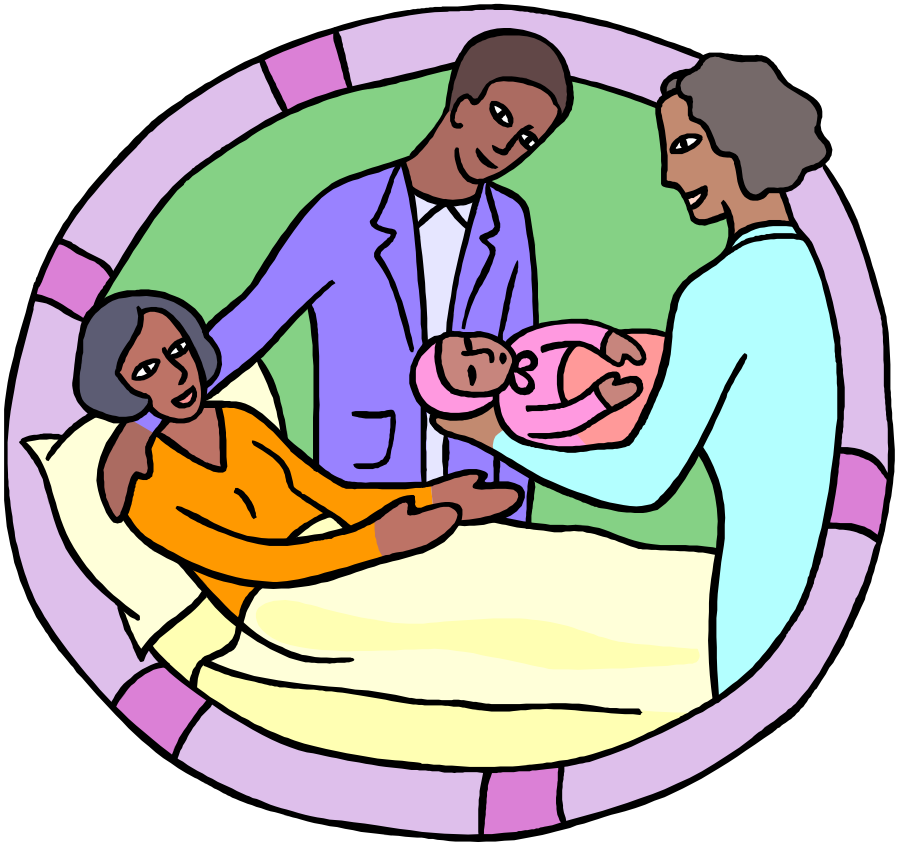
Nedarvning



Plan

- Overlæsning af metoder og konstruktører
- Nedarvning fra klasser
- Implementering af grænseflader
- Retningslinjer for design af klasser
- Animering i appletter

Overlæsning

Java tillader mulighed for at metoder og konstruktører kan deles om det samme navn. Navnet siges da at være **overlæsset** med flere implementationer.

Lovligheden af en overlæsning afgøres af metodernes og konstruktørernes **signatur**, d.v.s. sekvensen af parametrenes typer.

Bemærk at returtypen ikke indgår i signaturen.

Signaturer

Metode	Signatur
`String toString()`	`()`
`void move(int dx, int dy)`	`(int, int)`
`void paint(Graphics g)`	`(Graphics)`

Hvis to metoder eller konstruktører har forskellige signaturer, må de gerne dele det samme navn.

Eksempel på overlæsning (kontruktører)

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

    public Point() {
        x = 0.0; y = 0.0;
    }

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

    ...
}
```

Eksempel på overlæsning (metoder)


```
public class Point {
    double x, y;
    // ...

    public double distance(Point other) {
        double dx = this.x - other.x, dy = this.y - other.y;
        return Math.sqrt(dx*dx + dy*dy);
    }

    public double distance(double x, double y) {
        double dx = this.x - x, dy = this.y - y;
        return Math.sqrt(dx*dx + dy*dy);
    }

    public double distance(int x, int y) {
        double dx = this.x - x, dy = this.y - y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

Alternativ implementering

```
public class Point {
    double x, y;
    // ...

    public double distance(double x, double y) {
        double dx = this.x - x, dy = this.y - y;
        return Math.sqrt(dx*dx + dy*dy);
    }

    public double distance(Point other) {
        return distance(other.x, other.y);
    }

    public double distance(int x, int y) {
        return distance((double) x, (double) y);
    }
}
```

Brug af overlæsning

Overlæsning bør kun bruges, når der findes en generel beskrivelse af funktionaliteten, der passer på alle overlæssede metoder.

```
public class StringBuffer {
    StringBuffer append(String str) { ... }
    StringBuffer append(boolean b) { ... }
    StringBuffer append(char c) { ... }
    StringBuffer append(int i) { ... }
    StringBuffer append(long l) { ... }
    StringBuffer append(float d) { ... }
    StringBuffer append(double d) { ... }
    // ...
}
```

Hvorfor er returværdien ikke `void`?

Svar

For at forenkle notationen ved successive af kald af `append` på samme `StringBuffer`-object.

```
StringBuffer buf = new StringBuffer();
buf.append("A").append(38);
```

Implementation:

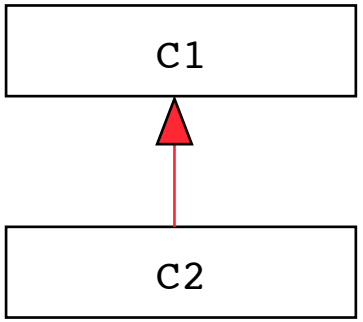
```
private char[] value;
private int count;

public StringBuffer append(String str) {
    int newCount = count + str.length();
    ensureCapacity(newCount);
    str.getChars(0, str.length(), value, count);
    count = newCount;
    return this;
}
```

Brug af overlæsning (2)

```
public class String {
    public String substring(int i, int j) {
        // base method: return substring [i..j-1]
    }

    public String substring(int i) {
        return substring(i, length());
    }
    //...
}
```

Nedarvning fra klasser

Når klassen C2 **nedarver** fra (eller **udvider**) klassen C1, siges C2 at være en **underklasse** af C1, og C1 siges at være **overklasse** for C2.

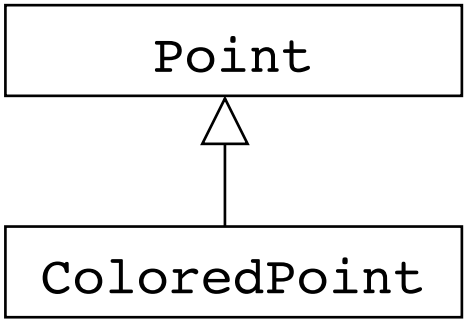


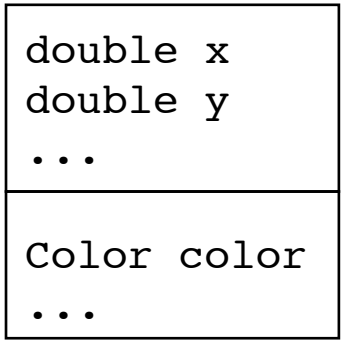

```
class C2 extends C1 {
    //...
}
```

Alle medlemmer i C1, der er `public` eller `protected`, er tilgængelige i C2.

Eksempel på nedarvning

```
class ColoredPoint extends Point {
    Color color;
    // ...
}
```





Brug af arv

En klasse, som arver fra en anden klasse, kan være en **specialisering** (typisk), en **udvidelse**, eller både en specialisering og en udvidelse af denne klasse.

Specialisering:

```
class ChessFrame extends JFrame {
    // ...
}
```

Udvidelse:

```
class ColoredPoint extends Point {
    // ...
}
```

Former for arv
(god, dårlig, kompliceret)

Specialisering. Underklassen er et specialtilfælde af overklassen.

Udvidelse. Underklassen tilføjer ny funktionalitet til overklassen, men ændrer ikke den nedarvede adfærd.

Specifikation. Overklassen definerer adfærd, som implementeres i underklassen, men ikke i overklassen.

Begrænsning. Underklassen begrænser brugen af en del af den nedarvede adfærd fra overklassen.

Konstruktion. Underklassen gør brug af overklassens adfærd, men er ikke en undertype af sin overklasse.

Kombination. Underklassen nedarver adfærd fra mere end én overklasse.

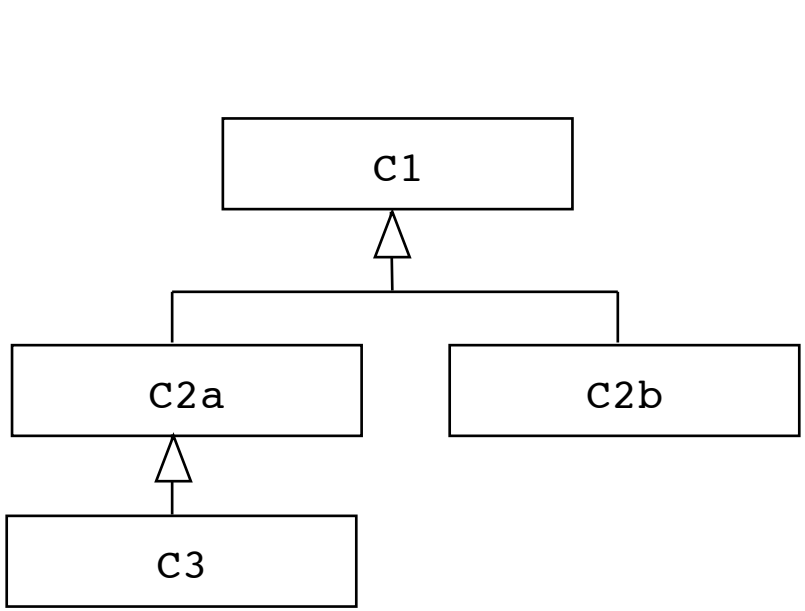
Designregel for klasser

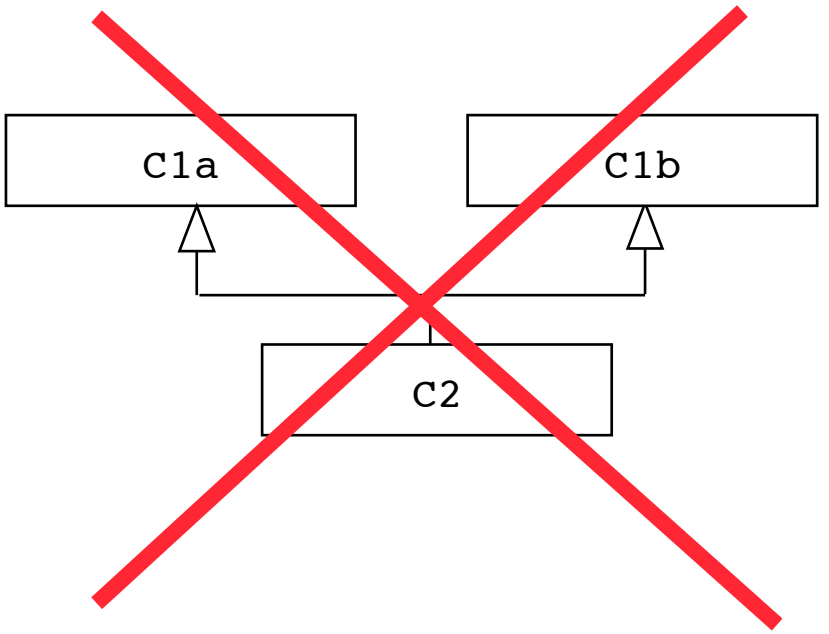
Åben-lukket-princippet: En klasse bør være åben for udvidelse, men lukket for modifikation.

Bertrand Meyer

Singulær nedarvning

I Java må enhver klasse højst have én overklasse.





Klassen `Object`

I Java er alle klasser organiseret i et hierarki, der har klassen `Object` som rod.

Enhver klasse, undtagen `Object`, har en unik overklasse.

Hvis der for en klasse ikke eksplicit defineres nogen overklasse, er `Object` dens overklasse.

Klassen `Object`

```
public class Object {
    public String toString();
    public boolean equals(Object obj);
    public int hashCode();
    protected Object clone();

    public final void notify();
    public final void notifyAll();
    public final wait()
                  throws InterruptedException;
    public final wait(long millis)
                  throws InterruptedException;
    public final wait(long millis, int nanos)
                  throws InterruptedException;
    public void finalize();
    public final Class getClass();    // reflection
}
```

Klassen `Class`

```
public class Class implements Serializable {
    public java.lang.reflect.Constructor[] getConstructors()
           throws SecurityException;
    public java.lang.reflect.Field[] getFields()
           throws SecurityException;
    public java.lang.reflect.Method[] getMethods()
           throws SecurityException;
    public Class getSuperclass();

    public static Class forName(String s)
                        throws ClassNotFoundException;

    public Object newInstance()
           throws InstantiationException,
                  IllegalAccessException;
    ...
}
```

Konstruktører for underklasser

```
import java.awt.Color;

public class ColoredPoint extends Point {
    public Color color;

    public ColoredPoint(double x, double y, Color color) {
        super(x, y);              // must be the first statement
        this.color = color;
    }

    public ColoredPoint(double x, double y) {  // black point
        this(x, y, Color.black); // must be the first statement
    }

    public ColoredPoint() {
        color = Color.black;     // invokes super() implicitly
    }
}
```

Konstruktion af egne `Exception`-klasser

```
class MyException extends Exception {}
```

eller

```
class MyException extends Exception {
    MyException() {
        super();
    }

    MyException(String s) {
        super(s);
    }
}
```

Klassen `Exception`

```
public class Exception extends Throwable {
    public Exception() {
        super();
    }

    public Exception(String s) {
        super(s);
    }
}
```

Klassen `Throwable`

```
public class Throwable {
    private String detailMessage;

    public Throwable() {
        fillInStackTrace();
    }

    public Throwable(String message){
        this();
        detailMessage = message;
    }

    public String toString() {
        String s = getClass().getName();
        String message = detailMessage;
        return message != null ? (s + ": " + message) : s;
    }

    public void printStackTrace() { ... }

    public native Throwable fillInStackTrace();
}
```

```
public class ExceptionTest {
    static void pip() throws MyException {
        throw new MyException("Oops!");
    }

    public static void main(String[] args) {
        try {
            pip();
        } catch (MyException e) {
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

```
MyException: Oops!
MyException: Oops!
        at ExceptionTest.pip(ExceptionTest.java)
        at ExceptionTest.main(ExceptionTest.java)
```

Initialisering af nedarvede klasser

```
public class Super {
    int x = ...;                // executed first

    public Super() {
        x = ...;                // executed second
    }
}

public class Extended extends Super {
    int y = ...;                // executed third

    public Extended() {
        y = ...;                // executed fourth
    }
}
```

Referencer

En variabel af referencetype indeholder en henvisning til et objekt.

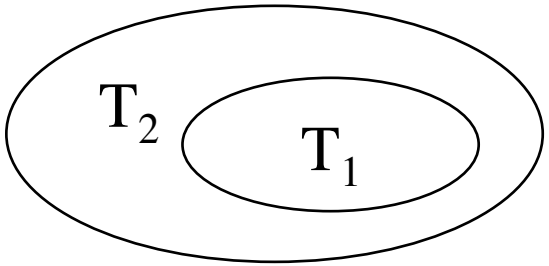
En reference kan henvise til objekter fra forskellige klasser, blot **reglen om undertyper** er overholdt.

```
Super s = new Extended();
```

Reglen om undertyper

Definition: En **type** er en mængde af værdier.

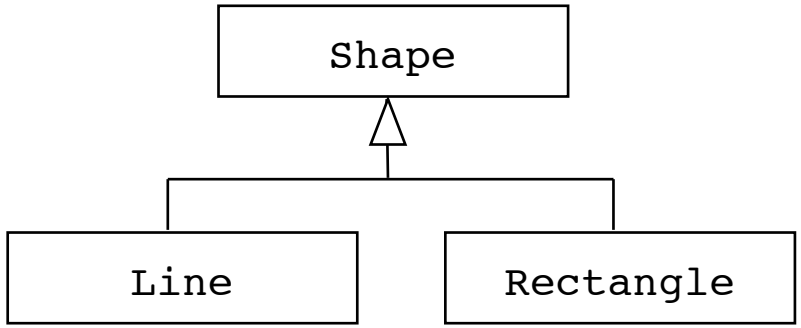
Definition: Typen T_1 er en **undertype** af typen T_2, hvis enhver værdi i T_1 også er en værdi i T_2.

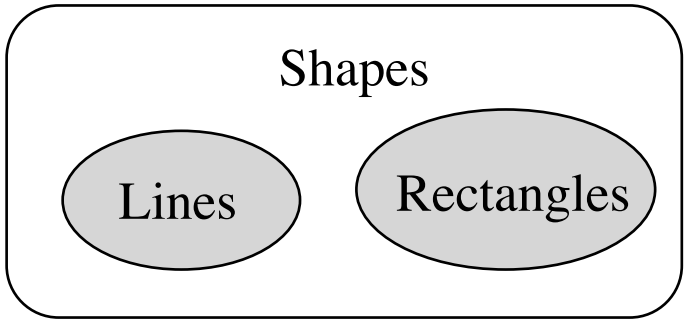


Regel: En værdi af en undertype kan optræde overalt, hvor en værdi af dens overtype forventes.

Designregel for underklasser

Liskovs substitutionsprincip: Et objekt af en underklasse bør kunne optræde overalt, hvor et objekt af dens overklasse forventes.





Er-relation imellem klasser (is-a)

Polymorf tildeling

Tildelingsreglen: Typen af et udtryk på højresiden af en tildeling skal være en undertype af typen på venstresiden.

```
Shape aShape;
Line aLine = new Line(...);
Rectangle aRectangle = new Rectangle(...);

aShape = aLine;              // ok

aShape = aShape;             // ok

aLine = aRectangle;          // compilation error

aLine = aShape;              // compilation error
```

Typekonvertering
(casting, kvalificering)

Tildelingsreglen checkes på **oversættelsestidspunktet**.

```
aLine = aShape;          // compilation error
```

Reglen kan tilfredsstilles ved at indsnævre typen på højresiden (narrowing, down casting):

```
aLine = (Line) aShape;         // ok
```

Lovligheden af tildelingen checkes på **kørselstidspunktet**.

En ulovlig indsnævring bevirker, at der kastes en `ClassCastException.`

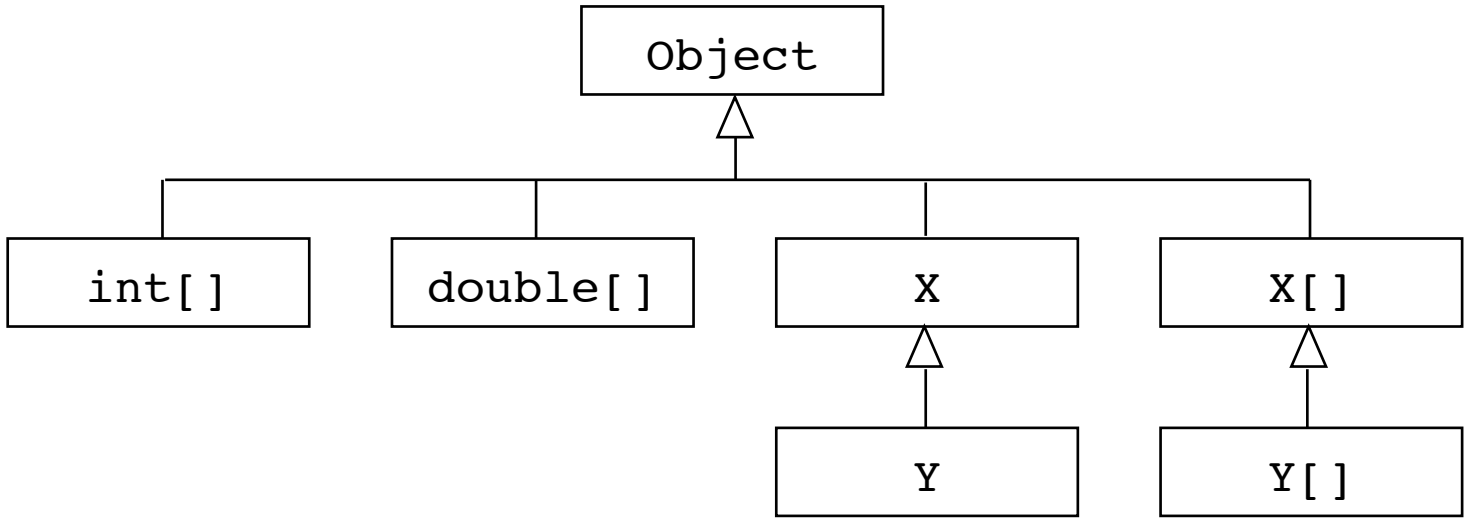
Bestemmelse af klassetilhørsforhold

Operatoren `instanceof` kan benyttes til at afgøre, hvorvidt et objekt tilhører en given klasse (eller grænseflade).

```
if (aShape instanceof Line)
    aLine = (Line) aShape;
```

Arraytyper

I Java er arrays objekter. Alle arrays er undertyper af `Object`.



Overskrivning af metoder

Overskrivning foretages, når der i en underklasse defineres en metode med samme **navn**, **signatur** og **returværdi** som en metode i overklassen. Ellers foretages overskrivning ikke.

```
class Shape {
    public String toString() {
        return "Shape";
    }
}

class Line extends Shape {
    Point p1, p2;

    public String toString() {
        return "Line from " + p1 + " to " + p2;
    }
}
```

Præcision ved overskrivning

```
class B {
    public void m(int i) { ... }
}

class C extends B {
    public void m(char c) { ... }
}
```

resulterer ikke i overskrivning, men derimod i **overlæsning**.

Giver ikke oversætterfejl, som hævdet i lærebogen (s. 172).

Præcision ved overskrivning

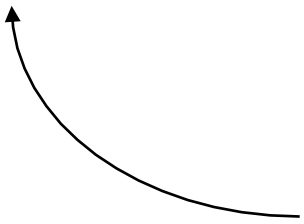
```
public interface Comparable {
    int compareTo(Object obj);
}

class Number implements Comparable {
    public int compareTo(Number number) {
        return value < number.value ? -1 :
               value > number.value ?  1 : 0;
    }

    private int value;
}
```

giver fejl på oversættelsestidspunktet:

```
Number is not abstract and does not override
abstract method compareTo(java.lang.Object) in
Comparable
```

Løsning 1

```
class Number implements Comparable {
    public int compareTo(Object obj) {
        return value < ((Number) obj).value ? -1 :
               value > ((Number) obj).value ?  1 : 0;
    }

    private int value;
}
```



Løsning 2
(med generisk type)

```
class Number implements Comparable<Number> {
    public int compareTo(Number number) {
        return value < number.value ? -1 :
               value > number.value ?  1 : 0;
    }

    private int value;
}
```

idet `Comparable` i Java 5.0 er defineret således

```
interface Comparable<T> {
    public int compareTo(T obj);
}
```

@Override

I Java 5.0 er det muligt med annotationen @Override at få oversætteren til at kontrollere, at en metode overskrives.

```
class Number {
    @Override
    public String toSting() {
        ...
    }
}
```

Stavefejl

Ved oversættelse gives fejlmeddelelsen

```
method does not override a method from its superclass
    @Override
     ^
```

Polymorfe metodekald

Hvilken implementation af en overskrevet metode, der udføres, afhænger af objekttypen og afgøres på **kørselstids-punktet** (dynamisk binding).

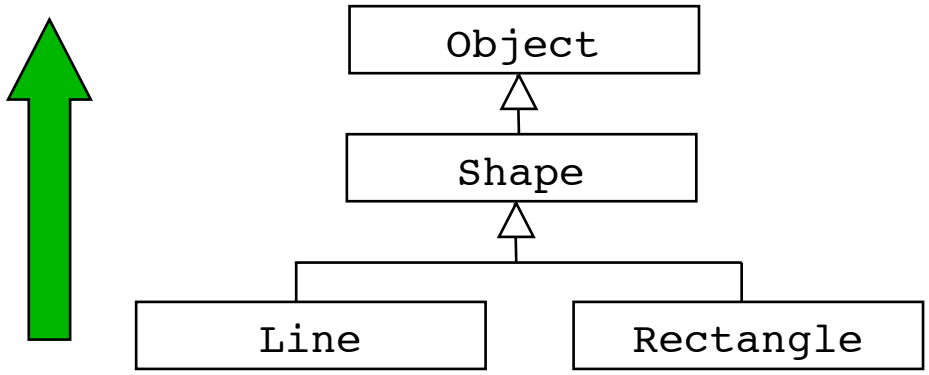
```
Shape[] shapes = new Shape[100];
shapes[0] = new Line(...);
shapes[1] = new Rectangle(...);
...

for (int i = 0; i < 100; i++)
    System.out.println(shapes[i].toString());
```

Kun instansmetoder kan kaldes polymorft.

Polymorfe metodekald

Hvilken implementation af en overskrevet metode, der udføres, afhænger af objekttypen og afgøres på kørselstids-punktet **ved søgning op igennem klassehierarkiet**, startende ved den klasse, som objektet tilhører.

Den første implementation af metoden, der mødes, udføres.



Kald af overskrevne metoder

```
class Point {
    // ...
    public boolean equals(Object other) {
        if (other instanceof Point) {
            Point p = (Point) other;
            return x == p.x && y == p.y;
        }
        return false;
    }
}

class ColoredPoint extends Point {
    //...
    public boolean equals(Object other) {
        if (other instanceof ColoredPoint) {
            ColoredPoint p = (ColoredPoint) other;
            return super.equals(p) && color.equals(p.color);
        }
        return false;
    }
}
```

Nedarvning fra og implementering af grænseflader

En grænseflade erklærer metoder uden implementation. Klasser, der implementer en grænseflade, skal angive implementationer for **alle** grænsefladens metoder (hvis klassen da ikke skal være abstrakt).

En grænseflade kan nedarve fra en eller flere andre grænseflader, men ikke fra en klasse.

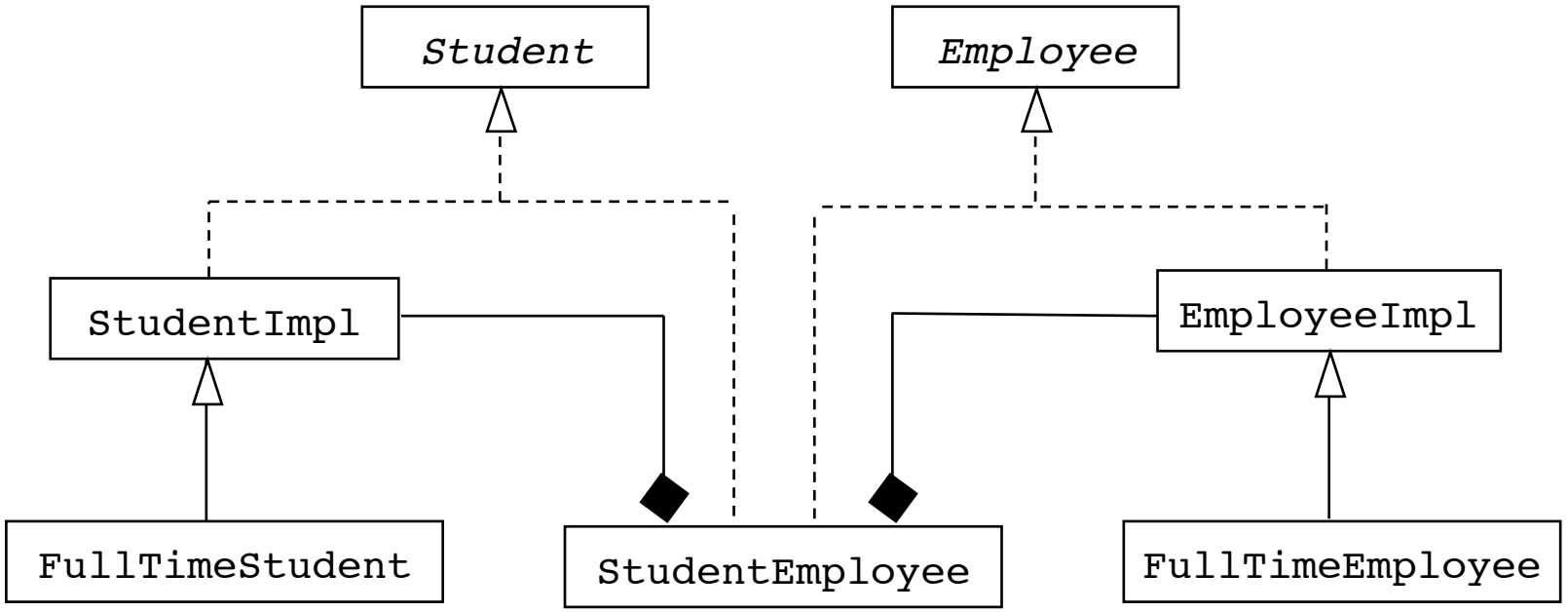
En klasse kan implementere flere grænseflader.

Implementation af flere grænseflader

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

class Rectangle implements Drawable, Movable {
    public void draw(Graphics g) {
      // draw the rectangle
    }

    public void move(double dx, double dy) {
      // move the rectangle
    }
}
```

Delegering

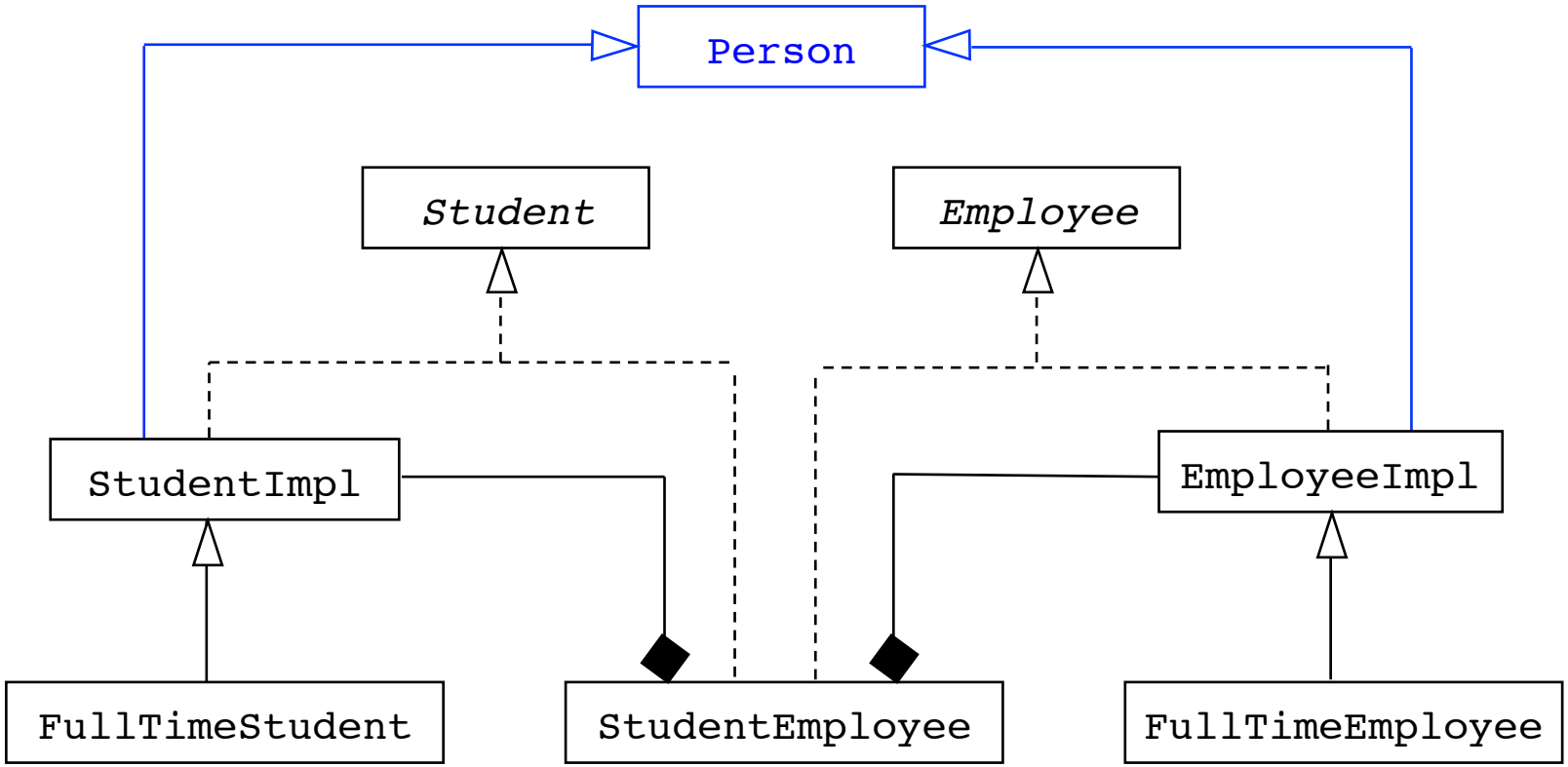
Multipel nedarvning fra klasser er ikke tilladt i Java, men kan opnås ved hjælp af **delegering**:



```
interface Student {
    float getGPA();         // Grade Point Average
}

interface Employee {
    float getSalary();
}

public class StudentImpl implements Student {
    public float getGPA() {
        return ...;         // calculate GPA
    }
}

public class EmployeeImpl implements Employee {
    public float getSalary() {
        return ...;         // calculate salary
    }
}
```

```
public class StudentEmployee implements Student, Employee {
    private StudentImpl myStudentImpl   = new StudentImpl();
    private EmployeeImpl myEmployeeImpl = new EmployeeImpl();

    public float getGPA() {
        return myStudentImpl.getGPA();          // delegation
    }

    public float getSalary() {
        return myEmployeeImpl.getSalary();      // delegation
    }
}
```

Fælles egenskaber kan samles i en overklasse



Mærkegrænseflader (marker interfaces)

En **mærkegrænseflade** er en tom grænseflade, altså en grænseflade uden metoder og konstanter.

De to mest benyttede er

```
interface Cloneable                (java.lang)
```

og

```
interface Serializable             (java.io)
```

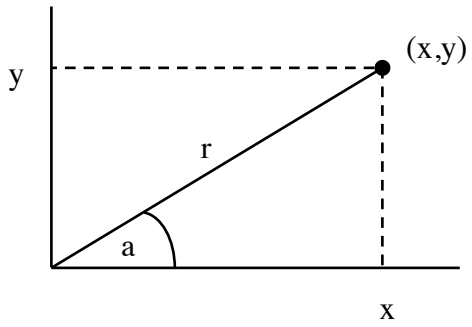
Retningslinjer ved design af klasser

(1) **Undgå offentlige felter**

Et felt må kun være offentligt, hvis klassen er `final`, og en ændring af feltets værdi frit kan foretages.

```
public class Point {
    protected double x, y;

    public double getX() { return x; }
    public double getY() { return y; }

    public void setX(double x) { this.x = x; }
    public void setY(double y) { this.y = y; }
}
```

```
public class PolarPoint extends Point {
    protected double r, a;

    public PolarPoint() {}

    public PolarPoint(double r, double a)
      { this.r = r; this.a = a; polarToRectangular(); }

    public double getRadius() { return r; }
    public double getAngle()  { return a; }
    public void setRadius(double r) { this.r = r; polarToRectangular(); }
    public void setAngle(double a)  { this.a = a; polarToRectangular(); }
    public void setX(double x)      { this.x = x; rectangularToPolar(); }
    public void setY(double y)      { this.y = y; rectangularToPolar(); }

    protected void polarToRectangular() {
        x = r * Math.cos(a);
        y = r * Math.sin(a);
    }

    protected void rectangularToPolar() {
        r = Math.sqrt(x * x + y * y);
        a = Math.atan2(y, x);
    }
}
```



(2) **Adskil grænseflade fra implementation**

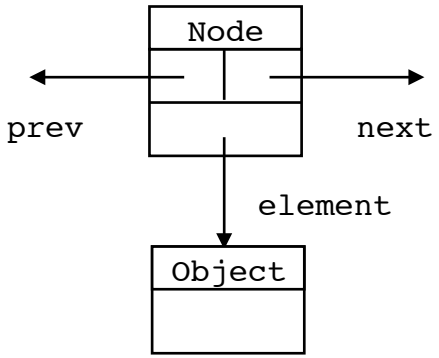
Når funktionaliteten af en klasse kan opnås på forskellig vis, bør grænsefladen adskilles fra implementationen.

```
public interface List {
    Object elementAt(int pos);
    int getCount();
    boolean isEmpty();
    void insertElementAt(Object obj, int pos);
    void removeElementAt(int pos);
}

public class LinkedList implements List {
    // body of LinkedList
}

public class ArrayList implements List {
    // body of ArrayList
}
```

(3) **Placer en klasses hjælpeklasser i samme fil som klassen**

Definer eventuelt en hjælpeklasse som en indre klasse.

```
public class LinkedList implements List {
    protected static class Node {
        Node prev, next;
        Object element;
    }
    //...
}
```



(4) **Klasser, der skal være generelt anvendelige, bør være på *kanonisk* form**

- Erklær en offentlig no-arg-konstruktør (tillader dynamisk indlæsning af klassen).
- Overskriv metoderne `toString`, `equals` og `hashCode`.
- Implementer grænsefladen `Cloneable` og overskriv, om nødvendigt, metoden `clone`.
- Implementer grænsefladen `Serializable`, hvis objekter skal gemmes i en fil eller sendes over netværk.

```
public class LinkedList implements List, Cloneable {
    public String toString() {
        StringBuffer s = new StringBuffer();
        for (int i = 0; i < getCount(); i++)
            s.append("[" + i + "] = " + elementAt(i) + "\n");
        return s.toString();
    }

    //...
}
```

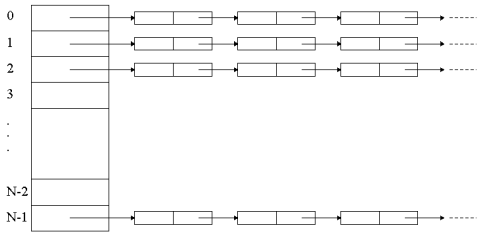
fortsættes

```
public boolean equals(Object otherList) {
    if (otherList instanceof List) {
        if (getCount() == otherList.getCount()) {
            for (int i = 0; i < getCount(); i++) {
                Object thisElement = elementAt(i);
                Object otherElement = otherList.elementAt(i);
                if ((thisElement == null && otherElement != null) ||
                    !thisElement.equals(otherElement))
                    return false;
            }
            return true;
        }
    }
    return false;
}
```

fortsættes

Om brug af hashCode

`int hashCode()`



Tænk på værdien af `hashCode` som et vink om, på hvilken plads i hashtabellen, objektet ligger.

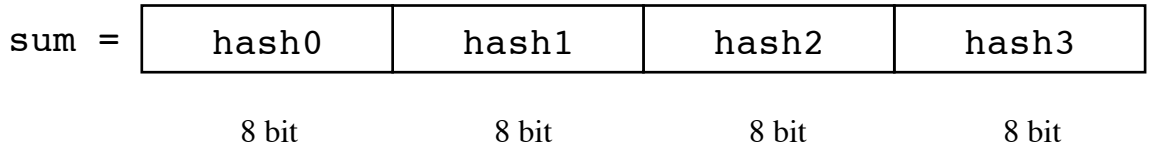
Der bør for ethvert par af objekter (`o1`, `o2`) gælde:

`o1.equals(o2)` $\Rightarrow$ `o1.hashCode() == o2.hashCode()`

Ideelt set (men ofte ikke praktisk muligt) desuden:

`!o1.equals(o2)` $\Rightarrow$ `o1.hashCode() != o2.hashCode()`

```
public int hashCode() {
    int sum = 0;
    for (int i = 0; i < 4 && i < getCount(); i++) {
        sum <<= 8;
        sum |= elementAt(i).hashCode() & 0xFF;
    }
    return sum;
}
```



Konsistent med `equals`

fortsættes

```
public Object clone()
                throws CloneNotSupportedException {
    LinkedList list = (LinkedList) super.clone();
    list.head = list.tail = null;
    list.count = 0;
    for (Node node = head; node != null; node = node.next)
        if (node.item != null)
            list.insertLast(node.item);
    return list;
}
```

Konsistent med `equals`.

Her er der tale om en **dyb** kopi ("deep copy").
Hvis der kun foretages bitvis kopiering af objektets felter
(med `super.clone()`), er der tale om en **lav** kopi ("shallow copy").

(5) **Ordn klassens medlemmer i forhold til deres tilgængelighed og roller**

```
public class TypicalClass {
    <public constants>
    <public constructors>
    <public accessors>
    <public mutators>
    <nonpublic fields>
    <nonpublic auxiliary methods and nested classes>
}
```

(6) **Sørg for at de offentlige metoder tilfredsstiller følgende to krav**:

Fuldstændighed:
Brugeren skal have adgang til klassens fulde funktionalitet.

Sikkerhed:
Et kald må aldrig føre til en ikke-konsistent tilstand.
Bevar klasseinvarians.

(7) **Afprøv hver klasse for sig (unit testing)**

Afprøvningen af en klasse kan passende foretages
i en `main`-metode i klassen.

Ekstern afprøvning (black-box test)
Kald enhver metode mindst én gang - og med passende kombinationer af mulige parameterværdier.
XP: Programmér dette før implementering af metoderne.

Intern afprøvning (white-box test)
Udfør enhver mulig vej i enhver metode mindst én gang.

Værktøjer: jUnit og jTest

Eksempel på brug af jUnit

```
import junit.framework.*;

public class FractionTest extends TestCase {
    private Fraction f1, f2, f3;

    public void setUp() {
        f1 = new Fraction(3, 4);
        f2 = new Fraction(4, 5);
        f3 = new Fraction(-3, 4);
    }

    public void testAdd() {
        assertTrue(f1.add(f2).equals(new Fraction(31, 20)));
        assertTrue(f1.add(f3).equals(new Fraction(0)));
    }

    public static void main(String[] args) {
        new junit.textui.TestRunner().run(FractionTest.class);
    }
}
```

(8) **Dokumenter kildekoden med `javadoc`**

Eksempel på en `javadoc` kommentar:

```
/**
* Retrieves the element from the LinkedList
* at the position pos
*
*@param pos The position
*@return    The retrieved element
*@see       #insertElementAt(Object element, int pos)
*/
public Object elementAt(int pos) {
```

Andre specielle mærker (tags):

```
@author
@version
@exception  (eller @throws)
```

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: INNER | FIELD | CONSTR | METHOD

FRAMES NO FRAMES
DETAIL: FIELD | CONSTR | METHOD

Class LinkedList

```
java.lang.Object
  |
  +--LinkedList
```

public class **LinkedList**
extends java.lang.Object

Constructor Summary

LinkedList()

Method Summary

java.lang.Object	elementAt(int pos) Retrieves the element from the LinkedList at the position pos
void	insertElementAt(java.lang.Object element, int pos)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

LinkedList

```
public LinkedList()
```

Method Detail

elementAt

```
public java.lang.Object elementAt(int pos)
```

Retrieves the element from the LinkedList at the position pos

Parameters:

pos - The position

Returns:

The retrieved element

See Also:

insertElementAt(Object element, int pos)

---

insertElementAt

```
public void insertElementAt(java.lang.Object element,
                            int pos)
```

---

(9) **Angiv kontrakter og klasseinvarianter**

Eksempel på kontrakt:

```
/**
* Retrieves the element from the LinkedList
* at the position pos
*
*@pre           pos >= 0 && pos < size()
*@post          @nochange
*/
public Object ElementAt(int pos)
```

Værktøj: jContract

(10) **Benyt påstande (assertions) aggressivt**

Defensiv programmering:
Hver metode bør afprøve sine pre- og postbetingelser samt klassens invarianter.

```
public Object ElementAt(int i) {
    assert i >= 0 && i < size();
    ...
}
```

Ny i JDK 1.4
Kan kaste en `AssertionError` undtagelse

Frameworks
(programskeletter)

Et **framework** (ramme, skelet, stel) består typisk af en mængde af abstrakte klasser og grænseflader, der udgør delene i halvfærdige programmer.

Et framework anvendes ved at udfylde dets "huller".

Kontrollen ligger typisk i frameworket.

Et eksempel er klassen `java.applet.Applet`.

Appletter

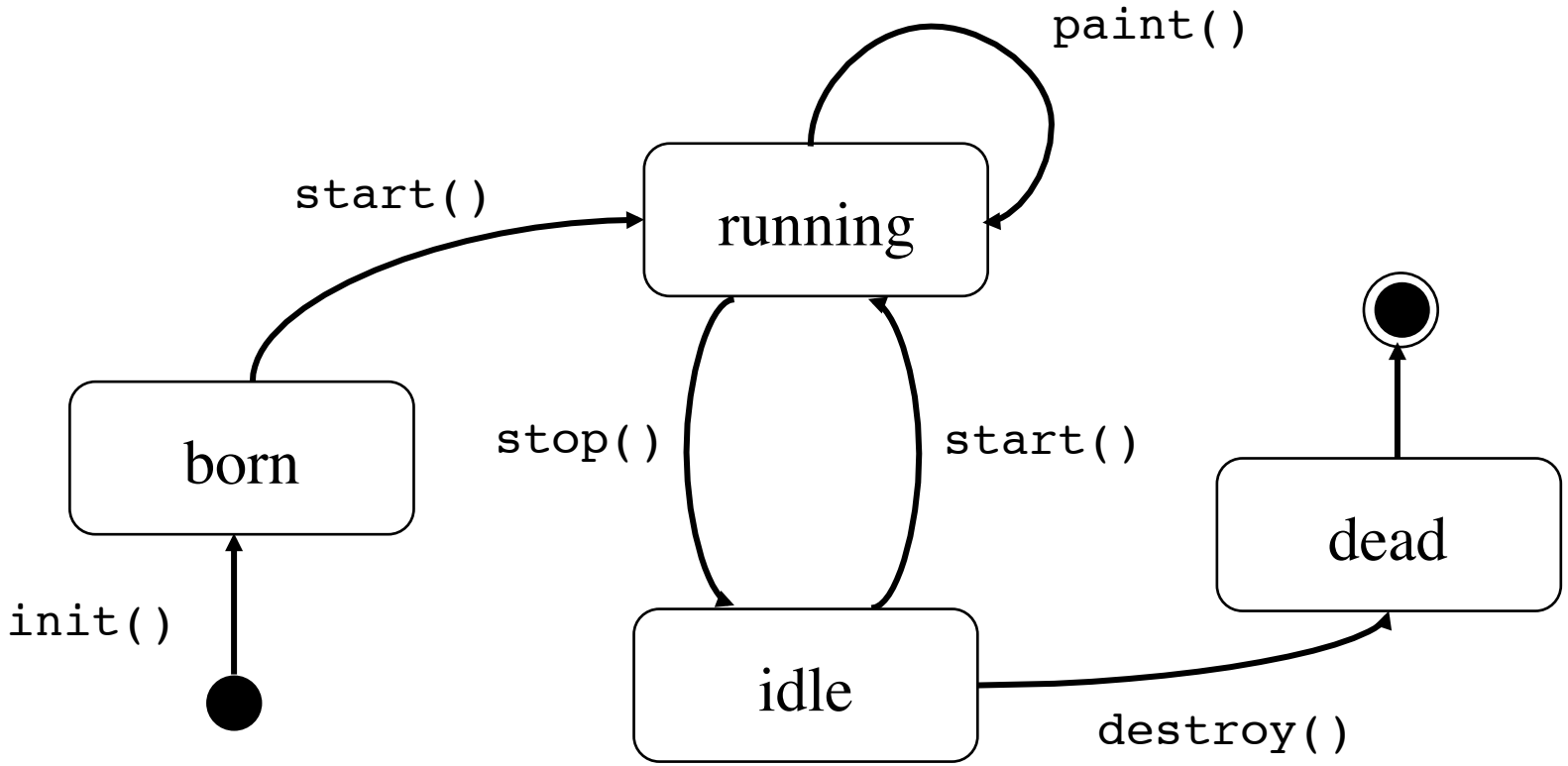
Klassen `java.applet.Applet` er et skelet af en applet.

En underklasse kan tilpasse en applet ved at overskrive en eller flere af følgende metoder:

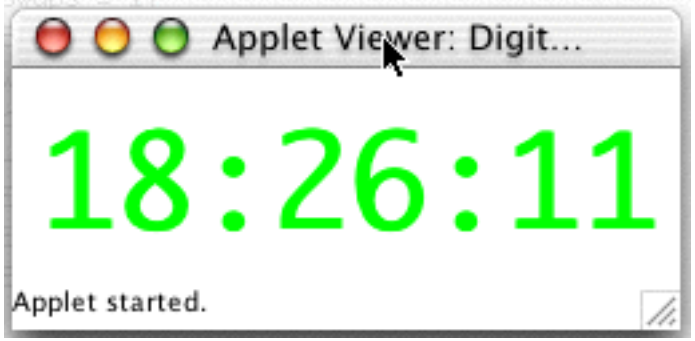
`init()`	kaldes, når appletten indlæses
`start()`	kaldes, når websiden besøges
`stop()`	kaldes, når websiden forlades
`destroy()`	kaldes, når websiden fjernes
`paint()`	kaldes, når websiden skal (gen)tegnes

En applets livscyklus

(tilstandsdiagram)



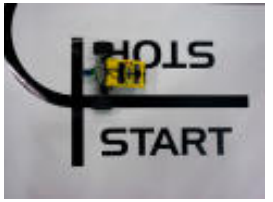
En applet til visning af tid



DigitalClock.java

```
import java.awt.*;
import java.util.Calendar;

public class DigitalClock
             extends java.applet.Applet implements Runnable {
    protected Thread clockThread = null;
    protected Font font = new Font("Monospaced", Font.BOLD, 48);
    protected Color color = Color.green;

    public void start() { ... }
    public void stop() { ... }
    public void run() { ... }

    public void paint(Graphics g) { ... }
}
```

Metoderne `start` og `stop`




```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this);
        clockThread.start();
    }
}

public void stop() {
    clockThread = null;
}
```

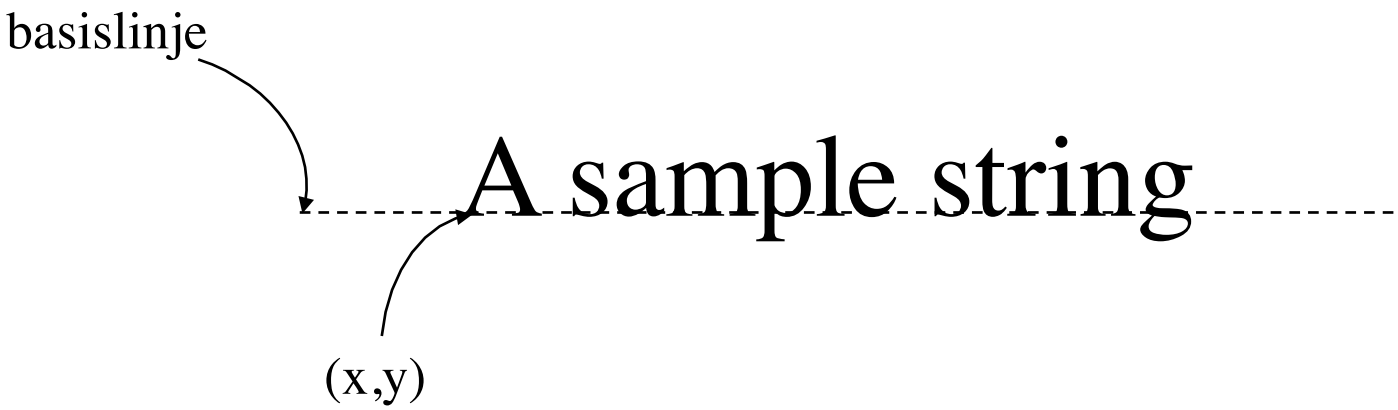
Metoden `run`

```
public void run() {
    while (clockThread != null) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}
```

Metoden `paint`

```
public void paint(Graphics g) {
    Calendar calendar = Calendar.getInstance();
    int hour = calendar.get(Calendar.HOUR_OF_DAY);
    int minute = calendar.get(Calendar.MINUTE);
    int second = calendar.get(Calendar.SECOND);
    g.setFont(font);
    g.setColor(color);
    g.drawString(hour + ":" + minute / 10 + minute % 10
                       + ":" + second / 10 + second % 10,
                 10, 60);
}
```

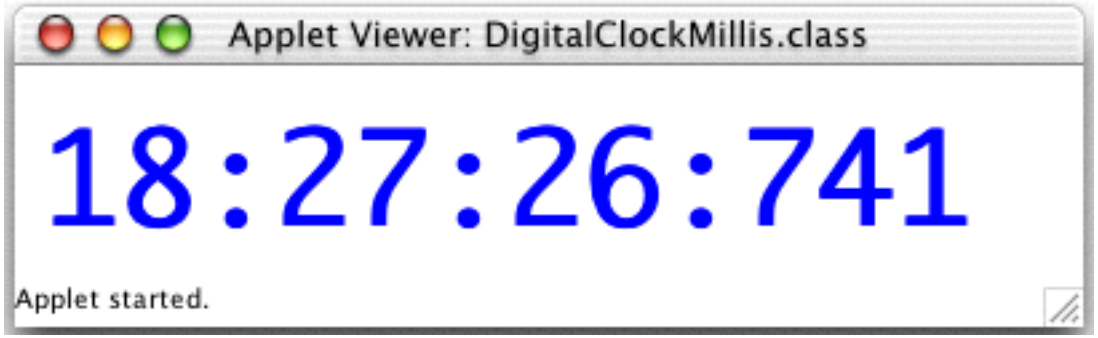
`drawString`



HTML-filen

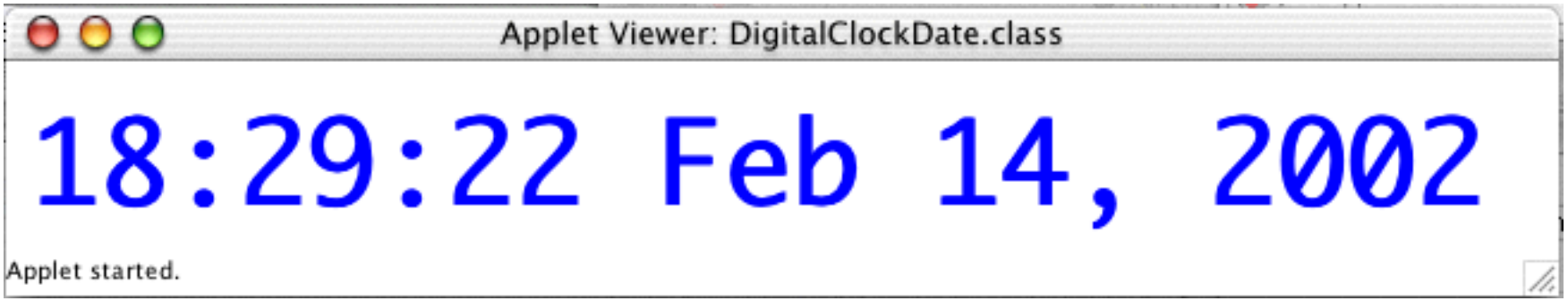
```
<html>
<body bgcolor=white>
  <center>
    <h1> The Digital Clock Applet</h1>
    <applet codebase="Java Classes"
            code=DigitalClock.class
            width=250 height=80>
    </applet>
  </center>
</body>
</html>
```

Visning af tidspunkt i millisekunder



```
public void run() {
    while (clockThread != null) {
        repaint();
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {}
    }
}

public void paint(Graphics g) {
    Calendar calendar = Calendar.getInstance();
    int hour = calendar.get(Calendar.HOUR_OF_DAY);
    int minute = calendar.get(Calendar.MINUTE);
    int second = calendar.get(Calendar.SECOND);
    int millisecond = calendar.get(Calendar.MILLISECOND);
    g.setFont(font);
    g.setColor(color);
    g.drawString(hour + ":" + minute / 10 + minute % 10 +
                         ":" + second / 10 + second % 10 +
                         ":" + millisecond / 100 +
                               (millisecond / 10) % 10 +
                               millisecond % 10, 10, 60);
}
```

Visning af tidspunkt og dato



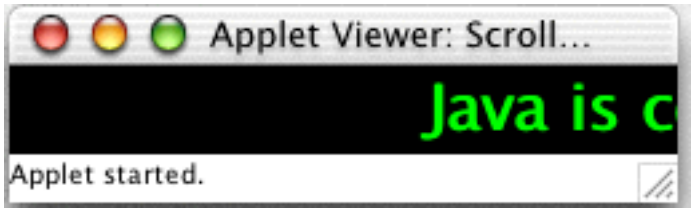
```
public void paint(Graphics g) {
    Calendar calendar = Calendar.getInstance();
    int hour = calendar.get(Calendar.HOUR_OF_DAY);
    int minute = calendar.get(Calendar.MINUTE);
    int second = calendar.get(Calendar.SECOND);
    Date date = new Date();
    DateFormat df = DateFormat.getDateInstance();
    g.setFont(font);
    g.setColor(color);
    g.drawString(hour + ":" + minute / 10 + minute % 10 +
                        ":" + second / 10 + second % 10 +
                        " " + df.format(date), 10, 60);
}
```

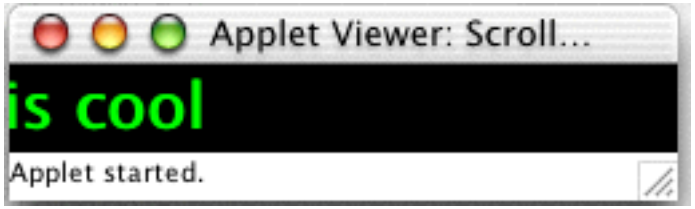
Parametre til appletter

```
<html>
<body bgcolor=white>
  <center>
    <h1>The <b>Extended</b> Digital Clock Applet</h1>
    <applet codebase="Java Classes"
            code=DigitalClock.class
            width=250 height=80>
            <param name=color value=blue>
    </applet>
  </center>
</body>
</html>
```

```
import java.awt.Color;

public class DigitalClock2 extends DigitalClock {
    public void init() {
        String param = getParameter("color");
        if ("red".param.equals(param))
            color = Color.red;
        else if ("blue".equals(param))
            color = Color.blue;
        else if ("yellow".equals(param))
            color = Color.yellow;
        else if ("orange".equals(param))
            color = Color.orange;
        else
            color = Color.green;
    }
}
```

En applet til animering af en lysavis





Klassen `ScrollingBanner`

```
import java.awt.*;

public class ScrollingBanner
             extends java.applet.Applet
             implements Runnable {
    protected Thread bannerThread;
    protected String text;
    protected Font font = new Font("Sans-serif", Font.BOLD, 24);
    protected int x, y;
    protected int delay = 100;
    protected int offset = 1;
    protected Dimension d;

    public void init() { ... }
    public void start() { ... }
    public void stop() { ... }
    public void paint(Graphics g) { ... }
    public void run() { ... }
}
```

HTML-filen

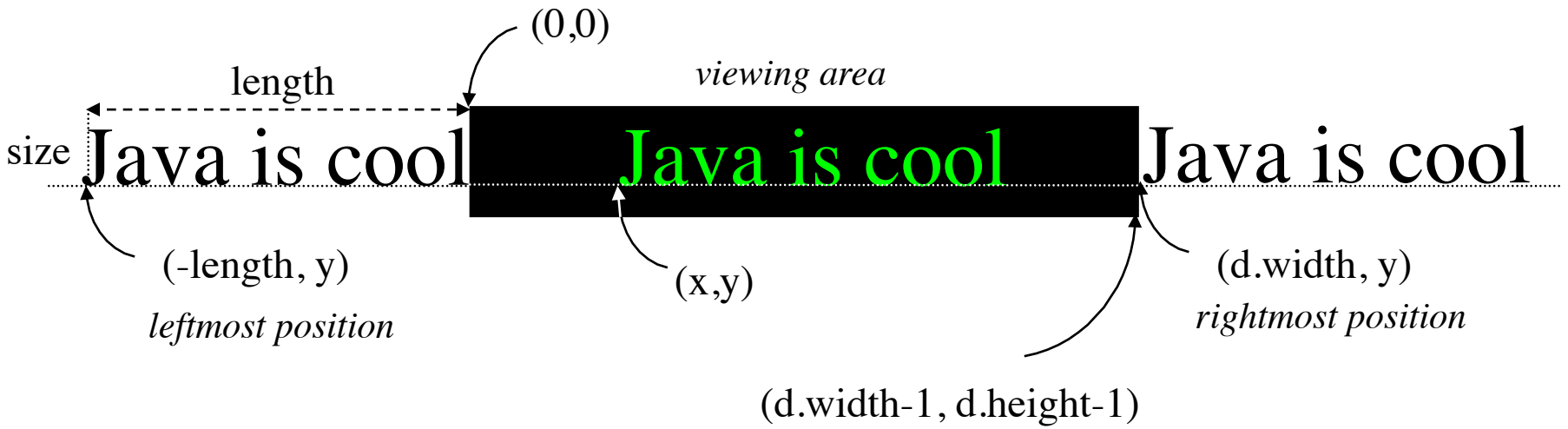
```
<html>
  <head>
     <title> Scrolling Banner Applet </title>
  </head>
<body bgcolor=black text=white>
  <center>
    <h1> The Scrolling Banner Applet</h1>
      <p>
        <applet
               codebase="Java Classes"
               code=ScrollingBanner.class
               width=250 height=33>
        <param name=text value="Java is cool">
        <param name=delay value=50>
        </applet>
  </center>
</body>
</html>
```

Metoden `init`

```
public void init() {
    String att = getParameter("text");
    text = att != null ? att : "Scrolling banner.";
    att = getParameter("delay");
    if (att != null)
        delay = Integer.parseInt(att);
    d = getSize();
    x = d.width;
    y = font.getSize();
}
```

Initialiseringen foretages i `init()` - ikke i en konstruktør

Animeringen



Metoden `paint`

```
public void paint(Graphics g) {
    g.setFont(font);
    FontMetrics fm = g.getFontMetrics();
    int length = fm.stringWidth(text);
    x -= offset;
    if (x < -length)
        x = d.width;
    g.setColor(Color.black);
    g.fillRect(0, 0, d.width, d.height);
    g.setColor(Color.green);
    g.drawString(text, x, y);
}
```

Metoderne `start` og `stop`

```
public void start() {
    if (bannerThread != null) {
        bannerThread = new Thread(this);
        bannerThread.start();
    }
}

public void stop() {
    bannerThread = null;
}
```

Metoden `run`

```
public void run() {
    while (bannerThread != null) {
        repaint();
        try {
            Thread.sleep(delay);
        } catch (InterruptedException e) {}
    }
}
```

Hvorledes undgås flimmer?

Flimmer skyldes kaldet af `repaint`.

`repaint` kalder `update`, der som standard

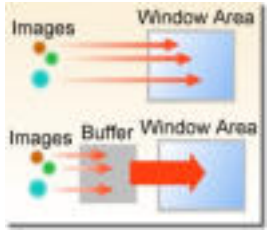
1. maler hele området med baggrundsfarven (typisk hvid),
2. sætter skrivefarven til forgrundsfarven (typisk sort), og
3. kalder `paint`

Løsning:

Overskriv `update`.

Benyt et `Image`-objekt som buffer (mellemlager).

Brug af dobbeltbuffer for at eliminere flimmer




```
import java.awt.*;

public class ScrollingBanner2 extends ScrollingBanner {
    protected Image image;
    protected Graphics offscreen;

    public void update(Graphics g) {     // called by repaint
        if (image == null) {
            image = createImage(d.width, d.height);
            offscreen = image.getGraphics();
        }
        super.paint(offscreen);          // paint in buffer
        g.drawImage(image, 0, 0, this);  // copy buffer to screen
    }

    public void paint(Graphics g) {
        update(g);
    }
}
```

Brug et Swing-JPanel for at eliminere flimmer

```
class ScrollingBannerAnimator
      extends JPanel implements Runnable {
    protected Thread bannerThread;
    protected String text;
    protected Font font =
        new java.awt.Font("Sans-serif", Font.BOLD, 24);
    protected int x, y;
    protected int delay;
    protected int offset = 1;
    protected Dimension d;

    ScrollingBannerAnimator(int delay, String text)
      { this.delay = delay; this.text = text;
        /* setDoubleBuffered(true); */ }
    void start() { ... }
    void stop() { ... }
    void run() { ... }
    void paintComponent(Graphics g) { ... }
}
```

fortsættes

```
public class ScrollingBanner extends JApplet {
    ScrollingBannerAnimator animator;
    int delay = 100;
    String text;

    public void init() {
        String att = getParameter("delay");
        if (att != null)
            delay = Integer.parseInt(att);
        att = getParameter("text");
        if (att != null)
            text = att  != null ? att : "Scrolling banner.";
        animator = new ScrollingBannerAnimator(delay, text);
        animator.setSize(getSize());
        getContentPane().add(animator);
    }

    public void start() { animator.start(); }
    public void stop() { animator.stop(); }
}
```

Klassen `java.awt.Graphics`

```
void setColor(Color c)
void setFont(Font f)
void setPaintMode()
void setXORMode(Color c)

Color getColor()
Font getFont()
FontMetrics getFontMetrics()
FontMetrics getFontMetrics(Font f)
```

fortsættes

```
void drawString(String s, int x, int y)
void drawLine(int x1, int y1, int x2, int y2)
void drawRectangle(int x, int y, int w, int h)
void fillRectangle(int x, int y, int w, int h)
void drawOval(int x, int y, int w, int h)
void fillOval(int x, int y, int w, int h)
void drawRoundRect(int x, int y, int w, int h)
void fillRoundRect(int x, int y, int w, int h)
void draw3DRoundRect(int x, int y, int w, int h,
                             boolean raised)
void fill3DRoundRect(int x, int y, int w, int h,
                             boolean raised)
void drawArc(int x, int y, int w, int h,
             int startAngle, int arcAngle)
void fillArc(int x, int y, int w, int h,
             int startAngle, int arcAngle)
void drawImage(Image img, int x, int y, ... )
```

Læsning af filer fra appletter

En applet kan ikke læse filer på klientmaskinen,
kun på servermaskinen.

```
import java.net.*;
...
try {
    URL url = new URL(getDocumentBase(), filename);
    BufferedReader in = new BufferedReader(
                            new InputStreamReader(
                                url.openStream()));
    String line;
    while ((line = in.readLine()) != null) {
        process line;
    }
} catch (IOException e) {}
```

Ugeseddel 3

14. september - 21. september

- Læs kapitel 7 i lærebogen (side 249 - 304)

- Løs opgave 5.1, **projekt** 5.4 (fortsættelse af opgave 5.1) og opgave 5.5.

  Vink til opgave 5.5: Beregningerne af koordinater kan foretages ud fra koden i metoden `polarToRectangular` på side 212 i lærebogen (se også figur 6.3 på side 211).