TDA 545: Objektorienterad programmering

Föreläsning 15:
Exceptions & lite swing, gränssnitt

Magnus Myréen

Chalmers, läsperiod 1, 2015-2016

Idag

Idag: exceptions, kap 15 fram t.o.m. sida 655

Extra:

- lite problemlösning med swing
- ett stiligt sätt att definiera gränssnitt

Exceptions:

- skillnaden mellan checked / unchecked exceptions
- att kasta (throw) exceptions
- att fånge (catch) exceptions
- när och hur man bör använda exceptions

Efter denna föreläsning… (nästa sida)

Efter denna föreläsning:

Inget nytt material.

Nästa gång (dvs *imorgon*): talar om tentan, förberedelse, exempel, mm.

Nästa vecka: Jag är här i HA4/HC4 de vanliga föreläsningstiderna.
Man kan komma och fråga mig frågor.
Inget nytt material.
Vad vill ni att jag gör på dessa '*föreläsningar*'?

Lite problemlösning med swing

Hur skriver man kod som får ett fönster att se så här ut?

Se uppgift 3 i http://www.cse.chalmers.se/edu/course/TDA545/2014-om-tenta-1.pdf

Lite problemlösning med swing

Kod som skrevs på föreläsningen:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Demo1 extends JFrame {

    public Demo1() {

        JPanel main = new JPanel();
        main.setBackground(Color.red);
        main.setLayout(new BorderLayout());

        JPanel pN = new JPanel();
        pN.setBackground(Color.blue);
        main.add(pN,BorderLayout.NORTH);

        JLabel l = new JLabel("56");
        pN.add(l);

        JPanel pC = new JPanel();
        pC.setBackground(Color.yellow);
        pC.setLayout(new GridLayout(4,4));
        main.add(pC,BorderLayout.CENTER);

        JButton b;
        b = new JButton("1");
        pC.add(b);
        b = new JButton("2");
        pC.add(b);
        b = new JButton("3");
        pC.add(b);
        b = new JButton("+");
        pC.add(b);
        b = new JButton("4");
        pC.add(b);
        b = new JButton("5");
        pC.add(b);
        b = new JButton("6");
        pC.add(b);
        b = new JButton("-");
        pC.add(b);
        b = new JButton("7");
        pC.add(b);
        b = new JButton("8");
        pC.add(b);
        b = new JButton("9");
        pC.add(b);
        b = new JButton("*");
        pC.add(b);
        b = new JButton("0");
        pC.add(b);
        b = new JButton("C");
        pC.add(b);
        b = new JButton("=");
        pC.add(b);
        b = new JButton("/");
        pC.add(b);

        add(main);
        pack();
        setVisible(true);

        this.addComponentListener(new ComponentListener() {
                public void componentHidden(ComponentEvent e) {}
                public void componentMoved(ComponentEvent e) {}
                public void componentResized(ComponentEvent e) {
                    pN.setPreferredSize(
                       new Dimension(main.getWidth(),main.getHeight()/3));
                }
                public void componentShown(ComponentEvent e) {}
            });

    }

    public static void main(String[] args) {
        Demo1 f = new Demo1();
    }

}
```

Gränssnitt

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class L implements ActionListener {

    private int i = 0;

    public void actionPerformed(ActionEvent e) {
        System.out.println(i);
        i = i+1;
    }

}
```

En ActionListener klass som inte gör mycket.

```
public class Test1 {

    public static void main(String[] args) {
        JFrame f = new JFrame();
        JButton b = new JButton("Knapp");
        ActionListener l = new L();
        b.addActionListener(l);
        JPanel panel = new JPanel();
        panel.add(b);
        f.add(panel); f.pack();
        f.setVisible(true);
    }

}
```

… den enda instansen skapas och används här.

Gränssnitt

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

Man kan skriva en kort klassdefinition (för gränssnitt) inne i vanlig kod.

```
public class Test2  {

    public static void main(String[] args) {
        JFrame f = new JFrame();
        JButton b = new JButton("Knapp");
        ActionListener l = new ActionListener() {
                private int i = 0;
                public void actionPerformed(ActionEvent e) {
                    System.out.println(i);
                    i = i+1;
                }
            };
        b.addActionListener(l);
        JPanel panel = new JPanel();
        panel.add(b);
        f.add(panel); f.pack();
        f.setVisible(true);
    }

}
```

Här definierar vi en klass som implementerar ActionListener och genast skapar en instans.

Gränssnitt

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

Man kan skriva en kort klassdefinition (för gränssnitt) inne i vanlig kod.

```
public class Test2  {

    public static void main(String[] args) {
        JFrame f = new JFrame();
        JButton b = new JButton("Knapp");
        ActionListener l = new ActionListener() {
            private int i = 0;
            public void actionPerformed(ActionEvent e) {
                System.out.println(i);
                i = i+1;
                b.setText("i = " + String.valueOf(i));
            }
        };
        b.addActionListener(l);
        JPanel panel = new JPanel();
        panel.add(b);
        f.add(panel); f.pack();
        f.setVisible(true);
    }

}
```

Här definierar vi en klass som implementerar ActionListener och genast skapar en instans.

Man kan använda variabler som är deklarerade utan för klassen *enkelt och snyggt*!

Exceptions

≈ exceptional event

Program skall aldrig behöva "krascha".

Javas exceptions används för att ta hand om sådana situationer som normalt inte skall inträffa ***och*** som är krävande att testa på.

Exempel på fel som kan vara lämpliga att hantera med exceptions:

- öppnande av fil som inte existerar
- indexering utanför gränserna i ett fält
- division med noll
- försök att läsa/skriva mot en enhet som inte fungerar
- vissa inmatningsfel, tex en bokstav i stället för en siffra
- minnet tar slut

Obs. Exceptions är "långsamma" jämfört med vanlig kod.

Syntax

Kastar en exception:

```
throw exception
```

Man fångar exceptions med:

```
try {
    kod som kanske kastar exception

} catch (MyExceptionClass1 name) {

    kod som reagerar till exception
    som kom i namn variabeln.

} catch (MyExceptionClass2 name) {

    ...

} finally {

    kod som alltid körs

}
```

Semantik

När ett fel inträffar skapas ett objekt som innehåller

- felets typ,
- ett felmeddelande
- och information om systemets status när felet uppstod. (en "trace")

Detta objekt lämnas till runtime-systemet som försöker hitta en felhanterare (dvs ett catch-block) genom att

- först söka i metoden där felet uppstod,
- sedan i metoden som kallade på metoden där felet uppstod, sedan osv...
- tills man når huvudprogrammet main.

Om ingen felhanterare för felet hittas avslutas programmet och felet och en "trace" skrivs ut på `standard.err` (vanligen skärmen).

Kontrollen kan inte återgå till det ställe där felet uppstod utan ett fel innebär alltid att blocket etc. avslutas. Hanterare i tex funktioner måste därför ha en return sats.

Syntax

Kastar en exception:

```
throw exception
```

Man fångar exceptions med:

```
try {
    kod som kanske kastar exception
} catch (MyExceptionClass1 name) {
    kod som reagerar till exception
    som kom i namn variabeln.
} catch (MyExceptionClass2 name) {
    ...
} finally {
    kod som alltid körs
}
```

Fångar exceptions av typen/klassen **MyExceptionClass1**

Om exception inte fångades så kollar vi om det är en **MyExceptionClass2**

Behöver inte finnas.

Kod som alltid utförs oavsett om ett fel inträffade eller ej. Vanligen för att “städa upp” tex stänga filer.

Varför exceptions?

Separerar felhantering från “vanlig” kod.

Man programmerar först som om felen inte inträffade, sen tar man hand om felen.

Enkel felpropagering

… tackvare propagering till närmsta felhanterare. Ibland långsamt.

Ger gruppering av feltyper

Fel är ju objekt som tillhör klasser, arv!

Terminologi för exceptions

Exceptions kan vara:

fördefinierade (finns i Javas API), eller

egendefinierade (du får skriva själv)

Exceptions kan vara:

checked (måste hanteras eller nämnas i signaturen), eller

unchecked (kan ignoreras)

Exempel strax….

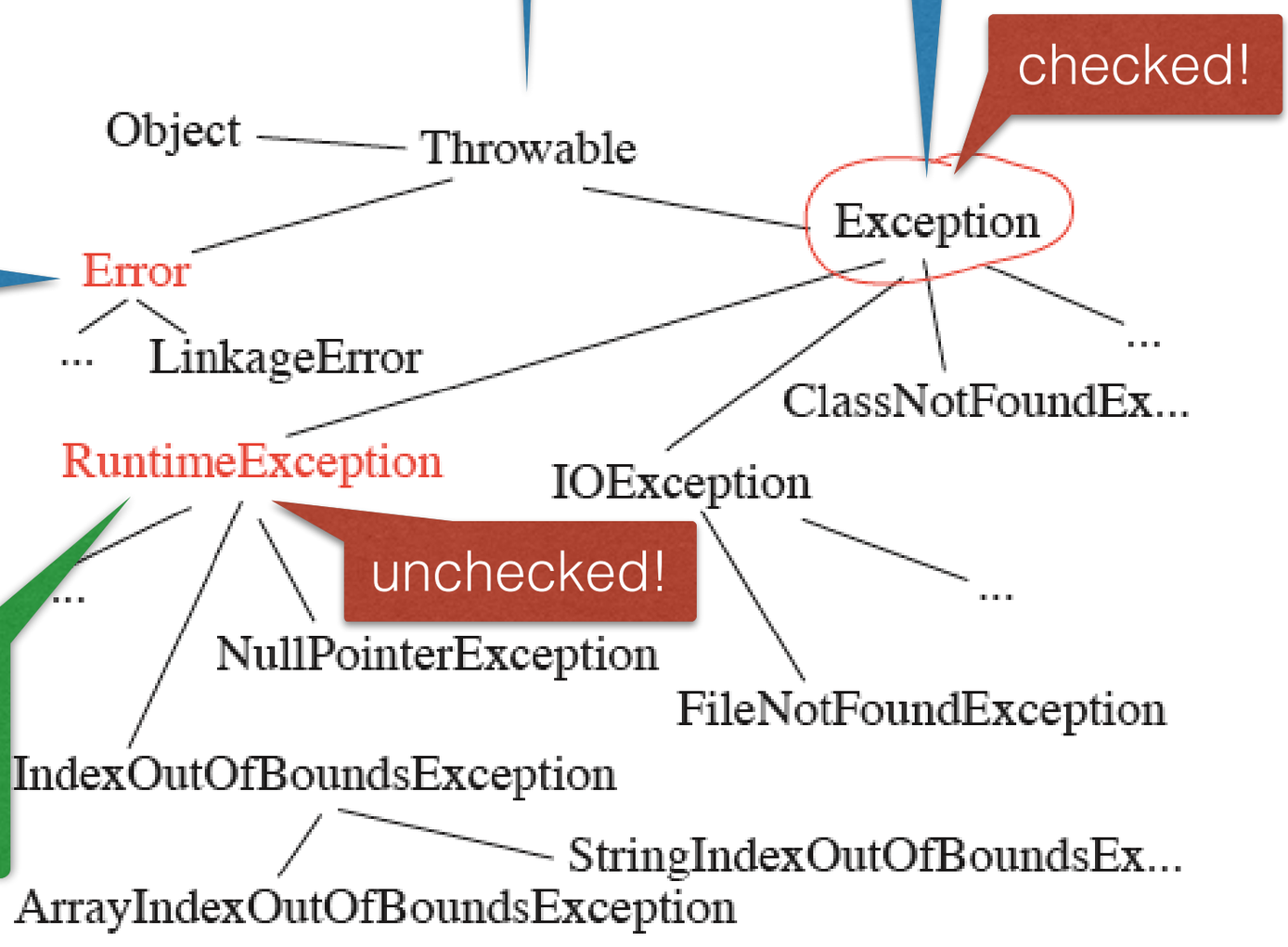
Gruppering av exceptions

En exceptionell händelse beskrivs med hjälp av ett objekt som tillhör någon subklass till standardklassen **Throwable**.

Subklasser till **Exception** beskriver vanliga fel. Denna typ av fel orsakas av själva programmet och vid interaktionen med programmet. Sådana fel kan fångas och hanteras i programmet med **try…catch**.

Klassen **Error** används av Java interpretatorn för att signalera allvarliga interna systemfel t.ex. länkningsfel och att minnet tagit slut. Vanligtvis finns inte mycket att göra åt denna typ av fel. *Vi kommer inte att behandla denna typ av fel.*

Exceptions under **RuntimeExceptions** behöver inte fångas och inte hanteras om man inte vill. (unchecked)



Från Javas API

java.lang

Class NumberFormatException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IllegalArgumentException
          java.lang.NumberFormatException

Ett **catch**-block fångar alla exceptions som är av specificerad klass och subklasser till denna klass.

Ordningen av **catch**-blocken är således av betydelse.

Fångar man en **Throwable** eller en **Exception** så fångar man allt och det är vanligtvis inte OK,
man skall vara mer specifik.

Mera från Javas API

parseInt

```
public static int parseInt(String s)
            throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s` - a `String` containing the `int` representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` - if the string does not contain a parsable integer.

Javas API nämner exceptions.

Att kasta exceptions

```
public class Circle {

    private int radius = 0;

    public void setRadius(int r) {
        if (0 <= r) {
            radius = r;
        } else {
            throw new IllegalArgumentException("Cannot have negative radius");
        }
    }
}
```

Skapar ett nytt exception objekt av typen **IllegalArgumentEx**.

Kastar exception objektet.

Att kasta exceptions

Man kan skriva i signaturen att en exception kanske kastas.

Obs. måste nämnas i signaturen ifall det är en ***checked exception***.

```
public class Circle {

    private int radius = 0;

    public void setRadius(int r) throws IllegalArgumentException {
        if (0 <= r) {
            radius = r;
        } else {
            throw new IllegalArgumentException("Cannot have negative radius.”);
        }
    }
}
```

Skapar ett nytt exception objekt av typen **IllegalArgumentEx**.

Kastar exception objektet.

Att kasta egna exceptions

```
class CircleEx extends Exception {
    public CircleEx(String str) { super(str); }
}

class CircleRunEx extends RuntimeException {
    public CircleRunEx(String str) { super(str); }
}
```

Ärver *inte* **RuntimeException**, är alltså en *checked* exception.

Ärver **RuntimeException**, är alltså en ***un****checked* exception.

```
public class Circle2 {

    private int radius;

    public void setRadius(int r) {
        if (0 <= r) {
            radius = r;
        } else {
            throw new CircleRunEx("Cannot have negative radius.");
        }
    }

}
```

Här kastar vi vår egen exception.

Att kasta egna exceptions

```
class CircleEx extends Exception {
    public CircleEx(String str) { super(str); }
}

class CircleRunEx extends RuntimeException {
    public CircleRunEx(String str) { super(str); }
}
```

Ärver *inte* **RuntimeException**, är alltså en *checked* exception.

Ärver **RuntimeException**, är alltså en ***un****checked* exception.

```
public class Circle3 {

    private int radius;

    public void setRadius(int r) throws CircleEx {
        if (0 <= r) {
            radius = r;
        } else {
            throw new CircleEx("Cannot have negative radius.");
        }
    }

}
```

Obs. Man måste skriva **throws** här för *checked* exc.

Här kastar vi vår egen *checked* exception.

Hur skall felet (exception) hanteras?

Hur ska exceptions hanteras?

Exempel:

- Ta inte hand om felet alls här (vanligt!)
- Kasta vidare med bättre felutskrift
- Kasta vidare som annan exception
- Felutskrift+avsluta
- Ta hand om felet och vidtag åtgärd (“error recovery”) så programmet kan fortsätta som om inget hänt (måste göras på rätt ställe och rätt sätt…)
- Städa, tex stänga filer, gör felutskrift och avsluta programmet.

Exempel: negligera

Om vi inte vill/kan ta hand om felet så kan vi negligera det.

```
public static int getInt() throws NumberFormatException {
  ... // ev annan kod
  int i = 0;
  String str = myIn.nextLine();
  i = Integer.parseInt(str);
  return i;
}
```

Bra i så fall om vi nämner att vi eventuellt kastar det.

Exempel: kasta vidare (1)

Vi kan ändra på meddelandet och kast vidare.

```
public static int getInt() {
  ...// ev annan kod
  int i = 0;
  try {
    String str = myIn.nextLine();
    i = Integer.parseInt(str);
  } catch (NumberFormatException e) {
    throw new NumberFormatException("getInt: unable to parse int.”);
  }
  return i;
}
```

Ofta skriver man metodens namn in i meddelandet.

Exempel: kasta vidare (2)

… eller så kan man ändra på exception och kast vidare.

```
public static int getInt() {
  ...// ev annan kod
  int i = 0;
  try {
    String str = myIn.nextLine();
    i = Integer.parseInt(str);
  } catch (NumberFormatException e) {
    String str = "getInt: expected an int, parsing failed.";
    throw new IllegalArgumentException(str);
  }
  return i;
}
```

Vi byter exception typ.

Exempel: felutskrifter och avsluta

I vissa fall är det bäst att avsluta programmet.

```
public static int getInt() {
  ...// ev annan kod
  int i = 0;
  try {
    String str = myIn.nextLine();
    i = Integer.parseInt(str);
  } catch (NumberFormatException e) {
    System.out.println("Read failed!");
    e.printStackTrace();
    System.exit(1);
  }
  return i;
}
```

Skriver ut alla metodanrop som gjorts för att komma hit.

Stänger programmet.

Exempel: ta hand om felet, vidtag åtgärd

Vi kan ta hand om felet och fortsätta eller försöka igen.

```
public static int getInt() {
  int i=0;
  // ev. annan kod här
  boolean notReady = true;
  while (notReady) {
    try {
      String str = myIn.nextLine();
      i = Integer.parseInt(str);
      notReady = false;
    } catch (NumberFormatException e) {
      System.out.println("Read failed!");
      System.out.print(" Ge talet på nytt: ");
    }
  }
  return i;
}
```

try-catch inuti en **while** loop.

Användbara metoder

Ur objektet som skapas (e ovan) kan man få en del information:

`String toString()`
Returns a short description of this throwable.

`void printStackTrace()`
Prints this throwable and its backtrace to the standard error stream.

`String getMessage()`
Returns the detail message string of this throwable.

http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

Några alternativ för catch blocket

```
System.out.println("<felmeddelande>");

e.printStackTrace();

System.out.println(e.getMessage());
```

```
throw e;

throw new <Exnamn>("<felmeddelande>");

System.exit(1);

... kod som gör en åtgärd.
```

http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

Var skall felet (exception) hanteras?

Var skall exception hanteras?

Mycket dåligt:

```
public Circle (int newRadius) {
  try {
    setRadius(newRadius);
    numberOfObjects++;
  } catch (IllegalArgumentException e) {
    e.printStackTrace();
  }
}
```

kan kasta **IllegalArgumentException**

skriver ut en anrops lista… men inget mera

Är detta bra?

1) här kan vi ju inte göra något åt händelsen annat än att skriva ut en felutskrift.

2) Dessutom fortsätter sedan programmet som om allt gick bra. Gjorde det det?

Man skall ta hand om felhändelserna där man kan göra vettiga åtgärder. Vanligtvis är det INTE där felet uppstår utan hos någon som anropade metoden där felet uppstod eller längre upp i anropskedjan.

När skall felet (exception) hanteras?

När skall man använda exceptions?

Exempel — ArrayIndexOutOfBounds

Exceptions lämpar sig väl för händelser

- som normalt inte skall inträffa och
- som är omständliga att testa på.

```
a[x][y] = a[k][l] + 3;
```

Här blir det väldigt omständligt att testa alla indexgränser varje gång man skall använda fältet så här funkar exceptions bättre än egen testning, speciell som det sköts av runtime systemet.

```
if ( x >= 0 && x < a.length &&
     y >= 0 && y < a[x].length &&
     k >= 0 && k < a.length &&
     l >= 0 && l < a[k].length ) {
  a[x][y] = a[k][l] + 3;
}
```

Skall man använda exceptions här? (1)

Hur kan detta program crasha?

```
import java.util.Scanner;

public class Demo5 {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String str = in.nextLine();
        if (str.equals("yes")) {
            System.out.println("Hello there!");
        } else {
            System.out.println("See you later!");
        }
    }

}
```

kan fastna här

Utskrift:

Här tryckte jag på Ctrl-D

```
$ java Demo5
Exception in thread "main" java.util.NoSuchElementException: No line found
    at java.util.Scanner.nextLine(Scanner.java:1540)
    at Demo5.main(Demo5.java:7)
```

Skall man använda exceptions här? (1)

Man kan lätt undvika exceptions:

```
import java.util.Scanner;

public class Demo6 {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String str = "";
        if (in.hasNextLine()) {
            str = in.nextLine();
        }
        if (str.equals("yes")) {
            System.out.println("Hello there!");
        } else {
            System.out.println("See you later!");
        }
    }

}
```

kollar först, kör sen

Skall man använda exceptions här? (2)

Hur kan detta program crasha?

```
import java.util.Scanner;

public class Demo7 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n1 = 0;
        int n2 = 0;
        if (in.hasNextInt()) { n1 = in.nextInt(); }
        if (in.hasNextInt()) { n2 = in.nextInt(); }
        int n3 = n1 / n2;
        System.out.println("res = " + n3);
    }
}
```

division med 0

Utskrift:

```
$ java Demo7
3
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Demo7.main(Demo7.java:11)
```

Skall man använda exceptions här? (2)

Man kan lätt undvika exceptions:

```
import java.util.Scanner;

public class Demo7 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n1 = 0;
        int n2 = 0;
        if (in.hasNextInt()) { n1 = in.nextInt(); }
        if (in.hasNextInt()) { n2 = in.nextInt(); }
        if (n2 != 0) {
            int n3 = n1 / n2;
            System.out.println("res = " + n3);
        }
    }
}
```

Övning

Uppgift: definiera en egen exception DivByZero och kasta den.

checked / unchecked ?

Övning

I/O har flera exceptions…

Uppgift: skriv ett program som skriver ut texten från en fil.

FileReader, BufferedReader, IOException, …

… and finally: `finally`

Uppgift: skriv ett program som testar semantiken av `finally`

Påminnelse:

Man fångar exceptions med:

```
try {
    kod som kanske kastar exception
} catch (MyExceptionClass1 name) {
    ...
} finally {
    kod som alltid körs
}
```

körs denna kod fast vi kastar exception i **catch**?

Sammanfattning av föreläsningen

Extra:

- lite problemlösning med swing
- ett stiligt sätt att definiera gränssnitt

Viktigt i denna föreläsning:

- skillnaden mellan checked / unchecked exceptions
- att kasta (throw) exceptions
- att fånge (catch) exceptions
- när och hur man bör använda exceptions

---

*Det var **allt material** för denna kurs!*

Nästa gång (dvs imorgon):
talar om tentan, förberedelse, exempel, mm.