen (side 587 - 643).
- Løs opgave 11.3.
- Løs opgaven på de næste sider.

Ekstraopgave 6

Fem filosoffer sidder omkring et rundt bord i dybe tanker. Udover at tænke må de en gang imellem spise. Foran hver af filosofferne er en skål med ris. Før en filosof kan spise, må han have to kinesiske spisepinde. Han tager én pind ad gangen – en fra venstre, eller en fra højre.



Filosofferne må finde en måde at deles om pindene, således at de alle får noget at spise.

På de følgende sider er vist et udkast til et program, der simulerer forløbet. Programmet stopper, når alle filosoffer har indtaget 100 måltider.

(a) Programmér klassen `Chopstick`.

I denne udgave af programmet kan der opstå baglås (deadlock). Det sker, hvis alle filosoffer tager den venstre spisepind samtidigt. Så bliver de fastlåst i deres forgæves forsøg på at få en højre pind.

Problemet kan løses på flere måder. En løsning er at lade ulige nummererede filosoffer tage den venstre spisepind først, og lade lige nummererede filosoffer tage den højre pind først.

(b) Programmér denne løsning.

En anden løsning er højst at give 4 af filosofferne adgang til bordet samtidigt.

(c) Programmér denne løsning.

En tredje løsning er kun at give en filosof lov til at tage pinde op, hvis begge pinde er fri. Begge pinde tages da op som en atomisk operation.

(d) Programmér denne løsning.

```
public class DiningPhilosophers {
    public static void main(String args[]) {
        ChopStick[] chopStick = new ChopStick[N];
        for (int i = 0; i < N; i++)
            chopStick[i] = new ChopStick();
        for (int i = 0; i < N; i++)
            new Philosopher(i, chopStick[(i - 1 + N) % N],
                            chopStick[i]).start();
    }

    static final int N = 5;
}
```

```
class Philosopher extends Thread {
    public Philosopher(int id, ChopStick left, ChopStick right) {
       this.id = id;
       leftChopStick = left;
       rightChopStick = right;
    }

    public void run() {
        for (int meals = 0; meals < 100; meals++) {
            think();
            leftChopStick.grab();
            rightChopStick.grab();
            eat();
            leftChopStick.release();
            rightChopStick.release();
        }
        System.out.println("Philosopher #" + id + " leaves the room");
    }

    ...
```

fortsættes

```
    void think() {
        System.out.println("Philosopher #" + id + " is thinking");
        try {
            sleep((long) (Math.random() * 10));
        } catch (InterruptedException e) {}
        System.out.println("Philosopher #" + id + " is hungry");
    }

    void eat() {
        System.out.println("Philosopher #" + id + " starts eating");
        try {
            sleep((long) (Math.random() * 20));
        } catch (InterruptedException e) {}
        System.out.println("Philosopher #" + id + " is stuffed");
    }

    int id;
    ChopStick leftChopStick, rightChopStick;
}

class ChopStick { /* spørgsmål (a) */ }
```