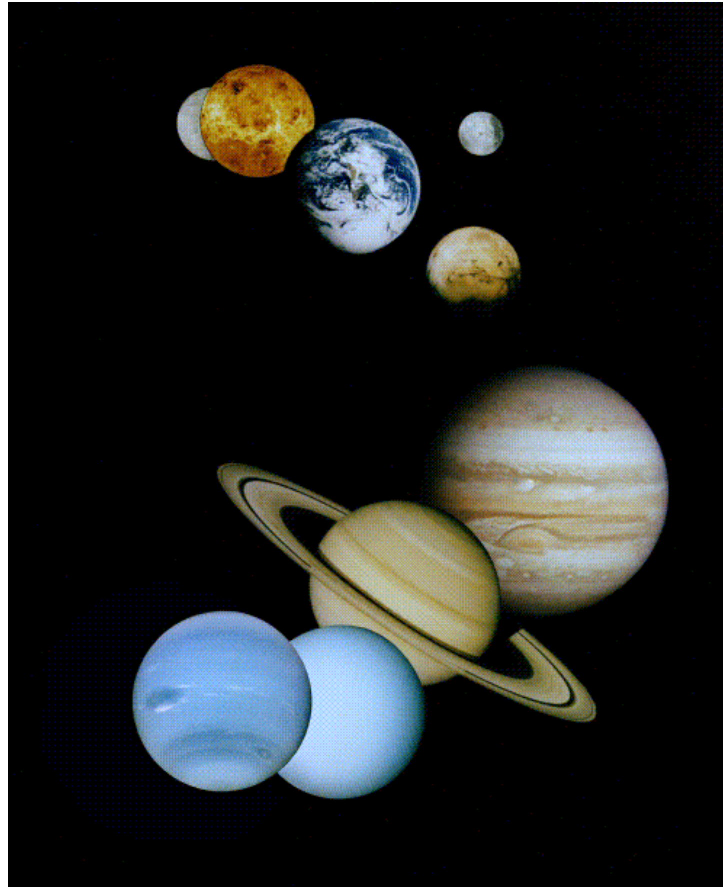
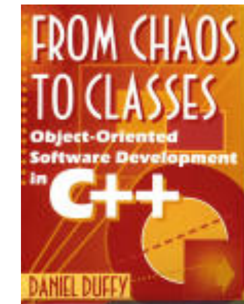
Klasser



Plan



- Typer
- Sætninger
- **Klasser**
- Strenge
- **Pakker**
- **Undtagelser**

Tre vigtige begreber i tilknytning til sprog

- **Syntaks** (ordføjningslære, grammatik)
- **Semantik** (betydningslære)
- **Pragmatik** (anvendelseslære)

Om behovet for grammatik

Når vi lærer vort første naturlige sprog, bruger vi lang tid på at høre sproget (eller støj) og ser det i forbindelse med andre menneskers handlinger. Herudfra lærer vi sprogets struktur, og hvordan det knytter sig til det, der sker i verden. Vi kender måske ikke reglerne, men vi lærer, hvordan sproget virker.

Ved programmering har vi ingen erfaring med den verden, vi er i, og vi har ingen erfaring med andre, som opererer i denne verden. Vi er derfor afhængige af grammatik.

Alan Creak (min oversættelse)

Typper

Primitive typer

	størrelse	*værdiområde*	
`boolean`		`{false, true}`	
`char`	16 bit	`\u0000` til `\uFFFF`	(Unicode)
`byte`	8 bit	`-128` til `127`	
`short`	16 bit	`-32768` til `32767`	
`int`	32 bit	`-2147483648` til `2147483647`	
`long`	64 bit	`-9223372036854775808` til `9223372036854775807`	
`float`	32 bit	6 betydende cifre ($\pm 10^{36}$, $\pm 10^{-34}$)	
`double`	64 bit	15 betydende cifre ($\pm 10^{308}$, $\pm 10^{-324}$)	

`char`, `byte`, `short`, `int` og `long` kaldes **heltalstyper**
`float` og `double` kaldes **flydendetalstyper**

Konstanter (literaler)

Heltalskonstanter kan angives med

decimal notation:	28	28L
oktal notation:	034	
heksadecimal notation:	0x1C	0X1c

Flydendetalskonstanter angives som decimaltal med en valgfri eksponentdel:

3.24	3.24e5	3.24e-5	3.24E-5
3.24f	3.24d	3.24F	

Tegnkonstanter omsluttes af to apostroffer:

'z'	'\u007A'	'\172'	(3 repræsentationer for tegnet 'z')
' \n '	' \t '		
' \' '	' \" '	'\\'	

Erklæring og initialisering

Enhver variabel erklæres ved at angive dens type, dens navn og eventuelt dens startværdi

Eksempler:

```
int num1;
double minimum = 4.50;
int x = 0, num2 = 2;
int num3 = 2 * num2;
```

Operatorers præcedens og associativitet

Kategori	Operatorer	Associativitet
Referenceoperatorer	`. []`	Venstre-mod-højre
Monadisk	`++ -- + - ~ !`	*Højre-mod-venstre*
Typificering	`new (`*type*`)`	*Højre-mod-venstre*
Multiplikativ	`* / %`	Venstre-mod-højre
Additiv	`+ -`	Venstre-mod-højre
Bitvis skift	`<< >> >>>`	Venstre-mod-højre
Sammenligning	`< <= > >= instanceof`	Venstre-mod-højre
Lighed	`== !=`	Venstre-mod-højre
Bitvis AND	`&`	Venstre-mod-højre
Bitvis XOR	`^`	Venstre-mod-højre
Bitvis OR	`|`	Venstre-mod-højre
Logisk AND	`&&`	Venstre-mod-højre
Logisk OR	`||`	Venstre-mod-højre
Betinget udtryk	`?:`	*Højre-mod-venstre*
Tildeling	`= += -= *= /= %=` `<<= >>= >>>= &= ^= |=`	*Højre-mod-venstre*

Betinget udtryk

```
max_ab = a > b ? a : b;
```

er ækvivalent med

```
if (a > b)
    max_ab = a;
else
    max_ab = b;
```

Typekonvertering

Typekonvertering er konvertering af en værdi af en type til en værdi af en anden type

Eksplicit typekonvertering (*casting*) bruges for at skabe en temporær værdi af en ønsket type

Eksempel:

```
int x = 6, y = 10;
double quotient = x / y;   // sandsynligvis forkert
```

Erstat med:

```
double quotient = (double) x / y;
```

Hvad sker der, hvis `y` er 0?

se side 83 i lærebogen

Udvidelse og indsnævring og af typer

Konvertering af en type fra et mindre værdiområde til en type af et større værdiområde kaldes for type**udvidelse** (widening)

Konvertering af en type fra et større værdiområde til en type af et mindre værdiområde kaldes for type**indsnævring** (narrowing)

Eksempel:

```
int i = 10;
long m = 10000;
double d = Math.PI;
i = (int) m;   // indsnævring (cast nødvendig)
m = i;         // udvidelse
m = (long) d;  // indsnævring (cast nødvendig)
d = m;         // udvidelse
```

Sætninger

Simple sætninger:

Udtrykssætninger (tildeling, metodekald, inkrementering, dekrementering)
Erklæringssætninger
`break`-sætninger
`continue`-sætninger
`return`-sætninger
`throw`-sætninger

Sammensatte sætninger:

sætningsblokke	(sætninger omkranset af `{}`)
valgsætninger	(`if`, `switch`)
løkkesætninger	(`while`, `do`, `for`)
`try-catch`-sætninger	

Udtrykssætninger

En udtrykssætning udgøres af et udtryk efterfulgt af et semikolon.

Eksempler:

```
x = y;                    (tildeling)
x = y = 3;                (multipel tildeling)
y += x;                   (tildeling)
x++;                      (inkrementering)
p.move(x, y);             (metodekald)
```

Erklæringssætninger

Placeringen af en variabels erklæring er bestemmende for variablens **virkefelt** (scope).

Eksempel:

```
{
   ...
   int i;       // Her begynder i's virkefelt
   i = 10;
   int j = 20; // Her begynder j's virkefelt
   i += j;
   ...
} // Her ender både i og j's virkefelt
```

`if`-sætninger

```
if (Condition)
    Statement
```

eller

```
if (Condition)
    Statement1
else
    Statement2
```

switch-sætninger

```
switch (IntegerExpression) {
case IntegerConstant_1:
     Statement_1
   .
   .
   .
case IntegerConstant_n:
     Statement_n
default:
     Statement_n+1
}
```

Eksempel på `switch`-sætning

```
switch (someCharacter) {
case '(':
case '[':
case '{':
    // Kode til at behandle startparenteser
    break;
case ')':
case ']':
case '}':
    // Kode til at behandle slutparenteser
    break;
default:
    // Kode til at behandle alle andre tegn
}
```

Løkkesætninger

```
while (Condition)
    Statement
```

```
for (InitExpr; Condition; UpdateExpr)
    Statement
```

ækvivalent med

```
do
    Statement
while (Condition);
```

```
InitExpr;
while (Condition) {
    Statement
    UpdateExpr;
}
```

```
for (Declaration : Expression)
    Statement
```

Ny i Java 5.0: for-each-løkke

Brug af **break** i løkker

```
while (...) {
    ...
    if (something)
        break;
    ...
}
```

```
outerLoop:
while (...) {
    while (...) {
        if (disaster)
            break outerLoop;
    }
}
```

Brug af `continue` i løkker

```
for (int i = 1; i <= 100; i++) {
    if (i % 10 == 0)
        continue;
    System.out.println(i);
}
```

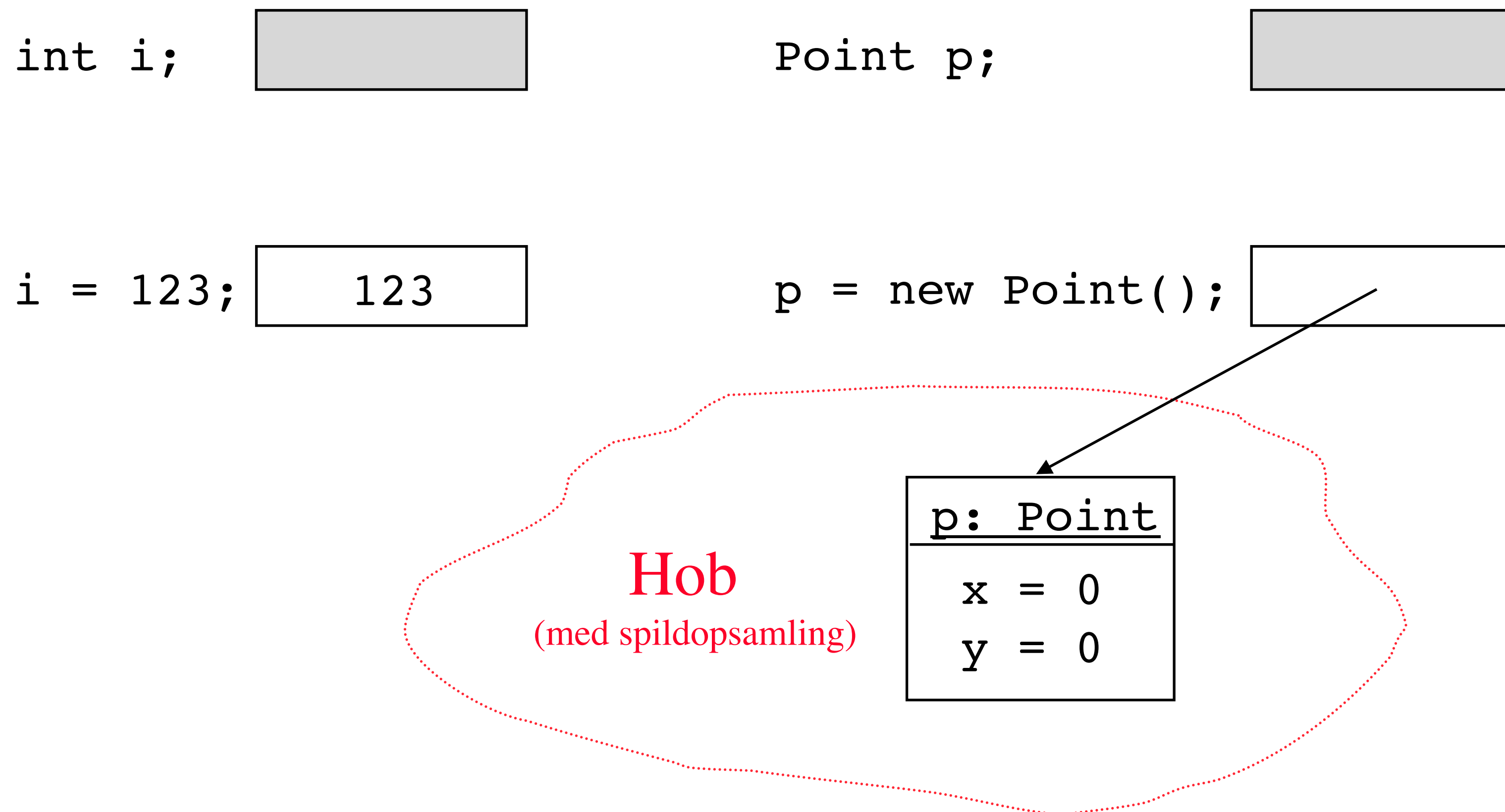
Referencetyper

En *referencetype* er en *klasse*-type, en *grænseflade*-type eller en *array*-type.

Referencevariable kan indeholde henvisninger til objekter eller arrays. Faktisk er arrays også objekter.
Internt repræsenteres henvisninger ved lageradresser (32/64 bit).

En referencevariabel kan indeholde den specielle referenceværdi `null`, hvilket angiver, at variablen ikke henviser til noget objekt.

Simple typer contra referencetyper



Endimensionale arrays

Erklæring

Type[] *Identifier*

eller

Type Identifier[]

Skabelse med new:

new *Type*[*n*] Eks.

```
int[] a = new int[10];
Point[] p = new Point[20];
```

eller ved initialisering (*initializer*):

{ $v_0, v_1, ..., v_{n-1}$ } Eks.

```
int[] a = { 7, 9, 1, 3 };
```

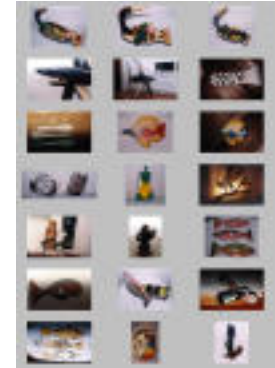
Gennemløb af endimensionale arrays

```
for (int i  = 0; i < a.length; i++)
    use a[i];
```

eller med for-each-løkke

```
for (int elem : a)
    use elem;
```

Flerdimensionale arrays
(array af arrays)



Erklæring

Type[]...[] *Identifier*

eller

Type Identifier[]...[]

Skabelse med new:

new*Type*[n_1][n_2]...[n_k]   Eks.
```
int[][] a = new int[2][3];
Point[][] p = new Point[4][5];
```

eller ved intialisering:

{ $I_0, I_1, ..., I_{k-1}$ }   Eks.
```
int[][] a = { {7, 9, 1}, {3, 2, 6} };
```

Gennemløb af todimensionale arrays

```
for (int i  = 0; i < a.length; i++)
    for (int j = 0; j < a[i].length; j++)
        use a[i][j];
```

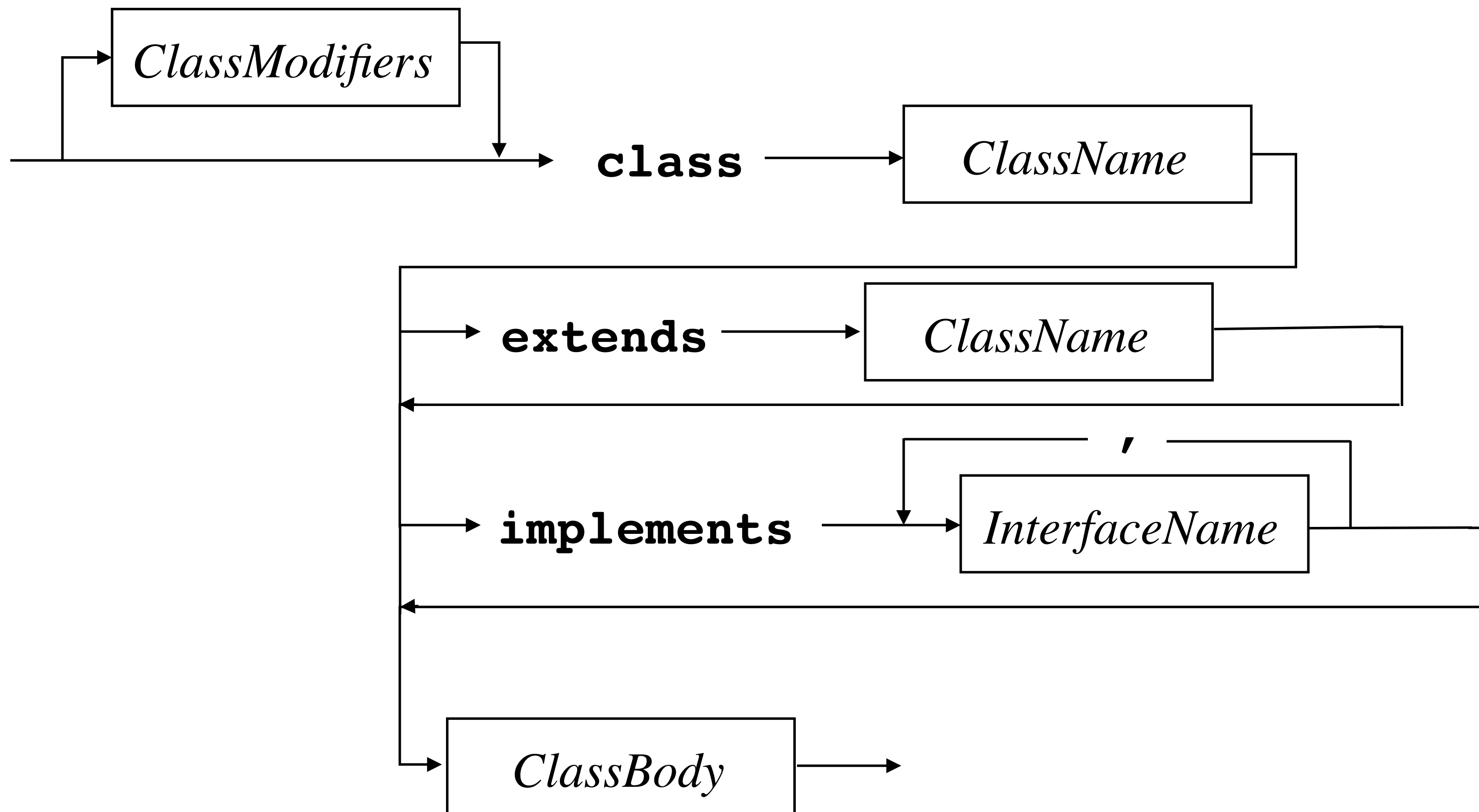
eller med for-each-løkker

```
for (int[] row : a)
    for (int elem : row)
        use elem;
```

Klasseerklæringer

```
[ClassModifiers] class ClassName
        [extends SuperClass]
        [implements Interface1, Interface2  ...] {
        ClassMemberDeclarations
}
```

Syntaksdiagram for klasseerklæring



Klassemodifikatorer

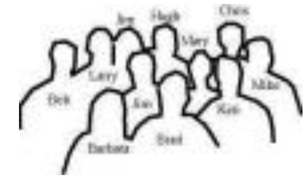
`public`
  Tilgængelig overalt.
  Kun én offentlig klasse per fil.
  Filen skal hedde *ClassName*`.java`

`private`
  Kun tilgængelig inden for samme fil.

Hverken **`public`** eller **`private`**
  Tilgængelig inden for samme pakke.

`abstract`
  Indeholder abstrakte metoder.

`final`
  Må ikke have underklasser.



Klassemedlemmer

Medlemmer af en klasse udgøres af

felter
metoder
indre klasser

Deres indbyrdes rækkefølge er ligegyldig.

Felt- og metodeerklæringer

```
[FieldModifiers] Type  FieldName1  [= Initializer1],
                       FieldName2  [= Initializer2] ...;
```

```
[MethodModifiers] ReturnType MethodName([ParameterList])  {
    Statements
}
```

ReturnType kan være `void`

Felt- og metodemodifikatorer

`public`
Tilgængelig overalt.

`private`
Kun tilgængelig i klassen selv.

`protected`
Kun tilgængelig i klassen selv, dennes underklasser og klasser i samme pakke.

Hverken **`public`**, **`private`** eller **`protected`**
Kun tilgængelig i samme pakke.

`final`
Et **`final`** felt har en konstant værdi.
En **`final`** metode kan ikke overskrives (omdefineres i underklasser).

`static`
Et **`static`** felt deles af alle instanser af klassen.
En **`static`** metode må kun tilgå klassens statiske felter.

Øvrige metodemodifikatorer

- Kun for metoder:

  `abstract`
  Implementeringen er udsat (og overladt til underklasser).

  `synchronized`
  Udføres atomisk i et flertrådet program.

  `native`
  Er implementeret i C eller C++.

Øvrige feltmodifikatorer

- Kun for felter:

`transient`
Er ikke del af klasseinstansernes persistente tilstand (serialiseres ikke).

`volatile`
Kan blive modificeret samtidigt af usynkroniserede metoder. Garanterer, at enhver tråd, der læser feltet, vil se den sidst skrevne værdi.

Skabelse og initialisering af objekter

Initialisering af felter:

Eksplicit: `double x = 3.14;`
Felter, der ikke initialiseres eksplicit, initialiseres automatisk med deres standardstartværdi.

Med konstruktører:
En konstruktør er en metode uden returværdi og med samme navn som klassen.

Med en initialiseringsblok:
En sætningsblok, der er placeret i klassen uden for enhver metode og konstruktør.

Konstruktører

```
public class Point {
    int x, y;

    public Point() {            // no-arg contructor
        x = 0; y = 0;
    }

    public Point(int x0, int y0) {
        x = x0; y = y0;
    }
}
```

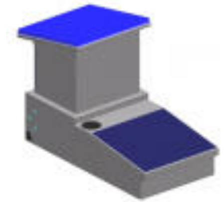
Konstruktører kan **overlæsses**:
En klasse kan have mange konstruktører, blot de har et forskelligt antal parametre, eller deres parametertyper er forskellige.

Skabelse af `Point`-objekter

```
Point p1 = new Point();

Point p2 = new Point(13, 17);
```

En klasse *uden brugerdefineret konstruktør* forsynes implicit med en konstruktør, der er uden parametre og ikke foretager sig noget.

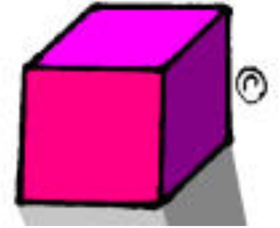
Initialiseringsblok



```
public class Point {
    public int x, y;

    {
        System.out.println("new point");
    }

    public Point() {
        x = 0; y = 0;
    }

    public Point(int x0, int y0) {
        x = x0; y = y0;
    }
}
```

En initialiseringsblok udføres før enhver konstruktør.

Statisk initialiseringsblok



```
public class Point {
    public int x, y;
    static int points;

    static {
        points++;
    }

    public Point() {
       x = 0; y = 0;
    }

    public Point(int x0, int y0) {
        x = x0; y = y0;
    }
}
```

En statisk initaliseringsblok kan kun referere til klassens statiske medlemmer.

Tilgang til medlemmer

Members

Tilgang til felter og metoder sker med **priknotation**:

objectReference.*field*
objectReference.*method*(*Arguments*)

```
Point p = new Point();

double x1 = p.x;
p.move(10, 20);
```

Implementering af metoder

Hvis returtypen er `void`, må en eventuel `return`-sætning i metoden ikke returnere en værdi.

Hvis returtypen **ikke** er `void`, skal enhver vej i metoden afsluttes med en `return`-sætning med et udtryk, der matcher metodens returtype.

Lokale variable i en metode initialiseres **ikke** til deres standardstartværdi. Lokale variable **skal** initialiseres eksplicit, inden de bruges.

Parameteroverførsel

Alle parametre i Java overføres **by value**.

Eventuelle ændringer af en parametervariabel udføres på en **kopi** af den aktuelle parameter.

```
void inc(int i) {
    i++;
}

int i = 0;
inc(i);
// i is still zero
```

Statiske felter og metoder

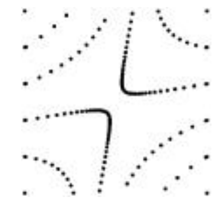
Som standard er felter i en klasse **instansfelter**.
Hvert objekt har sin egen kopi af feltet.

Et felt, der er erklærer `static`, deles af af **alle** instanser af klassen. Der en kun én kopi af et statisk felt.

En metode, der er erklæret `static`, kan kun anvende klassens statiske felter og statiske metoder.

Statiske metoder kaldes **klassemetoder**.
Ikke-statiske metoder kaldes **instansmetoder**.

Eksempel



```
public class Point {
    public int x, y;
    private int moveCount;
    private static int globalMoveCount;

    public void move(int dx, int dy) {
        x += dx; y+= dy;
        moveCount++;
        globalMoveCount++;
    }

    public int getMoveCount()
      { return moveCount; }

    public static int getGlobalMoveCount()
      {  return globalMoveCount; }
}
```

Tilgang til statiske felter og metoder

objectReference.*staticField*
objectReference.*staticMethod*(*Parameters*)

eller bedre

ClassName.*staticField*
ClassName.*staticMethod*(*Parameters*)

```
Math.PI
Math.sqrt(2.0);
```

Konstanter

Konstanter erklæres som `final static`

```
public class Font {
    public final static int PLAIN  = 0;
    public final static int BOLD   = 1;
    public final static int ITALIC = 2;
    // ...
}
```

Konvention: Konstanter skrives med store bogstaver.

Konstante referencer

```
public class Color {
    public final static Color black  = new Color(0, 0, 0);
    public final static Color blue   = new Color(0, 0, 255);
    public final static Color gray   = new Color(128, 128, 128);
    public final static Color white  = new Color(255, 255, 255);
    public final static Color yellow = new Color(255, 255, 0);
    // ...
}
```

Red Green Blue

Opregningstyper

En opregningstype (enumerated type) er en type, for hvilken alle værdier er kendt, når typen defineres.

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

Suit currentSuit = ...;
if (currentSuit == Suit.DIAMONDS)
```

En opregningstype er *typesikker*. Det er ikke tilladt at tildele en reference til typen andre værdier end de for typen definerede konstanter.

Opregningstyper
(fortsat)

Enhver opregningstype *E* har følgende statiske metoder

```
public static E[] values()
```
returnerer et array indeholdende typens konstanter i den rækkefølge, de er erklæret.

```
public static E valueOf(String name)
```
returnerer konstanten med det givne navn.

Opregningstyper er klasser og kan defineres med egne felter, metoder og konstruktører.

Læse mere her: http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html

Singleton - et designmønster

En klasse, der kun kan have én instans, kaldes for en **singleton-klasse**. Eksempel: et topniveau-vindue

```
public class Singleton {
    public static Singleton getInstance() {
        if (theInstance == null)
            theInstance = new Singleton();
        return theInstance;
    }

    private Singleton() {
        // Initializing instance fields
    }

    // Instance fields and methods

    private static Singleton theInstance = null;
}
```

Skabelse af et Singleton-objekt

```
Singleton mySingletonObject = Singleton.getInstance();
```

Eventuelle efterfølgende kald af `getInstance` vil returnere en reference til det samme objekt.

Objektreferencen `this`

Nøgleordet `this` kan bruges i en instansmetode til at referere til det objekt, som er modtager af kaldet, samt i en konstruktør til at referere til det objekt, der bliver skabt.

Eksempel på anvendelse:

```
public class Point {
    int x, y;

    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
```

Tillader tilgang til "skyggede" felter.

Overførsel af `this` som parameter

```
class Line {
    Point p1, p2;

    public Line(Point p1, Point p2) {
        this.p1 = p1; this.p2 = p2;
    }
}


public class Point {
    int x, y;

    public Line connect(Point otherPoint) {
        return new Line(this, otherPoint);
    }
}
```

Håndtering af aliaser

```
public class Point {
    int x, y;

    public Line connect(Point otherPoint) {
        if (this == otherPoint)
            return null;
        return new Line(this, otherPoint);
    }
}
```

Grænseflader (interfaces)

Et interface kan opfattes som en klasse uden implementation.

Implementeringen udsættes til de klasser, der *implementerer* interfacet.

Syntaks for grænseflader

[*ClassModifiers*] **`interface`** *InterfaceName*
[**`extends`** $Interface_1$, $Interface_2$, ...] **`{`**
InterfaceMemberDeclarations
`}`

Et medlem kan være enten en abstrakt metode eller en konstant.

De abstrakte metoder erklæres således:

ReturnType *MethodName*`(`[*ParameterList*]`);`

Alle medlemmer er offentlige.

Eksempler på grænseflader

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

```
interface Iterator {
    boolean hasNext();
    Object next();
    Object remove();
}
```

Implementering af grænseflader

Klasser kan implementere en grænseflade ved at definere ("overskrive") grænsefladens metoder.

```
interface Drawable {
    void draw(Graphics g);
}

public class Line implements Drawable {
    Point p1, p2;

    public void draw(Graphics g) {
        g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }
}
```

Abstrakte klasser

Klasser kan erklære **abstrakte metoder** ved hjælp af nøgleordet `abstract`.

`abstract` [*MethodModifiers*] *ReturnType* *MethodName*`(`[*ParameterList*]`);`

En **abstrakt** klasse er en klasse, der indeholder eller nedarver mindst en abstrakt metode. En sådan klasse **skal** erklæres med modifikatoren `abstract`.

Hverken et interface eller en abstrakt klasse kan have instanser.

Strenge

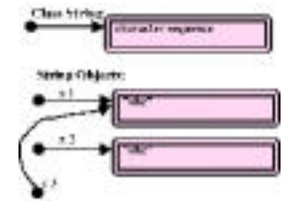
En **streng** er en sekvens af tegn.

I Java repræsenteres strenge som objekter fra en af følgende to klasser:

`String`:
konstante (immutable) strenge

`StringBuffer`:
ikke-konstante (mutable) strenge

Klassen `String`



To særlige egenskaber:

(1) Objekter kan skabes ved hjælp af strengkonstanter

```
String s = "Dette er en streng";
```

(2) Operatorerne + og += kan bruges til at sammensætte strenge

```
String s1 = "Dette er ";
String s2 = "en streng";
String s = s1 + s2;
```

De enkelte tegn i en streng er indekseret fra 0 til n-1, hvor n er længden af strengen.

Operationer på strenge

```
s.length()
s.charAt(i)
s.indexOf(c)
s.indexOf(c, i)
s1.indexOf(s2)
s1.indexOf(s2, i)
s.substring(i)
s.substring(i, j)
```

```
s.toLowerCase()
s.toUpperCase()
s.toCharArray()
s.trim()
s1.endsWith(s2)
s1.startsWith(s2)
s1.equals(s2)
s1.equalsIgnoreCase(s2)
s1.compareTo(s2)
s.intern()
s.split(regex)
```

`toString`-metoden

`toString`-metoden tillader en klasse at definere en `String`-repræsentation af sine objekter.

```
public class Point {
    int x, y;

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

Hvis `str` en `String`-objekt, og `obj` er et objekt, vil udtrykket

```
str + obj
```

blive fortolket som

```
str + obj.toString()
```

Indlæsning og udskrivning af strenge

Standardstrømme er erklæret i klassen `System`:

```
public class System {
    public static final InputStream in;
    public static final PrintStream out;
    public static final PrintStream err;
}
```

Kopiering fra `in` til `out`

java Copy < in.txt > out.txt

```
import java.io.*;

public class Copy {
    public static void main(String[] args) {
        try {
            BufferedReader in = new BufferedReader(
                                    new InputStreamReader(
                                        System.in));
            String line;
            while ((line = in.readLine()) != null)
                System.out.println(line);
        } catch (IOException e) {}
    }
}
```

Decorator

Kopiering af en tekstfil

java CopyTextFile in.txt out.txt

Kommandolinje-argumenter

```
import java.io.*;

public class CopyTextFile {
    public static void main(String[] args) {
        if (args.length >= 2) {
            try {
                BufferedReader in = new BufferedReader(
                                        new FileReader(args[0]));
                PrintWriter out = new PrintWriter(
                                      new BufferedWriter(
                                          new FileWriter(args[1])));
                String  line;
                while ((line = in.readLine()) != null)
                    out.println(line);
                out.close();
            } catch (IOException e) {}
        }
    }
}
```

Klassen `StringBuffer`

```
public final class StringBuffer implements Serializable {
    public StringBuffer();
    public StringBuffer(int length);
    public StringBuffer(String str);

    public StringBuffer append(...);
    public char charAt(int i);
    public StringBuffer delete(int start, int end);
    public StringBuffer deleteCharAt(int i);
    public StringBuffer setCharAt(int i, char c);
    public StringBuffer insert(int offset, ...);
    public StringBuffer replace(int start, int end, String str);
    public int length();
    public String subString(int start);
    public String subString(int start, int end);
    public StringBuffer reverse();
    public String toString();
    // ...
}
```

Opdeling af en streng i dens grundbestanddele (tokens)

Klassen `StringTokenizer`:

```
package java.util;

public class StringTokenizer {
    public StringTokenizer(String str);
    public StringTokenizer(String str, String delim);
    public StringTokenizer(String str, String delim,
                                         boolean returnDelims);

    public boolean hasMoreTokens();
    public String nextToken();
    public int countTokens();
}
```

Opdeling af en indlæst tekst i ord

```
import  java.io.*;
import  java.util.*;

public class Words {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
        try {
            String line;
            String delim = " \t\n.,:;?!-/()[]\"\'";
            while((line =  in.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(line, delim);
                while(st.hasMoreTokens())
                    System.out.println(st.nextToken());
            }
        } catch(IOException e) {}
    }
}
```

Klassen Scanner

```
package java.util;

public class Scanner {
    public Scanner(File source);
    public Scanner(String source);

    public boolean hasNext();
    public boolean hasNextLine();
    public boolean hasNexttype();

    public String next();
    public String nextLine();
    public type nexttype();

    public void useDelimiter(String pattern);
}
```

Wrapper-klasser

Eftersom værdier af primitive typer *ikke* er objekter, tilbyder Java en række klasser, der kan bruges til at pakke sådanne værdier ind i objekter.

Primitiv type	Wrapper-klasse
`boolean`	`Boolean`
`byte`	`Byte`
`char`	**`Character`**
`double`	`Double`
`float`	`Float`
`int`	**`Integer`**
`long`	`Long`
`short`	`Short`

Et wrapper-objekt tildeles en værdi ved sin konstruktion. Herefter kan værdien ikke ændres!

Wrapper-klassen `Integer`

```
public final class Integer
             extends Number implements Comparable {
    public Integer(int value);
    public Integer(String s)
                   throws NumberFormatException;

    public int intValue();

    public static Integer valueOf(String s)
                  throws NumberFormatException;
    public static int parseInt(String s)
                  throws NumberFormatException;
    public final static int MIN_VALUE = -2147483648;
    public final static int MAX_VALUE =  2147483647;
    // ...
    private int value;
}
```

Boxing-konverteringer

Automatisk konvertering af en værdi af primitiv type til en instans af sin wrapper-klasse kaldes *boxing*.

```
Integer val = 7;
```

er ækvivalent med

```
Integer val = new Integer(7);
```

Automatisk konvertering af instans af en wrapper-klasse til en værdi af primitiv type kaldes *unboxing*.

```
int x = val;
```

er ækvivalent med

```
int x = val.intValue();
```

Matematiske konstanter og funktioner

Tilgængelige som statiske felter og metoder i klassen `Math`:

```
E, PI,
sin, cos, tan, asin, acos, atan, atan2,
exp, pow, log
sqrt
rint, ceil, floor, round
random
abs,
min, max
```

Flere af metoderne er overlæssede.

Pakker

En pakke er en samling af relaterede klasser, grænseflader eller andre pakker.

Pakker bruges til at organisere store programmer, så de bliver håndterbare.

Alle klasser i samme fil tilhører den samme pakke.
Pakkens navn kan angives således i starten af filen:

`package` *packageName* **`;`**

Hvis pakkeerklæringen udelades, tilhører alle klasser i filen en *unavngiven* pakke.

Eksempel på konstruktion af en pakke

Filen Point.java:

```
package geometry;

public class Point {
    public int x, y;
    // ...
}
```

Filen Line.java:

```
package geometry;

public class Line {
    Point p1, p2;
}
```

Eksempel på anvendelse af pakken

(1) Importering af pakken og brug af fuldt kvalificeret navn:

```
import geometry;

geometry.Point p = new geometry.Point(3, 4);
```

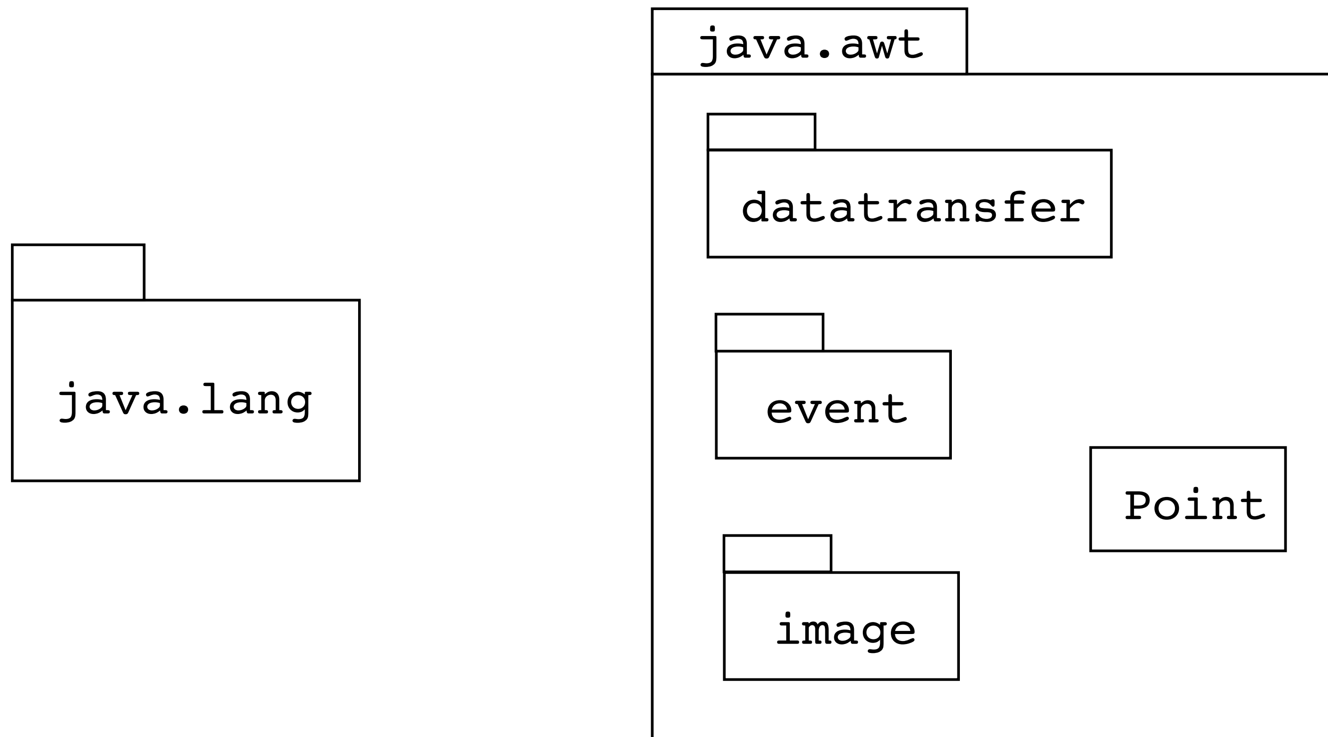
(2) Importering af en specifik klasse:

```
import geometry.Point;
Point p = new Point(3, 4);
```

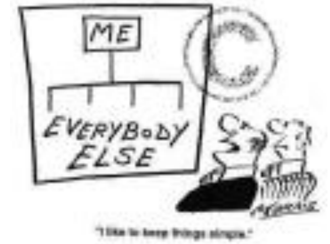
(3) Importering af alle pakkens klasser:

```
import geometry.*;
Point p1 = new Point(3, 4),
p2 = new Point(6, 9);
Line l = new Line(p1, p2);
```

Grafisk notation for pakker



Pakker og katalogstrukturer



Brug af en pakke forudsætter en katalogstruktur, der svarer til pakkens navn.

Eksempel:

Pakken

```
dk.ruc.jDisco
```

skal placeres i katalogstrukturen

```
dk/ruc/jDisco
```

Organisering af Javas klassebiblioteker

java.applet
java.awt
java.beans
java.io
java.lang
java.math
java.net
java.rmi
java.security
java.sql
java.text
java.util
java.vecmath

javax.activation
javax.mail
javax.media
javax.naming
javax.servlet
javax.swing

org.omg.CORBA

Java 5.0:
3562 klasser og grænseflader
i 166 pakker

Javas klassebiblioteker

JDK version (år)	Pakker	Klasser
1.0 (1996)	8	212
1.1 (1997)	25	504
1.2 (1998)	59	1520
1.3 (2000)	samme	samme
1.4 (2002)	135	2991
5.0 (2004)	166	3562

Undtagelser

En **undtagelse** (exception) repræsenterer en **uventet** begivenhed, typisk en fejlsituation.

I mange situationer er det hverken muligt eller hensigtsmæssigt at håndtere en fejl i umiddelbar nærhed af det sted, hvor fejlen opstod.

Gennem sit udtagelsesbegreb tillader Java adskillelse af fejldetektering og fejlhåndtering.

Kast af undtagelser

Når en undtagelse forekommer, **kastes** en undtagelse.

En undtagelse kan stamme fra to kilder:

(1) Køretidssystemet (JVM), ved udførelse af en ulovlig operation (bl.a. `NullPointerException`, `ArrayOutOfBoundException`)

(2) Java-programmet (inklusiv Javas klassebibliotek), såfremt der eksplicit kastes en undtagelse.

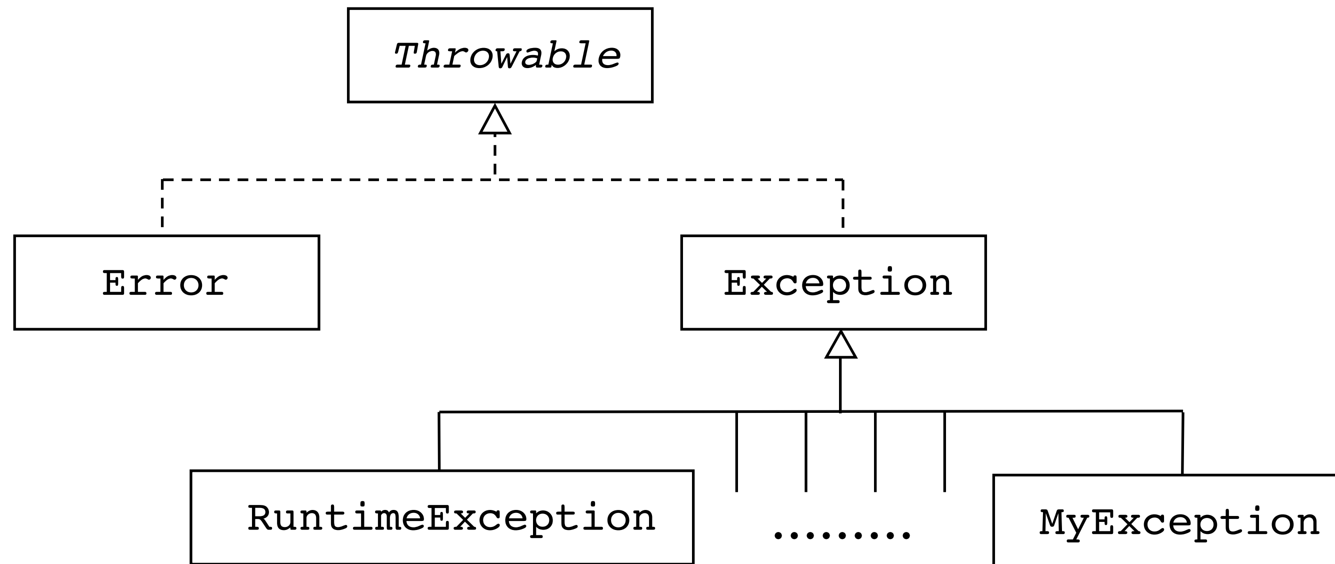
`throw`-sætningen

En undtagelse repræsenteres som et objekt og kastes med en `throw`-sætning:

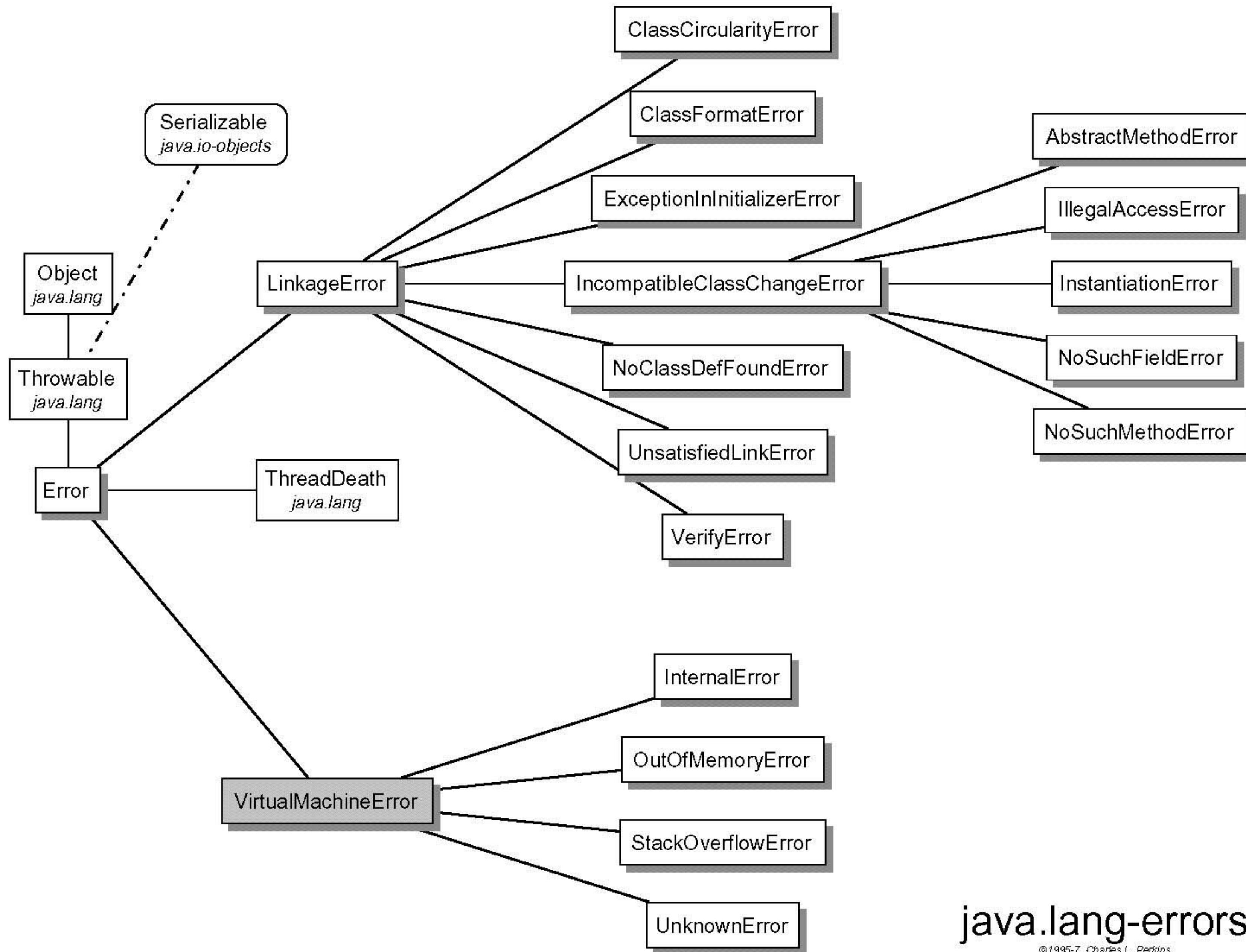
`throw` *exception*`;`

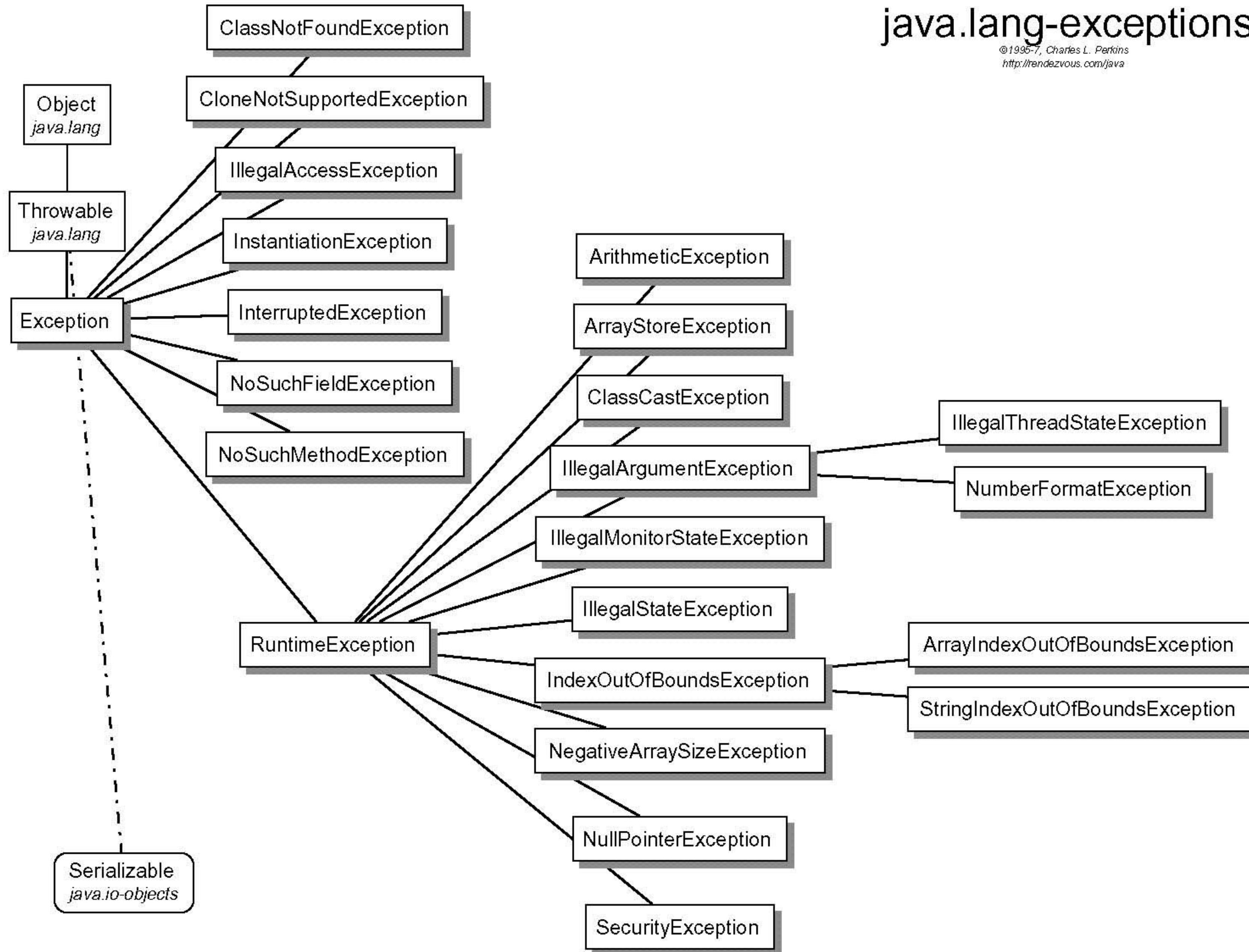
hvor *exception* er en instans af en klasse, der er en underklasse af `Throwable`.

Hierarkiet af undtagelser



Undtagelser af typen `Error` og `RuntimeException` er **ucheckede** (håndtering er ikke nødvendig).
Alle øvrige er **checkede** (håndtering er tvungen).
Oversætteren checker, at de ikke ignoreres.


Serializable
java.io-objects
Object
java.lang
Throwable
java.lang
Error
LinkageError
ThreadDeath
java.lang
VirtualMachineError
ClassCircularityError
ClassFormatError
ExceptionInInitializerError
IncompatibleClassChangeError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
java.lang-errors
©1995-7, Charles L. Perkins
http://rendezvous.com/java


java.lang-exceptions
©1995-7, Charles L. Perkins
http://rendezvous.com/java
Object
java.lang
Throwable
java.lang
Exception
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchFieldException
NoSuchMethodException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalThreadStateException
NumberFormatException
IllegalMonitorStateException
IllegalStateException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
Serializable
java.io-objects

Checkede undtagelser

Enhver metode **skal** specificere, hvilke checkede undtagelser, den kan kaste.

[*MethodModifiers*] *ReturnType MethodName* **(**[*ParameterList*]**)**
        `throws` *Exception*$_1$, *Exception*$_2$ ... **{**
    Statements
}

try-catch-sætningen

Sætninger, der kan kaste en undtagelse, kan placeres i en `try-catch`-sætning:

```
try {
    // ...
    throw new MyException();
    // ...
} catch (MyException e) {
    // handle the exception e
}
```

Hvis `method` kan kaste en undtagelse af typen `MyException`:

```
try {
    method();
} catch (MyException e) {
    // handle the exception e
}
```

Fangst af undtagelser

```
try {
    // statements that may throw exceptions
} catch (Exception1 e1) {
    // exception handler 1
} catch (Exception2 e2) {
    // exception handler 2
} finally {
    // finish up
}
```

Hvis en undtagelse kastes, søges sekventielt i den nærmeste `try-catch`-sætning. Hvis der findes en parametertype, der matcher undtagelses-typen, udføres den tilsvarende `catch`-blok. Ellers søges videre i den næstnærmeste `try-catch`-sætning, o.s.v.

`finally`-blokken er frivillig. Den udføres altid som det sidste, inden `try-catch`-sætningen afsluttes.

Behandling af undtagelser

Der er principielt 3 måder, hvorpå en undtagelse kan behandles:

(1) Den kan ignoreres
(2) En ny undtagelse kan kastes
(3) Programmet kan stoppes

```
try {
    // statements that may throw exceptions
} catch (Exception1 e1) {
    // ignore
} catch (Exception2 e2) {
    throw new MyException();
} catch (Exception3 e3) {
    System.exit(0);
}
```

Præcis beskrivelse af Java

J. Gosling, B. Joy & G. L. Steeele:
The Java Language Specification
http://java.sun.com/docs/books/jls/

Ugeseddel 2

7. - 14. september

- Læs kapitel 5 og 6 i lærebogen (side 159 - 247)

- Løs opgave 4.3 samt opgaven på de næste sider

Opgave 4.3 kan løses ved hjælp af enten

(a) klassen `StringTokenizer`,

(b) metoden `split` fra klassen `String`,

(c) klassen `Scanner` eller

(d) klassen `StreamTokenizer`

Løs opgaven under antagelse af, at inddata er korrekte

Ekstraopgave 1

Spørgsmål 1

Programmér en klasse, `Fraction`, der repræsenterer rationale tal (brøker). Felterne skal være to `int`-variable, der lagrer henholdsvis tæller og nævner for en brøk.

Forsyn klassen med en passende mængde af konstruktører og operationer, bl.a. `add`, `subtract`, `multiply`, `divide`, `toString`, `equals` og `compareTo`.

Gem ethvert rational tal på forkortet form og med en nævner, der altid er positiv. Nedenstående metode, der bestemmer største fælles divisor for to positive heltal `a` og `b`, kan benyttes til at forkorte en brøk:

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

Sørg for at klassen kan håndtere negative rationale tal.

fortsættes

Spørgsmål 2

Benyt klassen i et program, der beregner og udskriver værdien af summen 1 + 1/2 + 1/3 + … + 1/9 + 1/10 som en brøk.

Spørgsmål 3

Den udviklede klasse kan kun håndtere brøker, hvis tæller og nævner kan repræsenteres i 32 bit. Benyt Javas klasse `BigInteger` til at implementere en ny udgave af `Fraction`, der ikke har denne begrænsning.

Spørgsmål 4

Benyt den nye klasse i et program, der beregner og udskriver værdien af summen 1 + 1/2 + 1/3 + …+ 1/99 + 1/100 som en brøk.