

Jacob Nordfalk

Objektorienteret programmering i

# Java

4. udgave

Objektorienteret  
programmering i

# Java

**Copyright:**

2007© by Forlaget Globe A/S  
4. udgave, 2. oplag

**Udgiver:**



Forlaget Globe A/S  
Skodsborgvej 305  
2850 Nærum  
Tlf. 70 15 14 00  
Fax 70 15 14 10  
E-mail: [info@globe.dk](mailto:info@globe.dk)  
Internet: <http://www.globe.dk>

**Forfatter:**

Jacob Nordfalk

**Forlagsredaktion:**

Carsten Straaberg  
([redaktion@globe.dk](mailto:redaktion@globe.dk))

**Omslag:**

Carsten Straaberg

**DtP:**

Jacob Nordfalk

**Sats:**

Forlaget Globe A/S, Nærum

**Tryk:**

Eurographic Danmark, Taastrup  
Printed in Denmark 2009

**ISBN: 978-87-7900-811-3**

Forlaget Globe og forfatteren kan ikke holdes  
økonomisk ansvarlig for eventuelle fejl og  
mangler i bogen.

**[www.globe.dk](http://www.globe.dk)**

Bøg om fritid, hobby, IT og erhverv

# Indholdsfortegnelse

<b>0 Forord.....</b>	<b>15</b>
<b>0.1 Bogens opbygning.....</b>	<b>15</b>
0.1.1 Veje gennem bogen.....	15
0.1.2 Kapitlernes opbygning.....	17
<b>0.2 Til underviseren.....</b>	<b>17</b>
<b>0.3 Ændringer i fjerde udgave.....</b>	<b>18</b>
<b>0.4 Tak.....</b>	<b>18</b>
<b>1 Introduktion.....</b>	<b>19</b>
<b>1.1 Programmering.....</b>	<b>20</b>
1.1.1 Styresystemet.....	20
1.1.2 Hvorfor lære at programmere? .....	20
1.1.3 Et simpelt program.....	20
1.1.4 Hvordan lærer man at programmere.....	21
<b>1.2 Fordele og ulemper ved Java.....</b>	<b>21</b>
1.2.1 Stærke sider.....	21
1.2.2 Stor opbakning.....	22
1.2.3 Svagheder.....	22
<b>1.3 Programmeringsværktøjer til Java.....</b>	<b>23</b>
1.3.1 Borland JBuilder.....	23
1.3.2 Eclipse.org.....	24
1.3.3 Oracle JDeveloper.....	25
1.3.4 NetBeans.....	26
1.3.5 BlueJ.....	27
1.3.6 Andre værktøjer.....	27
1.3.7 Sun JDK.....	27
<b>1.4 Hjælp og videre læsning.....</b>	<b>28</b>
1.4.1 Projekter med åben kildekode.....	28
<b>2 Basal programmering.....</b>	<b>29</b>
<b>2.1 Det første javaprogram.....</b>	<b>30</b>
2.1.1 Kommentarer.....	30
2.1.2 Klassedefinitionen.....	31
2.1.3 Oversættelse og kørsel af programmet.....	31

<b>2.2 Variabler.....</b>	<b>33</b>
2.2.1 Heltal.....	33
2.2.2 Sammensætte strenge med +.....	34
2.2.3 Beregningsudtryk.....	34
2.2.4 Kommatal.....	35
2.2.5 Matematiske funktioner.....	37
2.2.6 Kald af metoder.....	38
2.2.7 Logiske variabler.....	38
2.2.8 Opgaver.....	39
<b>2.3 Betinget udførelse.....</b>	<b>39</b>
2.3.1 Indlæsning fra tastaturet.....	40
2.3.2 if-else.....	41
2.3.3 Opgaver.....	41
<b>2.4 Blokke.....</b>	<b>42</b>
2.4.1 Indrykning.....	42
<b>2.5 Løkker.....</b>	<b>43</b>
2.5.1 while-løkken.....	43
2.5.2 for-løkken.....	45
2.5.3 Indlejrede løkker.....	46
2.5.4 Uendelige løkker.....	47
2.5.5 Opgaver.....	47
<b>2.6 Værditypekonvertering.....</b>	<b>48</b>
2.6.1 EksPLICIT typekonvertering.....	48
2.6.2 Implicit typekonvertering.....	49
2.6.3 Misforståelser omkring typekonvertering.....	49
<b>2.7 Fejl.....</b>	<b>50</b>
2.7.1 Indholdsmæssige (logiske) fejl.....	50
2.7.2 Sproglige fejl.....	50
2.7.3 Køretidsfejl.....	51
<b>2.8 Test dig selv.....</b>	<b>52</b>
<b>2.9 Resumé.....</b>	<b>53</b>
2.9.1 Gode råd om programmering.....	53
<b>2.10 Opgaver.....</b>	<b>54</b>
2.10.1 Befordringsfradrag.....	54
2.10.2 Kurveprogram.....	54
<b>2.11 Appendiks.....</b>	<b>55</b>
2.11.1 Navngivningsregler.....	55
2.11.2 De simple typer.....	55
2.11.3 Værditypekonvertering.....	55
2.11.4 Aritmetiske operatorer.....	56
2.11.5 Regning med logiske udtryk.....	57
2.11.6 Sammenligningsoperatorer.....	57

<b>2.12 Avanceret.....</b>	<b>58</b>
2.12.1 Grafiske indtastningsvinduer.....	58
2.12.2 Formater tal og datoer.....	59
2.12.3 Primal.....	60
2.12.4 Klassemetoder.....	60
2.12.5 Kombination af logiske operatoren.....	62
2.12.6 Alternativudtryk.....	63
2.12.7 Brug af ++ og -- (optælling og nedtælling).....	63
2.12.8 Brug af += og -=.....	63
2.12.9 do-while-løkken.....	63
2.12.10 break.....	64
2.12.11 continue.....	64
2.12.12 break og continue med navngivne løkker.....	64
2.12.13 switch.....	65
2.12.14 Komma i for-løkker.....	65
<b>3 Objekter.....</b>	<b>67</b>
<b>3.1 Objekter og klasser.....</b>	<b>68</b>
<b>3.2 Punkter (klassen Point).....</b>	<b>69</b>
3.2.1 Erklæring og oprettelse.....	69
3.2.2 Objektvariabler.....	70
3.2.3 Metodekald.....	70
3.2.4 Eksempel.....	71
3.2.5 Import af standardklasser.....	72
<b>3.3 Rektangler (klassen Rectangle).....</b>	<b>72</b>
3.3.1 Konstruktører.....	72
3.3.2 Metoder.....	73
3.3.3 Metoders returtype.....	74
3.3.4 Metoders parametre.....	75
<b>3.4 Tekststreng (klassen String).....</b>	<b>76</b>
3.4.1 Streng-objekter er uforanderlige.....	77
3.4.2 Man behøver ikke bruge new til String-objekter.....	78
3.4.3 Navnesammenfald for metoder.....	78
3.4.4 At sætte streng sammen med +.....	78
3.4.5 Sammenligning.....	79
3.4.6 Opgaver.....	80
<b>3.5 Lister (klassen ArrayList).....</b>	<b>80</b>
3.5.1 Gennemløb af lister.....	82
3.5.2 Eksempel: Eventyrfortælling.....	82
3.5.3 Eksempel: Punkter i en liste.....	82
3.5.4 Historisk bemærkning: Lister før JDK 1.5.....	83
3.5.5 Eksempel: Blanding af kort med ArrayList.....	84
<b>3.6 Datoer (klassen Date).....</b>	<b>85</b>
3.6.1 Opgaver.....	86

<b>3.7 Test dig selv.....</b>	<b>87</b>
<b>3.8 Resumé.....</b>	<b>87</b>
<b>3.9 Appendiks.....</b>	<b>88</b>
3.9.1 Klassen Point.....	88
3.9.2 Klassen Rectangle.....	89
3.9.3 Klassen String.....	90
3.9.4 Specialtegn i strenge.....	91
3.9.5 Klassen Date.....	92
3.9.6 Klassen ArrayList.....	93
<b>3.10 Avanceret.....</b>	<b>94</b>
3.10.1 Sætte strenge sammen (klassen StringBuilder).....	94
3.10.2 Standardmetoder til at arbejde med lister.....	96
3.10.3 Lister af simple typer (autoboxing).....	96
3.10.4 Andre slags lister og mængder.....	96
3.10.5 Nøgleindekserede tabeller (klassen HashMap).....	97
<b>4 Definition af klasser.....</b>	<b>99</b>
<b>4.1 En Boks-klasse.....</b>	<b>100</b>
4.1.1 Variabler.....	100
4.1.2 Brug af klassen.....	101
4.1.3 Metodefinition.....	101
4.1.4 Flere objekter.....	102
<b>4.2 Indkapsling.....</b>	<b>103</b>
<b>4.3 Konstruktører.....</b>	<b>105</b>
4.3.1 Standardkonstruktører.....	106
4.3.2 Opgaver.....	106
<b>4.4 En Terning-klasse.....</b>	<b>107</b>
4.4.1 Opgaver.....	108
<b>4.5 Relationer mellem objekter.....</b>	<b>108</b>
4.5.1 En Raflebæger-klasse.....	108
4.5.2 Opgaver.....	110
<b>4.6 Nøgleordet this.....</b>	<b>111</b>
<b>4.7 Ekstra eksempler.....</b>	<b>112</b>
4.7.1 En n-sidet terning.....	112
4.7.2 Personer.....	114
4.7.3 Design af klasser.....	115
<b>4.8 Test dig selv.....</b>	<b>116</b>
<b>4.9 Resumé.....</b>	<b>116</b>
4.9.1 Formen af en klasse.....	117
4.9.2 Formen af en metode.....	118

<b>4.10 Opgaver.....</b>	<b>119</b>
4.10.1 BlueJ.....	119
4.10.2 Fejlfinding.....	120
<b>4.11 Avanceret.....</b>	<b>122</b>
4.11.1 Bankkonti.....	122
4.11.2 Opgaver.....	122
<b>5 Nedarvning.....</b>	<b>123</b>
<b>5.1 At udbygge eksisterende klasser.....</b>	<b>124</b>
5.1.1 Eksempel: En falsk terning.....	124
5.1.2 At udbygge med flere metoder og variable.....	126
5.1.3 Nøgleordet super.....	126
5.1.4 Opgaver.....	127
<b>5.2 Polymorfe variable.....</b>	<b>128</b>
5.2.1 Dispensation fra traditionel typesikkerhed.....	128
5.2.2 Polymorfi.....	129
5.2.3 Eksempel på polymorfi: Brug af Raflebaeger.....	129
5.2.4 Hvilken vej er en variabel polymorf ?.....	130
5.2.5 Reference-typekonvertering.....	131
<b>5.3 Eksempel: Et matador-spil.....</b>	<b>132</b>
5.3.1 Polymorfi.....	138
5.3.2 Opgaver.....	138
<b>5.4 Stamklassen Object.....</b>	<b>139</b>
5.4.1 Referencer til objekter.....	140
<b>5.5 Konstruktører i underklasser.....</b>	<b>141</b>
5.5.1 Konsekvenser.....	142
<b>5.6 Ekstra eksempler.....</b>	<b>143</b>
5.6.1 Matadorspillet version 2.....	143
5.6.2 Opgaver.....	145
<b>5.7 Test dig selv.....</b>	<b>146</b>
<b>5.8 Resumé.....</b>	<b>146</b>
<b>5.9 Avanceret.....</b>	<b>147</b>
5.9.1 Initialisering uden for konstruktørerne.....	147
5.9.2 Kald af en konstruktør fra en anden konstruktør.....	148
5.9.3 Metoder erklæret final.....	148
5.9.4 Metoder erklæret protected.....	148
5.9.5 Variabel-overskygning.....	148
<b>6 Pakker.....</b>	<b>149</b>
<b>6.1 At importere klassedefinitioner.....</b>	<b>150</b>
<b>6.2 Standardpakkerne.....</b>	<b>150</b>
6.2.1 Pakken java.lang.....	151

<b>6.3 Pakkers placering på filsystemet.....</b>	<b>151</b>
<b>6.4 At definere egne pakker.....</b>	<b>152</b>
6.4.1 Eksempel.....	152
<b>6.5 Pakke klasser i jar-filer (Java-arkiver).....</b>	<b>152</b>
6.5.1 Eksekverbare jar-filer.....	153
<b>6.6 Test dig selv.....</b>	<b>153</b>
<b>6.7 Resumé.....</b>	<b>153</b>
<b>6.8 Opgaver.....</b>	<b>153</b>
<b>6.9 Avanceret: public, protected og private.....</b>	<b>154</b>
6.9.1 Variabler og metoder.....	154
6.9.2 Indkapsling med pakker.....	154
6.9.3 Klasser.....	154
<b>7 Lokale, objekt- og klassevariabler.....</b>	<b>155</b>
<b>7.1 Klassevariabler og -metoder.....</b>	<b>156</b>
7.1.1 Eksempler på klassevariabler.....	157
7.1.2 Eksempler på klassemetoder.....	157
<b>7.2 Lokale variabler og parametre.....</b>	<b>158</b>
7.2.1 Parametervariabler.....	158
<b>7.3 Test dig selv.....</b>	<b>160</b>
<b>7.4 Resumé.....</b>	<b>160</b>
7.4.1 Tilknytning.....	160
7.4.2 Adgang til variabler og metoder.....	160
7.4.3 Adgang fra metoder.....	160
<b>7.5 Avanceret.....</b>	<b>161</b>
7.5.1 Introduktion til rekursive algoritmer.....	161
7.5.2 Rekursion: Beregning af formel.....	162
7.5.3 Rekursion: Listning af filer.....	163
7.5.4 Rekursion: Tegning af fraktaler.....	164
<b>8 Arrays.....</b>	<b>165</b>
<b>8.1 Erklæring og brug.....</b>	<b>166</b>
8.1.1 Eksempel: Statistik.....	167
8.1.2 Initialisere et array med startværdier.....	167
8.1.3 Arrayet i main()-metoden.....	168
<b>8.2 Gennemløb og manipulering af array.....</b>	<b>168</b>
<b>8.3 Array af objekter.....</b>	<b>169</b>
8.3.1 Polymorfi.....	169
<b>8.4 Arrays versus lister (ArrayList).....</b>	<b>169</b>
<b>8.5 Resumé.....</b>	<b>170</b>
<b>8.6 Opgaver.....</b>	<b>170</b>



<b>8.7 Avanceret.....</b>	<b>171</b>
8.7.1 Flerdimensionale arrays.....	171
8.7.2 Konvertere mellem arrays og lister.....	172
<b>9 Grafiske programmer.....</b>	<b>173</b>
<b>9.1 Klassen Graphics.....</b>	<b>175</b>
9.1.1 Eksempel: Grafikdemo.....	176
9.1.2 Eksempel: Kurvetegning.....	177
<b>9.2 Metoder du kan kalde.....</b>	<b>178</b>
<b>9.3 Opgaver.....</b>	<b>179</b>
9.3.1 Opgave: Grafisk Matador-spil.....	179
<b>9.4 Avanceret.....</b>	<b>181</b>
9.4.1 Simple animationer.....	181
9.4.2 Animationer med en separat tråd.....	182
9.4.3 Java2D - avanceret grafiktegning.....	182
9.4.4 Passiv versus aktiv visning.....	184
9.4.5 Fuldskærmstegning.....	185
9.4.6 Dobbeltbuffer.....	186
<b>10 Appletter.....</b>	<b>187</b>
<b>10.1 HTML-dokumentet.....</b>	<b>188</b>
<b>10.2 Javakoden.....</b>	<b>188</b>
<b>10.3 Metoder i appletter.....</b>	<b>189</b>
10.3.1 Metoder i appletter, som du kan kalde.....	189
10.3.2 Metoder i appletter, som systemet kalder.....	190
10.3.3 Eksempel.....	190
<b>10.4 Resumé.....</b>	<b>191</b>
<b>10.5 Avanceret.....</b>	<b>192</b>
10.5.1 Sikkerhedsbegrænsninger for appletter.....	192
10.5.2 Pakker og CODE/CODEBASE i HTML-koden.....	192
10.5.3 Begrænsninger i ældre netlæsere.....	192
<b>11 Grafiske standardkomponenter.....</b>	<b>193</b>
<b>11.1 Generering med et værktøj.....</b>	<b>194</b>
11.1.1 Interaktive programmer.....	196
11.1.2 Genvejstaster.....	197
<b>11.2 Overblik over komponenter.....</b>	<b>198</b>
11.2.1 JLabel.....	198
11.2.2 JButton.....	198
11.2.3 JCheckBox og JRadioButton.....	198
11.2.4 JTextField.....	199
11.2.5 JTextArea.....	199
11.2.6 JComboBox.....	199
11.2.7 JList.....	200

<b>11.3 Eksempel.....</b>	<b>200</b>
<b>11.4 Overblik over containere.....</b>	<b>202</b>
11.4.1 JWindow.....	202
11.4.2 JDialog.....	202
11.4.3 JFrame.....	202
11.4.4 JPanel.....	202
11.4.5 JApplet.....	202
11.4.6 JTabbedPane.....	203
<b>11.5 Layout-managere.....</b>	<b>203</b>
11.5.1 Ingen styring (null-layout).....	203
11.5.2 FlowLayout.....	203
11.5.3 BorderLayout.....	204
11.5.4 GridBagLayout.....	205
<b>11.6 Menuer.....</b>	<b>206</b>
<b>11.7 Test dig selv.....</b>	<b>208</b>
<b>11.8 Resumé.....</b>	<b>208</b>
<b>11.9 Avanceret.....</b>	<b>209</b>
11.9.1 HTML-kode i komponenter.....	209
11.9.2 Flertrådet komponentprogrammering.....	210
11.9.3 Brug af komponenter vs. paintComponent().....	210
<b>12 Interfaces - grænseflader til objekter.....</b>	<b>211</b>
<b>12.1 Definere et interface.....</b>	<b>212</b>
<b>12.2 Implementere et interface.....</b>	<b>212</b>
12.2.1 Variabler af type Tegnbar.....	213
12.2.2 Eksempler med interfacet Tegnbar.....	213
12.2.3 Visning af nogle Tegnare objekter.....	215
12.2.4 Polymorfi.....	216
<b>12.3 Interfaces i standardbibliotekerne.....</b>	<b>216</b>
12.3.1 Sortering med en Comparator.....	216
12.3.2 Flere tråde med Runnable.....	217
12.3.3 Lytte til musen med MouseListener.....	217
<b>12.4 Test dig selv.....</b>	<b>218</b>
<b>12.5 Resumé.....</b>	<b>218</b>
<b>12.6 Opgaver.....</b>	<b>218</b>
<b>12.7 Avanceret.....</b>	<b>219</b>
12.7.1 Collections - ArrayList's familie.....	219
12.7.2 Implementere flere interfaces.....	219
12.7.3 At udbygge interfaces.....	220
12.7.4 Multipel arv.....	220
12.7.5 Variabler i et interface.....	220
12.7.6 Statisk import.....	220

<b>13 Hændelser i grafiske brugergrænseflader.....</b>	<b>221</b>
13.1 Eksempel: LytTilMusen.....	222
13.2 Eksempel: Linjetegning.....	223
13.2.1 Linjetegning i én klasse.....	225
13.3 Ekstra eksempler.....	225
13.3.1 Lytte til musebevægelser.....	225
13.3.2 Lytte til en knap.....	226
13.3.3 Lytte efter tastetryk.....	227
13.4 Appendiks.....	228
13.4.1 Lyttere og deres metoder.....	228
13.5 Avanceret.....	230
13.5.1 Automatisk genereret kode til hændelseshåndtering.....	230
13.5.2 Adapte.....	230
<b>14 Undtagelser og køretidsfejl.....</b>	<b>231</b>
14.1 Almindelige undtagelser.....	233
14.2 At fange og håndtere undtagelser.....	233
14.2.1 Undtagelsesobjekter og deres stakspor.....	234
14.3 Undtagelser med tvungen håndtering.....	235
14.3.1 Fange undtagelser eller sende dem videre.....	235
14.3.2 Konsekvenser af at sende undtagelser videre.....	236
14.4 Præcis håndtering af undtagelser.....	238
14.5 Fange flere slags undtagelser.....	240
14.6 Resumé.....	240
14.7 Opgaver.....	240
14.8 Avanceret.....	241
14.8.1 Fange Throwable.....	241
14.8.2 Selv kaste undtagelser (throw).....	241
14.8.3 try - finally.....	242
14.8.4 Selv definere undtagelsestyper.....	242
<b>15 Datastrømme og filhåndtering.....</b>	<b>243</b>
15.1 Skrive til en tekstfil.....	244
15.2 Læse fra en tekstfil.....	245
15.3 Analysering af tekstdata.....	246
15.3.1 Opdele strenge og konvertere bidderne til tal.....	246
15.3.2 Indlæsning af tekst med Scanner-klassen.....	246
15.4 Binær læsning og skrivning.....	248
15.4.1 Optimering af ydelse.....	248

<b>15.5 Appendiks.....</b>	<b>249</b>
15.5.1 Navngivning.....	249
15.5.2 Binære data ( -OutputStream og -InputStream).....	249
15.5.3 Tekstdata ( -Writer og -Reader).....	250
15.5.4 Fillæsning og -skrivning (File- ).....	250
15.5.5 Strenger (String- ).....	250
15.5.6 Arrays (ByteArray- og CharArray- ).....	251
15.5.7 Læse og skriv objekter (Object- ).....	251
15.5.8 Dataopsamling (Buffered- ).....	251
15.5.9 Gå fra binære til tegnbaserede datastrømme.....	251
15.5.10 Filtreringsklasser til konvertering og databehandling.....	251
15.5.11 Brug af på filtreringsklasser.....	252
<b>15.6 Test dig selv.....</b>	<b>253</b>
<b>15.7 Resumé.....</b>	<b>253</b>
<b>15.8 Opgaver.....</b>	<b>253</b>
<b>15.9 Avanceret.....</b>	<b>254</b>
15.9.1 Klassen RandomAccessFile.....	254
15.9.2 Filhåndtering (klassen File).....	254
15.9.3 Platformuafhængige filnavne.....	254
<b>16 Netværkskommunikation.....</b>	<b>255</b>
16.1 At forbinde til en port.....	256
16.2 At lytte på en port.....	258
16.3 URL-klassen.....	259
16.4 Opgaver.....	260
16.5 Avanceret.....	261
16.5.1 FTP-kommunikation.....	261
16.5.2 Brug af FTP fra en applet.....	263
<b>17 Flertrådet programmering.....</b>	<b>265</b>
17.1 Princip.....	266
17.1.1 Eksempel.....	266
17.2 Ekstra eksempler.....	268
17.2.1 En flertrådet webserver.....	268
17.2.2 Et flertrådet program med hoppende bolde.....	269
17.3 Opgaver.....	270

<b>17.4 Avanceret.....</b>	<b>271</b>
17.4.1 Nedrivning fra Thread.....	271
17.4.2 Synkronisering.....	271
17.4.3 Synkronisering på objekter og semaforer.....	272
17.4.4 wait() og notify().....	272
17.4.5 Prioritet.....	272
17.4.6 Opgaver.....	272
<b>18 Serialisering af objekter.....</b>	<b>273</b>
<b>18.1 Hente og gemme objekter.....</b>	<b>274</b>
<b>18.2 Serialisering af egne klasser.....</b>	<b>275</b>
18.2.1 Interfacet Serializable.....	275
18.2.2 Nøgleordet transient.....	275
18.2.3 Versionsnummeret på klassen.....	275
18.2.4 Eksempel.....	276
<b>18.3 Opgaver.....</b>	<b>276</b>
<b>18.4 Avanceret.....</b>	<b>277</b>
18.4.1 Serialisere det samme objekt flere gange.....	277
18.4.2 Selv styre serialiseringen af en klasse.....	278
<b>19 RMI - objekter over netværk.....</b>	<b>279</b>
<b>19.1 Principper.....</b>	<b>280</b>
<b>19.2 I praksis.....</b>	<b>280</b>
19.2.1 På serversiden.....	281
19.2.2 På klientsiden.....	282
<b>19.3 Opgaver.....</b>	<b>282</b>
19.3.1 Server og klient to forskellige steder.....	282
19.3.2 Starte separat 'rmiregistry'.....	282
<b>20 JDBC - databaseadgang.....</b>	<b>283</b>
<b>20.1 Kontakt til databasen.....</b>	<b>284</b>
20.1.1 JDBC-ODBC-broen under Windows.....	284
<b>20.2 Kommunikere med databasen.....</b>	<b>284</b>
20.2.1 Kommandoer.....	284
20.2.2 Forespørgsler.....	285
<b>20.3 Adskille database- og programlogik.....</b>	<b>285</b>
<b>20.4 Opgaver.....</b>	<b>287</b>
<b>20.5 Avanceret.....</b>	<b>288</b>
20.5.1 Forpligtende eller ej? (commit).....	288
20.5.2 Optimering.....	288
20.5.3 Metadata.....	290
20.5.4 Opdatering og navigering i ResultSet-objekter.....	290

<b>21 Avancerede klasser.....</b>	<b>291</b>
<b>21.1 Nøgleordet abstract.....</b>	<b>292</b>
21.1.1 Abstrakte klasser.....	292
21.1.2 Abstrakte metoder.....	292
<b>21.2 Nøgleordet final.....</b>	<b>293</b>
21.2.1 Variabler erklæret final.....	293
21.2.2 Metoder erklæret final.....	294
21.2.3 Klasser erklæret final.....	294
<b>21.3 Indre klasser.....</b>	<b>295</b>
<b>21.4 Almindelige indre klasser.....</b>	<b>295</b>
21.4.1 Eksempel - Linjetegning.....	296
<b>21.5 Lokale indre klasser.....</b>	<b>297</b>
<b>21.6 Anonyme indre klasser.....</b>	<b>298</b>
21.6.1 Eksempel - filtrering af filnavne.....	298
21.6.2 Eksempel - Linjetegning.....	299
21.6.3 Eksempel - tråde.....	300
<b>21.7 Resumé.....</b>	<b>301</b>
<b>21.8 Opgaver.....</b>	<b>301</b>
<b>21.9 Avanceret.....</b>	<b>302</b>
21.9.1 public, protected og private på en indre klasse.....	302
21.9.2 static på en indre klasse.....	302
<b>22 Objektorienteret analyse og design.....</b>	<b>303</b>
<b>22.1 Krav til programmet.....</b>	<b>304</b>
<b>22.2 Objektorienteret analyse.....</b>	<b>305</b>
22.2.1 Skrive vigtige ord op.....	305
22.2.2 Brugssituationer - Hvem Hvad Hvor.....	305
22.2.3 Aktivitetsdiagrammer.....	307
22.2.4 Skærbilleder.....	307
<b>22.3 Objektorienteret design.....</b>	<b>308</b>
22.3.1 Design af klasser.....	308
22.3.2 Kollaborationsdiagrammer.....	309
22.3.3 Klassesdiagrammer.....	310
<b>Engelsk-dansk ordliste.....</b>	<b>311</b>
<b>Stikordsregister.....</b>	<b>315</b>

# 0 Forord

I denne bog kan du lære tre ting:

- **Programmering.** Bogen starter fra grunden af, men har du ikke programmeret før, bør du være rede til at gøre en indsats, dvs. løbende lave en række små programmer, for at øve dig i stoffet. Det forudsættes, at du har kendskab til computere på brugerniveau, dvs. at du uden problemer kan bruge internet, tekstbehandling og andre programmer og problemfrit navigere og flytte og kopiere filer i Windows, Linux eller et andet styresystem. En smule kendskab til matematik er en fordel, men bestemt ikke et krav.
- **Objektorienteret programmering.** Bogen arbejder grundigt med begreberne omkring objektorienteret programmering (forkortet OOP) og har mange praktiske eksempler på denne måde at programmere på. Den introducerer og anvender løbende relevante dele af UML-notationen, som er meget anvendt i objektorienteret analyse og design og beslægtede fag.
- **Java.** Programmeringssproget Java har en række faciliteter, som kan lette programmeringen meget. Har du – som mange andre – lært Java ved at prøve dig frem, kan det overblik, der præsenteres i denne bog, hjælpe, samtidig med, at hvert kapitel slutter af med at gå i dybden med avancerede emner.

Ideen til bogen opstod i efteråret 2000, i forbindelse med at jeg underviste på et kursus i objektorienteret programmering og Java på IT-Diplomuddannelsen på Ingeniørhøjskolen i København. Jeg savnede en lærebog på dansk og gik sammen med Troels Nordfalk og Henrik Tange i gang med at skrive mine egne noter. Det er blevet en praktisk orienteret lærebog, krydret med mindre, men komplette, eksempler.

Størstedelen af bogen (opgaver og avancerede emner er skåret væk) findes også på adressen <http://javabog.dk>, der frit kan bruges af alle, der vil lære Java. Bidrag til bogen og hjemmesiden er meget velkomne: Skriv til [jacob.nordfalk@gmail.com](mailto:jacob.nordfalk@gmail.com)

På <http://javabog.dk> kan du også hente programeksemplerne fra bogen.

## 0.1 Bogens opbygning

Hvert emne behandles i et særskilt kapitel. Det er i en vis udstrækning muligt, at læse kapitlerne uafhængigt af hinanden.

### 0.1.1 Veje gennem bogen

Det anbefales at læse kapitel 2, 3, 4 og 5 i rækkefølge. Derefter kan man vælge, om man vil:

- læse nogle af de valgfrie kapitler (6, 7 og 8)
- gå i gang med interfaces, grafik og brugergrænseflader (9, evt. 10, 11, 12, evt. 13)
- arbejde med undtagelser (eng.: exceptions), filer og netværk (14, 15 og 16)

Her er en oversigt over kapitlerne (midtersøjle) og hvilke kapitler de forudsætter (venstre søjle). Hvis et kapitel er forudsat i parentes, betyder det, at visse eksempler eller opgaver forudsætter det, men man kan godt klare sig uden.

<b>Forudsætter</b>	<b>Kapitelnummer og -navn</b>	<b>Er nødvendig for</b>
	<b>2 Basal programmering</b>	Alle efterfølgende
2	<b>3 Objekter</b>	Alle efterfølgende
3	<b>4 Definition af klasser</b>	Næsten alle efterfølgende
4	<b>5 Nedrivning</b>	Næsten alle efterfølgende
<b>Valgfrie emner</b>		
4	<b>6 Pakker</b>	Ingen
4	<b>7 Lokale, objekt- og klassevariabler</b>	Ingen
3 (5)	<b>8 Arrays</b>	Ingen
<b>Grafik og brugergrænseflader (klient-programmering)</b>		
3 (5)	<b>9 Grafiske programmer</b>	11
9	<b>10 Appletter</b>	Ingen
9 (5)	<b>11 Grafiske standardkomponenter</b>	13
5	<b>12 Interfaces - grænseflade til objekter</b>	13, 17, 18, 19, 21
11, 12	<b>13 Hændelser i grafiske brugergrænseflader</b>	21
<b>Filer og netværk (server-programmering)</b>		
4 (5)	<b>14 Undtagelser og køretidsfejl</b>	15
14	<b>15 Datastrømme og filhåndtering</b>	16 + 18
15	<b>16 Netværkskommunikation</b>	
<b>Videregående emner</b>		
12 (16)	<b>17 Flertrådet programmering</b>	Ingen
15 (12)	<b>18 Serialisering af objekter</b>	19
12, 18	<b>19 RMI - objekter over netværk</b>	Ingen
14	<b>20 JDBC - databaseadgang</b>	Ingen
5, 12 (13, 17)	<b>21 Avancerede klasser</b>	Ingen
5	<b>22 Objektorienteret analyse og design</b>	Ingen



## 0.1.2 Kapitlernes opbygning

Hvert kapitel starter med en **oversigt** over indholdet og hvilke andre kapitler det forudsætter, sådan at man altid har overblik over, om man har de nødvendige forudsætninger.

Så kommer **hovedteksten**, der introducerer emnerne og kommer med programeksempler, hvor de anvendes.

De fleste kapitler har herefter

- **Test dig selv**, hvor man kan afprøve, om man har fået fat i de vigtigste ting i kapitlet.
- **Resumé**, som i kort form repeterer de vigtigste ting.
- **Appendiks** giver en komplet oversigt over de fakta, som relaterer sig til kapitlet. Det er beregnet til at blive læst sammen med kapitlet, men er også velegnet til senere opslag.
- **Avanceret**, der handler om de mere avancerede ting i relation til kapitlets emne. Avancerede afsnit kan springes over ved første læsning, men kan være nyttige at kigge på senere hen. Resten af bogen forudsætter ikke, at man har læst de avancerede afsnit.

## 0.2 Til underviseren

Denne bog koncentrerer sig om OOP og Java i praksis. Den starter med det helt grundlæggende og kommer i avanceret-afsnittene til bunds i mange aspekter af stoffet. I OOP går den "hele vejen" og får dækket klasser og objekter, indkapsling, arv og polymorfi.

Der er mange måder at undervise i Java på og bogen giver mulighed for, at underviseren selv vælger, hvilken del af stoffet, han vil lægge vægt på i sit kursus:

- Objektorienteret programmering: kapitel 2, 3, 4, 5 og derefter 9, 11, 12, 13 (kapitel 7 om klassevariabler kan komme sent i kurset, så eleverne ikke fristes til at bruge dem som "globale variabler").
- Strukturel programmering: kapitel 2, dernæst dets avanceret-afsnit om do-while, switch, tastaturindlæsning og klassemetoder, kapitel 3 om brug af objekter, kapitel 8 om arrays, kapitel 4 om klasser, kapitel 7 om klassevariabler.
- Sproget Java: kapitel 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 osv. i rækkefølge.
- Hurtigt i gang med at lave grafiske programmer: kapitel 2, det meste af 3 og derefter 9, 11 og 13 om grafik, grafiske standardkomponenter og hændelser.

Bogen er egnet til kursusbrug. Der findes specielle projekter til udviklingsværktøjet BlueJ, der mange steder bruges til at støtte eleven i at komme i gang med objektorienteret programmering (se afsnit 1.3.5 og afsnit 4.10.1).

Der findes en (gratis) samlet pakke, hvor underviseren får:

- transparenter (præsentationer), der supplerer bogen
- forslag til lektionsplan og opgaver
- vejledende opgavebesvarelser

Skriv til [jacob.nordfalk@gmail.com](mailto:jacob.nordfalk@gmail.com), hvis du holder et kursus og er interesseret i undervisningspakken.

## 0.3 Ændringer i fjerde udgave

I de sidste år har jeg samlet på rettelser, kommentarer og forslag til bogen og derudover selv undervist i den. Rettelserne og forbedringerne er næsten alle ført ind i fjerde udgave.

Selvom der er kommet meget stof til, er sidetallet ikke øget. I stedet er de eksisterende sider blevet bedre udnyttet.

Fjerde udgave er blevet opdateret og forbedret på en række punkter:

- For klarheds skyld har variabelnavne, der optræder i brødteksten, fået ændret typografi til fastbreddeskrift (klassenavne, metoder etc. har almindelig skrift for ikke at forvirre øjet unødigt).
- Kapitel 1, Introduktion er opdateret mht. udviklingsværktøjer, og der er kommet vejledninger til hvert værktøj i, så man nemt kan komme i gang med bogens eksempler.
- Kapitel 5, Nedarvning har fået matadorspillets objekter inddelt efter ansvarområder.
- Kapitel 9, Grafiske programmer er baseret på Swing og har fået tilføjet afsnit om aktiv visning og om fuldskærmstegning.
- Kapitel 11, Grafiske standardkomponenter bruger nu udelukkende Swing-komponenter og der står mere om menuer, genvejstaster, faneblade, dialoger, tabeller, HTML-kode i komponenter, flertrådet komponentprogrammering og brug af komponenter vs. `paintComponent()`.
- Kapitel 12, Interfaces har fået `TerminingComparator` gjort generisk.
- Kapitel 13, Hændelser er ændret fra at bruge appletter til at bruge paneler og faneblade.
- Kapitel 18, Serialisering berører nu `serialVersionUID` og XML-serialisering.
- Kapitel 19, RMI opdateret og forenklet.
- Kapitel 22, Objektorienteret analyse og design er opdateret med brugssituationer (og kravslister og ansvarsområder) som arbejdsmetoder.

## 0.4 Tak

Min bror Troels' kritik og bidrag har været uundværlige og bogen afspejler mange af hans holdninger (som også er blevet mine) om objektorienteret programmering. Tak til Henrik Tange, Jakob Bendsen, Peter Sestoft og andre, som har ladet mig bruge deres materiale. Tak til Torben Vaarby Laursen for at læse faglig korrektur på manuskriptet til 1. udgave. Og tak til Linux-samfundet for at lave et styresystem, der styrer!

Denne bog er skrevet med OpenOffice.org under Mandriva Linux. Begge er projekter med åben kildekode (Open Source) og kan gratis hentes på henholdsvis <http://openoffice.org> og <http://mandriva.com>.

Og tak til min kære kone, Anne Mette, for hendes støtte og forståelse mens vi for en toårig periode bor med vores to små børn i Nepals hovedstad Kathmandu.

*Bonan plezuron legi la libron!*

(det er esperanto og betyder "god fornøjelse med at læse bogen")

Jacob Nordfalk

Kathmandu, Nepal, september 2007.

# 1 Introduktion

Indhold:

- Programmering
- Fordele og ulemper ved Java
- Værktøjer til at programmere i Java

Hvis du har lyst til at komme i gang med at programmere, kan du springe over dette kapitel.

# 1.1 Programmering

Ethvert program, f.eks. et tekstbehandlingsprogram, regneark, e-post, tegneprogram, spil, webserver består af nogle data (f.eks. hjælpefiler og konfigurationsfiler) og en samling instruktioner til computeren.

Hver instruktion er meget simpel og computeren udfører den ubetinget, uanset om det er smart eller ej. Den kan udføre instruktionerne ekstremt hurtigt (over 1 milliard pr. sekund) og det kan få computeren til at virke smart, selvom instruktionerne er simple.

## 1.1.1 Styresystemet

Styresystemet er det program, som styrer computeren og tillader brugeren at bruge andre programmer. Af styresystemer kan nævnes Linux, Windows, MacOS, UNIX.

Styresystemet styrer computerens hukommelse og eksterne enheder som skærm, tastatur, mus, disk, printere og netværksadgang. Det tilbyder tjenester til programmerne, f.eks. muligheden for at læse på disken eller tegne en grafisk brugergrænseflade.

Et program kan normalt kun køre på et bestemt styresystem. Javaprogrammer kan dog køre på flere styresystemer og de bruges derfor bl.a. som programmer, der automatisk hentes ned til brugerens netlæser/browser og afvikles der. Den type programmer kaldes appletter eller miniprogrammer.

## 1.1.2 Hvorfor lære at programmere?

Det er sjovt og spændende og det kan være en kilde til kreativitet og leg, at skabe sine egne programmer. Man kan bedre forestille sig nye løsninger og produkter og man får bedre kendskab til computeres formåen og begrænsninger.

Desuden er det et håndværk, der er efterspurgt blandt IT-virksomheder og mange andre. Ved hjælp af programmering kan du løse problemer og du er dermed ikke mere afhængig af, at andre laver et program, der opfylder dine behov.

Programmering er en af datalogiens helt basale discipliner og selv om man ikke arbejder som programmør, er kendskab til programmering en fordel i mange beslægtede fag.

Java er et sprog, der har stor udbredelse såvel i industrien som i akademiske kredse. Det er kraftfuldt og relativt let lært. Lærer du Java, har du et godt fundament til at lære andre programmeringssprog.

## 1.1.3 Et simpelt program

For at computeren kan arbejde, skal den have nogle instruktioner, den kan følge slavisk. For at lægge to tal, som brugeren oplyser, sammen kunne man forestille sig følgende opskrift:

```
1  Skriv "Indtast første tal" på skærmen
2  Læs tal fra tastaturet
3  Gem tal i lagerplads A
4  Skriv "Indtast andet tal" på skærmen
5  Læs tal fra tastaturet
6  Gem tal i lagerplads B
7  Læg indhold af lagerplads A og indhold af lagerplads B sammen
8  Gem resultat i lagerplads C
9  Skriv "Summen er:" på skærmen
10 Skriv indhold af lagerplads C på skærmen
```

Et program minder lidt om en kagebogsopskrift, som computeren følger punkt for punkt ovenfra og ned. Hvert punkt (eller instruktion eller kommando) gøres færdigt, før der fortsættes til næste punkt.

## 1.1.4 Hvordan lærer man at programmere

Man lærer ikke at programmere blot ved at læse en bog. Har man ikke tid til at øve sig og eksperimentere med det man læser om, spilder man bare sin tid. For de fleste kræver det en stor arbejdsindsats at lære, at programmere og for alle tager det lang tid, før de bliver rigtig dygtige til det.

---

**Der er kun én måde at lære at programmere på: Øv dig**

---

Der er blevet lavet forskning, der underbygger dette. P.M. Cheney konkluderer<sup>1</sup>, at den eneste betydende faktor i produktiviteten for programmører er: Erfaring. Han fandt i øvrigt forskelle i produktiviteten på en faktor 25.

## 1.2 Fordele og ulemper ved Java

Java er et initiativ til at skabe et programmeringssprog, der kan køre på flere styresystemer. Det er udviklet af Sun Microsystems, der i 1991 arbejdede med at designe et programmeringssprog, der var velegnet til at skrive programmer til fremtidens telefoner, fjernsyn, opvaskemaskiner og andre elektroniske apparater. Sådanne programmer skal være meget kompakte (begrænset hukommelseslager) og fejlsikre (risikoen for, at apparatet ikke virker, skal være minimal).

Med udviklingen af internettet blev Java samtidig meget udbredt, fordi teknologien bl.a. tillader, at små programmer kan lægges ind i en hjemmeside (se kapitlet om appletter).

### 1.2.1 Stærke sider

Sproget har på bemærkelsesværdigt kort tid udviklet sig, til at være fremherskende på grund af dets egenskaber. Java er et enkelt, objektorienteret, robust, netværksorienteret, platformuafhængigt, sikkert, fortolket, højtydende, flertrådet og dynamisk sprog:

- **Enkelt.** Java er i forhold til andre programmeringssprog et ret enkelt sprog og det er forholdsvis nemt at programmere (specielt for C- og C++ -programmører). Mange af de muligheder for at lave fejl, der eksisterer i andre programmeringssprog, er fjernet i Java.
- **Enorm brugerbase.** Java er et af verdens mest populære programmeringssprog<sup>2</sup>. De fleste undervisningsinstitutioner har ligeledes valgt Java som gennemgående sprog i undervisningen.
- **Objektorienteret.** Sproget kommer med over 1000 foruddefinerede objekt-typer, som kan udføre næsten enhver tænkelig opgave. Præcist hvad "objektorienteret" betyder, handler denne bog om.
- **Platformuafhængigt.** Java er platformuafhængigt. Det vil sige, at samme program umiddelbart kan udføres på mange forskellige styresystemer, f.eks. UNIX, Linux, Mac og Windows, og processor-typer f.eks. Intel IA32, PowerPC og Alpha.
- **Netværksorienteret.** Java har indbygget alskens netværkskommunikation (se kapitlet om netværk) og bruges meget på internettet, da javaprogrammer kan køre på næsten alle platforme. Samtidig er Javaprogrammer så kompakte, at de nemt kan indlejres i en hjemmeside.

---

1 Artiklen hedder 'Effects of Individual Characteristics, Organizational Factors and Task Characteristics on Computer Programmer Productivity and Job Satisfaction' og kan findes i Information and Management, 7, 1984.

2 Se f.eks. på <http://www.tiobe.com/tpci.htm>

- **Fortolket.** Java-kildetekst oversættes til en standardiseret platformuafhængig kode (kaldet bytekode), som derefter udføres af en javafortolker på det enkelte styresystem. Der ved opnås, at man kun behøver at oversætte sin kildetekst én gang. Javafortolkeren er en såkaldt virtuel maskine, der konverterer instruktionerne i bytekoden til maskinkode-instruktioner, som det aktuelle styresystem kan forstå.
- **Højtydende.** De nuværende fortolkere tillader javaprogrammer at blive udført nogenlunde lige så hurtigt, som hvis de var blevet oversat direkte til det pågældende styresystem.
- **Flertrådet.** Java er designet til at udføre flere forskellige programdele samtidigt og en programudførelse kan blive fordelt over flere CPU'er (se kapitel 17 om flertrådet programmering).
- **Robust.** Under afviklingen af et program tjekkes det, at handlingerne er tilladelige og opstår der en fejl, såsom en ønsket fil ikke kan findes, fortæller Java, at der er opstået en undtagelse. I mange andre sprog vil sådanne uventede fejl føre til, at programmet stopper. I Java har man let adgang til at fange og håndtere disse undtagelser, så programmet alligevel kan køre videre (se kapitel 14 om undtagelser).
- **Sikkert.** Et sikkerhedssystem tjekker al programkode og sørger for, at bl.a. hjemmesider med Java-appletter ikke kan gøre ting, de ikke har lov til (f.eks. læse eller ændre i brugerens filer), uden at brugeren selv har givet tilladelse til det.
- **Dynamisk.** Java kan dynamisk (i et kørende program) indlæse ekstra programkode fra netværket og udføre den, når det er nødvendigt og der er indbygget mekanismer, til at lade programmer på forskellige maskiner dele dataobjekter (se f.eks. kapitel 19 om RMI).
- **Åben kildekode.** Sun har frigivet kildekoden (se afsnit 1.4.1) til Java, og det står derfor enhver frit for bl.a. at forbedre eller rette fejl i Java (eller få andre til at gøre det for sig).

## 1.2.2 Stor opbakning

Ovenstående egenskaber gør, at Java også har vundet stor udbredelse i serversystemer de seneste år og Java bakkes i dag op af næsten alle større softwarefirmaer, f.eks. IBM, Oracle, BEA og Borland.

Softwaregiganten Microsoft er en undtagelse. Microsoft er ikke interesseret i, at programmer kan udføres under andre styresystemer end Windows. Efter at være blevet kendt skyldig ved domstole i USA og Europa for at misbruge sin monopollignende magt på PC-markedet, bl.a. for at skade Java, ser det dog ud til, at fredelig sameksistens nu hersker.

## 1.2.3 Svagheder

Java har også nogle svagheder:

- Java kræver en del hukommelse (RAM). Store Javaprogrammer kan kræve så meget, at de har problemer med at køre på mindre kontor-PC'ere.
- Java skal installeres på en computer, før den kan afvikle javaprogrammer. Hvis man vil distribuere sit program, skal man således pakke en version af Java med.
- Java satser på at være platformuafhængigt, men der er alligevel små forskelle på de forskellige platforme. Det kræver erfaring og afprøvning at sikre sig, at ens program virker tilfredsstillende på flere platforme. Dette er ikke kun et Java-relateret problem, udviklere af f.eks. hjemmesider har tilsvarende problemer. Java gør det nemmere at skrive platformuafhængige programmer, men det løser ikke alle problemer for programmøren.

## 1.3 Programmeringsværktøjer til Java

Under programudviklingen har man brug for arbejdsredskaber, der kan hjælpe en med:

- Redigering af kildeteksten.
- Oversættelse af kildeteksten til binær kode.
- Kørsel og fejlfinding.

De fleste foretrækker at bruge et grafisk udviklingsværktøj (f.eks. JBuilder vist herunder). De er bygget op med en menulinje øverst, der indeholder tilgang til filhåndtering, projektstyring og alle nødvendige værktøjer, hvoraf de vigtigste er "Run" og "Debug". "Run" oversætter først kildeteksten og kører derefter programmet. Uddata kan ses i en ramme nederst. "Debug" (der findes under "Run") bruges til fejlfinding og giver mulighed for at udføre koden trinvist og følge med i variablenes værdier.

I et udviklingsværktøj arbejder man som regel med et "projekt", der er en liste over kilde tekst-filer og alle indstillinger for, hvordan programmet skal køres.

---

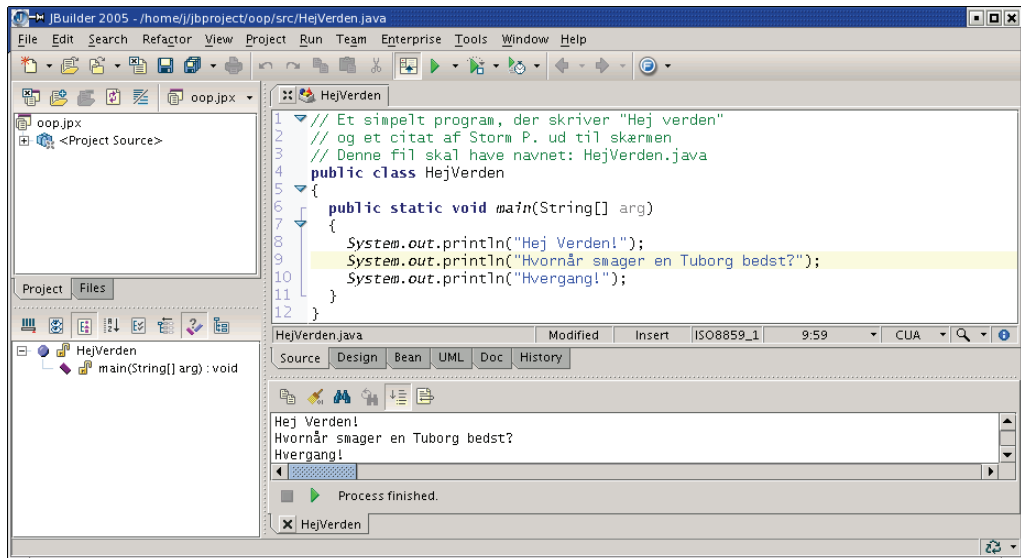
På <http://javabog.dk> kan bogens eksempler, sammen med projektfiler til de fleste af nedenstående programmeringsværktøjer, hentes

---

Herunder ses projektet oop.jpr i venstre side (i værktøjet JBuilder). I højre side ses kildeteksten på et faneblad. På de andre faneblade er typisk designværktøj til grafiske brugergrænseflader, dokumentation og versionskontrol.

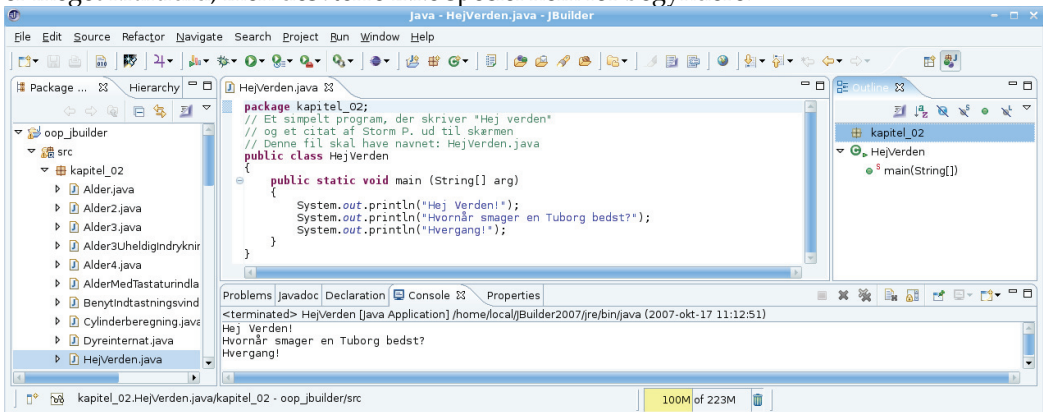
### 1.3.1 Borland JBuilder

En af de meget populære udviklingsværktøjer er JBuilder fra Borland. JBuilder er skrevet i Java og kan derfor bruges på både Linux, Macintosh, Windows og Sun Solaris. Det anbefales at have 512 MB RAM eller mere.



Ovenstående billede viser, hvordan JBuilder 2005 og 2006 ser ud. For at bruge bogens eksempler vælges File | Open Project og filen oop\_jbuilder.jpx åbnes. Derefter åbnes krydset ved <Project Source> og en javafil vælges. Højreklik og vælg 'Run' for at køre den.

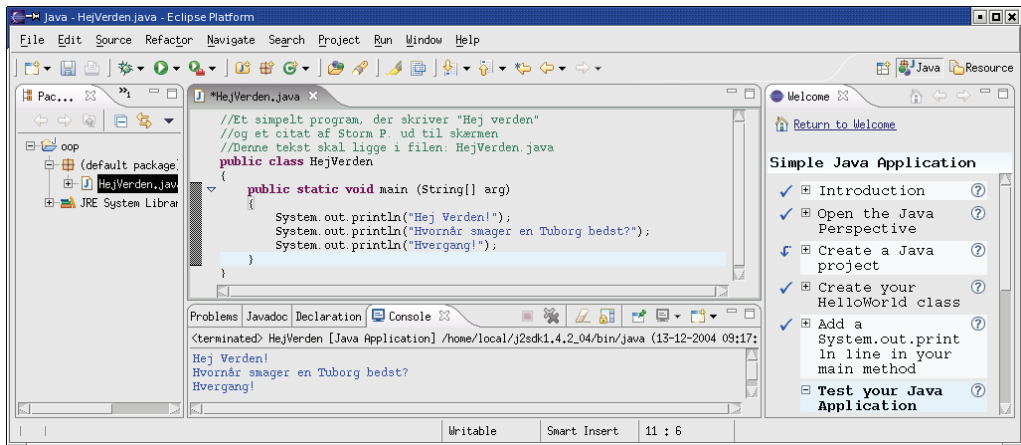
Fra og med JBuilder 2007 skiftede JBuilder til at bygge oven på Eclipse (se nedenfor), som er meget kraftfuld, men desværre ikke speciel nem for begyndere:



For at bruge bogens eksempler i JBuilder 2007 vælges File | New | Project... | Java Project from Existing JBuilder .jpx Project, og filen oop\_jbuilder.jpx åbnes. Derefter åbnes oop\_jbuilder | src og en javafil vælges. Højreklik og vælg 'Run as | Java Application' for at køre den.

En basisversion af JBuilder kan hentes gratis fra <http://www.borland.com/jbuilder>. Ønsker man adgang til de mere avancerede funktioner (som dog er unødvendige for en begynder og kun forvirrer), skal programmet købes.

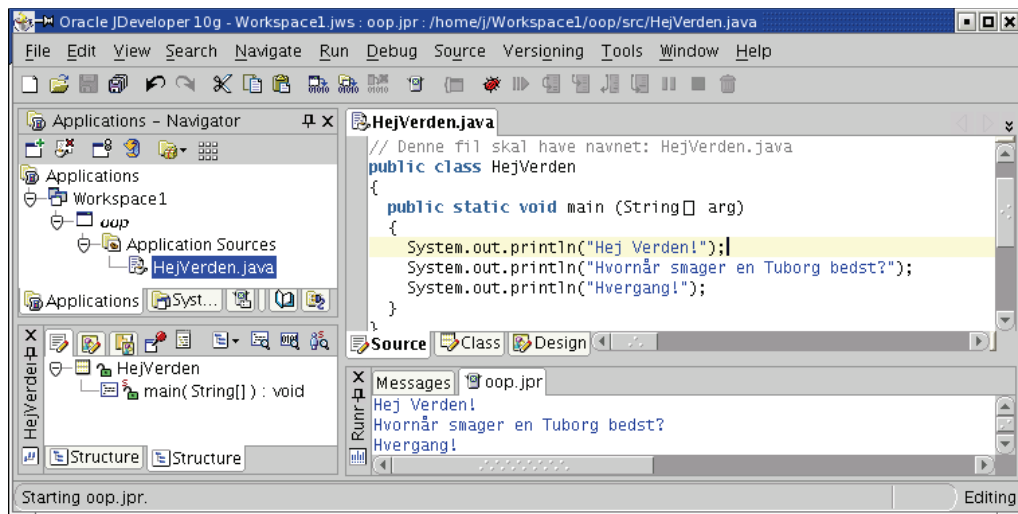
## 1.3.2 Eclipse.org



IBM har sit eget udviklingsmiljø til bl.a. Java, også med åben kildekode. under navnet Eclipse. Eclipse, der kræver 512 MB RAM, kan hentes gratis på <http://eclipse.org>.



## 1.3.3 Oracle JDeveloper



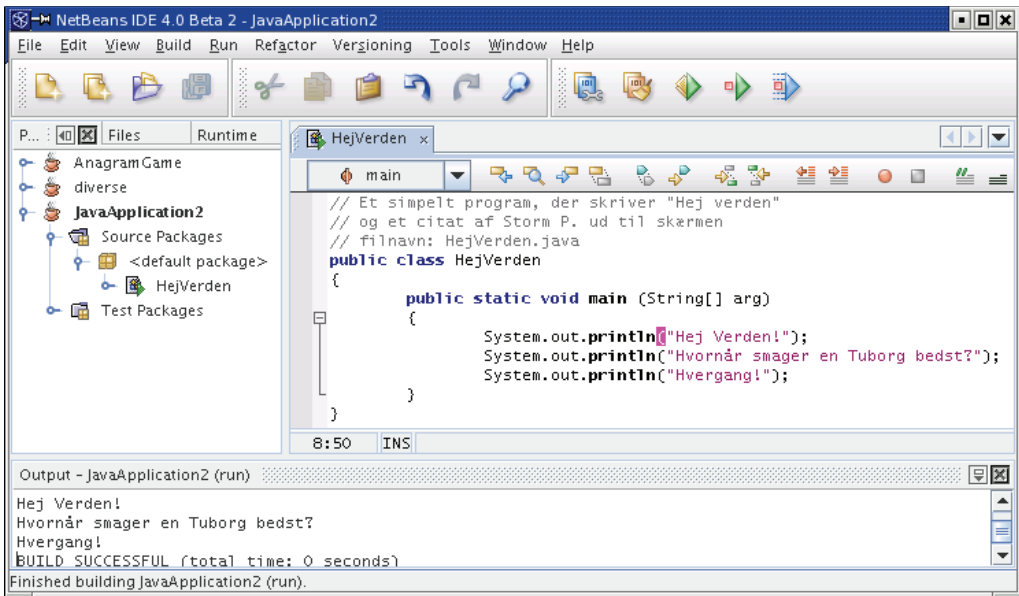
Databaseproducenten Oracle udgiver JDeveloper, også skrevet i Java, til bl.a. Linux, Windows og Macintosh.

JDeveloper kan en masse meget avancerede ting, men er for begyndere lidt mere indviklet at bruge end JBuilder 2006, bl.a. fordi projekter er samlet i arbejdsområder (eng.: workspace), en facilitet man sjældent har brug for i starten. Kan du ikke få fat i JBuilder 2006, er JDeveloper dog et udmærket valg.

JDeveloper kræver mindst 512 MB RAM. Den fulde udgave kan hentes gratis til privat- og undervisningsbrug på <http://oracle.com/technology/software/products/jdev/>.

For at bruge bogens eksempler vælges File | Open... og filen oop\_jdeveloper.jpr åbnes og der klikkes OK et par gange. Derefter åbnes krydset ved oop\_jdeveloper og Application Sources og en javafil vælges. Højreklik og vælg 'Run' for at køre den.

## 1.3.4 NetBeans



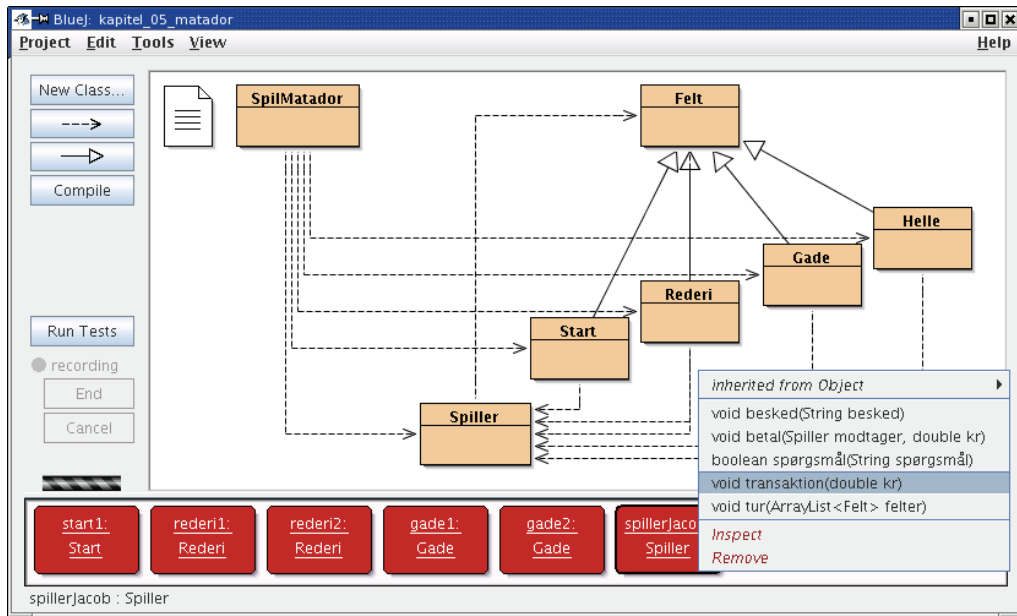
NetBeans er også skrevet i Java og kører på Linux, Windows, MacOS og Sun Solaris.

Programmet har åben kildekode (se afsnit 1.4.1) og kræver 512 MB RAM. Det kan hentes gratis på <http://netbeans.org> og er fint til begyndere.

For at bruge bogens eksempler vælges File | Open Project..., mappen oop-projekt vælges (du skal *ikke* dobbeltklikke) og der trykkes på knappen 'Open Project Folder'. Derefter åbnes oop-projekt | Source Packages og en javafil vælges. Højreklik og vælg 'Run File' for at køre den.

## 1.3.5 BlueJ

Et interessant værktøj til *undervisning* i objektorienteret programmering er BlueJ. Det hentes gratis på <http://bluej.org> og kræver 128 MB RAM. Det har en overskuelig brugergrænseflade, der lader eleven oprette objekter interaktivt, kalde metoder i dem og se resultatet:



Bogens eksempler er forberedt til BlueJ (se afsnit 4.10.1). Ovenfor ses således matadorspillet fra kapitel 5. Øverst er klasserne, nederst objekterne som eleven har oprettet. Til højre ses, hvordan eleven er ved at kalde metoden `transaktion()` på spilleren Jacob.

Desværre mangler BlueJ den avancerede syntaks- og formateringshjælp som er tilgængelig i de større værktøjer og som, efter forfatterens erfaring, er af afgørende betydning for begyndere.

Den er derfor fin som supplement men kan *ikke* erstatte et "rigtigt" udviklingsværktøj.

## 1.3.6 Andre værktøjer

Der findes mange andre udviklingsmiljøer til Java, bl.a. JCreator (<http://jcreator.com/> – kræver kun 128 MB RAM, men kun til Windows), IntelliJ (<http://jetbrains.com>) og Simpli-city (<http://datarepresentations.com>). De fleste findes i en prøveudgave, der kan hentes gratis fra internettet og som har alle de nødvendige faciliteter til at lave mindre programmer.

## 1.3.7 Sun JDK

Den mest skrabede løsning man kan vælge at redigere kildeteksterne i, er et ikke-Java-orienteret program, som for eksempel Notesblok under Windows eller kedit under Linux.

Til oversættelse og kørsel kan man installere et Java-udviklingskit udgivet af Sun, f.eks. JDK 1.6 (Java Development Kit version 1.6, kaldes også J2SE version 6.0). Det kan hentes gratis på <http://java.sun.com> til et væld af styresystemer.

JDK'et bruges fra kommandolinjen (f.eks. i et DOS-vindue). De vigtigste kommandoer er **javac**, der oversætter en kildetekstfil til bytekode og **java**, der udfører en bytekode-fil.

## 1.4 Hjælp og videre læsning

Mens du læser bogen, kan det være nyttigt at kigge på:

- Bogens hjemmeside med øvelser og ekstra materiale: <http://javabog.dk>
- Javadokumentationen: <http://java.sun.com/javase/reference/api.jsp>
- Vejledninger og introduktioner: <http://java.sun.com/docs/books/tutorial>
- OSS – ofte stillede spørgsmål om alt om Java: <http://jguru.com/faq>
- Du kan også deltage i diskussionsgruppen [dk.edb.programmering.java](http://dk.edb.programmering.java), hvor man kan få hjælp og finde mange relevante spørgsmål og svar. Der er 5-10 indlæg om dagen. Har du ikke adgang til diskussionsgrupper (eng.: newsgroups), kan de også læses via <http://groups.google.com/>. Den direkte henvisning er: <http://groups.google.com/groups?group=dk.edb.programmering.java>
- Et andet sted med en del spørgsmål og svar er <http://eksperten.dk> under Java.
- Eksemplerne (med kildetekst), der følger med, når et JDK installeres (kig i [jdk1.6/demo](#))
- På <http://javaspil.dk>, <http://spil.tv2.dk> og <http://vredungmand.dk> kan du se nogle ganske flotte spil skrevet i Java (uden kildetekst).

Hvis du efter denne bog vil læse mere om Java på dansk, kan du kigge på:

- Mine andre bøger om programmering i Java, på <http://javabog.dk>.
- Kristian Hansen: Avanceret Java-programmering, Ingeniøren i bøger, 1998.
- Skåne-Sjælland Linux-brugergruppens javabog: <http://sslug.dk/bog>

### 1.4.1 Projekter med åben kildekode

Åben kildekode (eng.: open source) vil sige, at kildeteksten er frit tilgængelig. Dette er i modsætning til de fleste kommercielle producenter, der holder kildeteksten for sig selv.

For en lettere øvet i Java og programmering er projekter med åben kildekode interessante, som kilde til inspiration og som eksempler på, hvordan man gør forskellige ting.

Et sted med åben kildekode-projekter er <http://sourceforge.net>, hvor der er fri adgang til titusindvis af projekter skrevet i Java. Java-siden <http://foundries.sourceforge.net/java> byder på generelle nyheder om Java og åben kildekode.

Åben kildekode blev før i tiden mest produceret af universiteter og idealistisk indstillede firmaer og enkeltpersoner. Styresystemet Linux er udelukkende baseret på åben kildekode.

Imidlertid er der sket et skred mod mere og mere åben kildekode, hvor store firmaer som Novell, Sun og IBM har ført an. Det er indlysende, at hvis programudviklingen sker åbent, sådan at alle kan være med til at afprøve programmet, komme med forbedringsforslag, kigge i kildeteksten og selv bidrage til at forbedre programmet, kan det have en meget positiv indvirkning på et programs kvalitet og opbakning blandt brugerne.

Samtidig føler mange, at de lukkede, proprietære systemers tid er ved at være forbi, både fordi man som forbruger er prisgivet producenten m.h.t. fejlrettelser og nye versioner, men også fordi lukkede systemer har en indre tendens, til at bevæge sig mod større og større monopoler og dermed mindre nyskabelse og sund konkurrence.

# 2 Basal programmering

Indhold:

- Variabler, tildelinger og regneudtryk
- Forgreninger og løkker
- Kald af metoder

Kapitlet forudsættes i resten af bogen.

## 2.1 Det første javaprogram

Lad os se på et simpelt javaprogram, der skriver "Hej verden" og et citat af Storm P. ud til skærmen. Neden under den vandrette linje er vist, hvad der sker, hvis programmet køres:

```
// Et simpelt program, der skriver "Hej verden"
// og et citat af Storm P. ud til skærmen
// Denne fil skal have navnet: HejVerden.java
public class HejVerden
{
    public static void main (String[] arg)
    {
        System.out.println("Hej Verden!");
        System.out.println("Hvornår smager en Tuborg bedst?");
        System.out.println("Hvergang!");
    }
}
```

---

```
Hej Verden!
Hvornår smager en Tuborg bedst?
Hvergang!
```

Alle javaprogrammer har den samme grundlæggende struktur, som også kan ses af dette eksempel.

### 2.1.1 Kommentarer

Kommentarer er dokumentation beregnet på, at gøre programmets kildetekst lettere at forstå for læseren. De påvirker ikke programudførelsen.

De første 3 linjer, der starter med `//`, er kommentarer:

```
// Et simpelt program, der skriver "Hej verden"
// og et citat af Storm P. ud til skærmen
// Denne fil skal have navnet: HejVerden.java
```

I dette tilfælde er der beskrevet, hvad programmet gør og hvilket filnavn kildeteksten bør gemmes i.

---

**Kommentarer bliver sprunget over og har ingen indflydelse på programmet**

**Kommentarer bør skrives, så de giver forståelse for, hvordan programmet virker - uden at være tvetydige eller forklare indlysende ting**

---

`//` markerer, at resten af linjen er en kommentar. Den kan også bruges efter en kommando til at forklare, hvad der sker, f.eks.

```
System.out.println("Hej verden!"); // Udskriv en hilsen
```

Java har også en anden form, som kan være nyttig til kommentarer over flere linjer: Man kan starte en kommentar med `/*` og afslutte den med `*/`. Al tekst mellem `/*` og `*/` bliver så opfattet som kommentarer. Vi kunne altså også skrive

```
/*
// Et simpelt program, der skriver "Hej verden"
// og et citat af Storm P. ud til skærmen
// Denne fil skal have navnet: HejVerden.java
*/
```

og

```
System.out.println("Hej verden!"); /* Udskriv en hilsen */
```

I denne bog skriver vi kommentarer i *kursiv* for at lette læsningen af eksemplerne.

## 2.1.2 Klassedefinitionen

Resten af teksten kaldes en klassedefinition og beskriver selve programmet (HejVerden).

Den består af en fast struktur:

```
public class HejVerden
{
    public static void main (String[] arg)
    {
        ...
    }
}
```

og noget programkode – kommandoer, der skal udføres, nærmest som en bageopskrift:

```
System.out.println("Hej verden!");
```

### Strukturdelen

Strukturdelen vil ikke blive ændret i de næste to kapitler og det er ikke så vigtigt, at du forstår, hvad der foregår i første omgang.

Al javakode er indkapslet i en klasse mellem { og } (blokstart og blokslut-parenteser). Beskrivelsen af en klasse er altid indkapslet i en blok bestående af:

```
public class HejVerden
{
    ...
}
```

Inde i klassen står der en main-metode med nogle kommandoer i.

```
public static void main (String[] arg)
{
    ...
}
```

Indholdet af metoden er altid indkapslet i en blok med { og }.

---

**Programudførelsen starter i metoden:**

```
public static void main (String[] arg)
```

---

### Programkode

I main-metoden giver man instruktioner til computeren:

```
System.out.println("Hej verden!");
System.out.println("Hvornår smager en Tuborg bedst?");
System.out.println("Hvergang!");
```

Instruktionerne udføres altid en efter en, ovenfra og ned. Hver instruktion afsluttes med et semikolon.

Disse 3 instruktioner skriver 3 strenge ("Hej verden!", ...) ud til skærmen. En streng er en tekst, som computeren kan arbejde med. Strenge er altid indkapslet i "".

Hver instruktion består af et kald til metoden System.out.println, som betyder, at der skal udskrives noget til skærmen og en streng som parameter.

En parameter er en oplysning, som man overfører til metoden. I dette tilfælde hvilken tekst, der skal skrives ud til skærmen.

Vores main-metode kalder altså andre metoder.

## 2.1.3 Oversættelse og kørsel af programmet

Når man skal udvikle et program, skriver man først en kildetekst (eng.: source code), der beskriver, hvad det er, man vil have programmet til at gøre. Programmet, vi lige har set, er et eksempel på en kildetekstfil.

Instruktionerne, som centralenheden i computeren arbejder med, er i en binær kode (kaldet maskinkode eller bytekode), der er umulig at læse for almindelige mennesker.

Kildeteksten skal derfor oversættes (eng.: compile; mange siger også kompilere på dansk) til den binære kode, som så kan udføres af computeren.

I Java kalder man den binære kode for bytekode. Bytekode er platformuafhængigt, dvs. at det kan køre på stort set alle hardware-platforme og alle styresystemer. De fleste andre sprogs binære kode er ikke indrettet til at være platformuafhængigt.

For at oversætte programmet HejVerden skal det gemmes i en fil med navnet "HejVerden.java".

---

**En kildetekstfil skal hedde det samme som klassen og skal have .java som filendelse**

---

Eksempel: Klassen hedder HejVerden og filen hedder HejVerden.java.

## Oversættelse og kørsel fra kommandolinjen

Hvis du bruger det kommandolinje-orienterede JDK direkte, skal du åbne en DOS/UNIX-kommandoprompt<sup>1</sup> og stå i den samme mappe som kildeteksten findes<sup>2</sup>. Skriv så<sup>3</sup>:

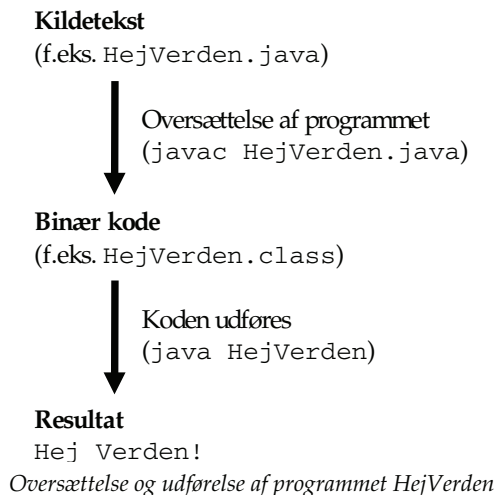
```
javac HejVerden.java
```

Dette oversætter programmet til bytekode (filen HejVerden.class skulle nu gerne ligge i samme mappe). Nu kan du køre programmet med kommandoen<sup>4</sup>:

```
java HejVerden
```

Resultatet udskrives i vinduet:

```
Hej Verden!  
Hvornår smager en Tuborg bedst?  
Hvergang!
```



---

1 I Windows åbner du en DOS-prompt ved at klikke i menuen Start, vælge 'Kør...' og skrive 'cmd'.

2 Er HejVerden.java i mappen C:\javafiler så skriv 'cd C:\javafiler' (og tryk retur-tasten).

3 Måske skal du angive den fulde sti til javac, f.eks. C:\jdk1.6\bin\javac HejVerden.java

4 Kan klassen HejVerden ikke findes (NoClassDefFoundError) så prøv: java -cp . HejVerden



## Oversættelse og kørsel med et udviklingsværktøj

I de fleste udviklingsværktøjer skal du oprette et *projekt* (f.eks. i JBuilder: File/New Project). Føj derefter din java-fil til projektet. Husk at placere filen i den mappe, som projektet angiver (eller rette projektets egenskaber).

Når man vil oversætte sit java-program, skal man vælge *make* (det er et engelsk ord). Når man har gjort det, kan man køre sit program med *run*.

## 2.2 Variabler

Variabler bruges til at opbevare og ændre data. En variabel kan opfattes som en navngiven papirlap, hvor der til enhver tid kan stå netop én ting.

Variabler skal altid erklæres, dvs. at man skal fortælle computeren, at der skal oprettes en variabel, hvad slags data den skal indeholde og hvad den skal hedde.

---

**En variabel er et navn på et sted i computerens hukommelse, beregnet på at indeholde data af en bestemt type**

**En variabel erklæres ved at skrive *variabeltype variabelnavn*;**

---

Det er en god vane, at give variablerne sigende navne. Navnene bør starte med et lille bogstav.

I det følgende gennemgår vi to af Javas variabeltyper: `int` (heltal) og `double` (kommatal).

### 2.2.1 Heltal

En variabel af typen `int` (et heltal, eng.: integer) erklæres med

```
int tal;
```

Nu er der reserveret plads i hukommelsen til et heltal<sup>1</sup>. Man får fat i pladsen, ved at bruge variabelnavnet `tal`. Efter at variabelen er erklæret, kan den tildeles en værdi, dvs. man kan skrive data ind i den:

```
tal = 22;
```

Nu er værdien af `tal` 22 (vist på figuren til højre).

Vi kan bruge `tal`-variablen i stedet for at skrive 22, f.eks. til at skrive ud til skærmen:

```
System.out.println("Svaret på livet, universet og alt det der: " + tal);
```

Her slår computeren op i hukommelsen, læser indholdet af `tal`-variablen og skriver det ud til skærmen (+et vil blive forklaret i næste afsnit).

Variabler kan, som navnet siger, ændre værdi. Det gør vi, ved at tildele variabelen en ny værdi:

```
tal = 42;
```

Herefter er den gamle værdi fuldstændigt glemt og erstattet med den nye. Når programudførelsen når et punkt, hvor variabelen læses, vil det være den nye værdi, 42, der gælder.

tal

22

Efter 1.  
tildeling

tal

42

Efter 2.  
tildeling

---

**I en tildeling læses værdien på højre side og gemmes i variabelen på venstre side**

---

<sup>1</sup> Nøjere bestemt til et tal mellem -2147483648 og 2147483647, da der kun er reserveret 4 byte, dvs. 32 bit.

Herunder er eksemplet i sin helhed (den væsentlige del af koden er fremhævet med fed):

```
// Eksempel på brug af en variabel
// koden skal være i filen Variabler.java
public class Variabler
{
    public static void main (String[] arg)
    {
        int tal;
        tal = 22;
        System.out.println("Svaret på livet, universet og alt det der: " + tal);

        tal = 42;
        System.out.println("Undskyld, svaret er: " + tal);
    }
}
Svaret på livet, universet og alt det der: 22
Undskyld, svaret er: 42
```

## 2.2.2 Sammensætte strenge med +

Som det er vist i ovenstående eksempel, kan vi med tegnet +, sætte strenge sammen med noget andet:

```
System.out.println("Svaret på livet, universet og alt det der: " + tal);
```

Herunder sætter vi to strenge sammen:

```
// Sammensæt to strenge med +
// koden skal være i filen HejVerden2.java
public class HejVerden2
{
    public static void main (String[] arg)
    {
        System.out.println("Hej " + "Verden!");
    }
}
Hej Verden!
```

Herunder skriver vi en streng og tallet 42 ud:

```
public class HejVerden3
{
    public static void main (String[] arg)
    {
        System.out.println("Svaret på livet, universet og alt det der: " + 42);
    }
}
Svaret på livet, universet og alt det der: 42
```

Det der egentlig sker er, at det hele bliver sat sammen til én streng og den sendes til System.out.println().

---

**En streng + noget andet sættes sammen til en samlet streng**

---

## 2.2.3 Beregningsudtryk

Man kan erklære flere variabler på samme linje:

```
int antalHunde, antalKatte, antalDyr;
antalHunde = 5;
antalKatte = 8;
```

Tildelinger kan indeholde regneudtryk på højre side af lighedstegnet. Udtrykket antalHunde + antalKatte udregnes og resultatet lægges i variablen på venstre side (det er ikke tilladt at have beregningsudtryk på venstre side):

```
antalDyr = antalHunde + antalKatte;
```

Beregningsudtrykkene undersøges af Java ved at indsætte værdien af variablerne. Her ind sætter Java  $5 + 8$  og får 13, som lægges i antalDyr.

```
public class Dyreinternat
{
    public static void main(String[] arg)
    {
        int antalHunde, antalKatte, antalDyr;
        antalHunde = 5;
        antalKatte = 8;

        //udregn summen
        antalDyr = antalHunde + antalKatte;

        // udskriv resultatet
        System.out.println("Antal dyr: " + antalDyr);

        antalHunde = 10;

        // antalDyr er uændret
        System.out.println("Antal dyr nu: " + antalDyr);
    }
}
```

Antal dyr: 13  
Antal dyr nu: 13

Beregningen sker én gang på det tidspunkt, hvor kommandoen udføres<sup>1</sup>. Derfor er antalDyr's værdi ikke påvirket af at, vi sætter antalHunde til noget andet efter udregningen.

Ligesom i almindelig matematik har \* (multiplikation) og / (division) højere prioritet end + og -.

---

### I Java skrives

"9 divideret med 3" som  $9/3$

"3 gange 3" som  $3*3$

---

Man kan ikke, som i almindelig matematisk notation, undlade at skrive gangetegn.

Resultatet af en heltalsudregning er også et heltal. Det skal man være opmærksom på ved division, hvor eventuelle decimaler efter kommaet smides væk. Heltalsudregningen  $13 / 5$  giver altså 2, fordi 5 går op i 13 to gange<sup>2</sup>.

---

### Et heltal divideret med et heltal giver et heltal

$95 / 100$  giver 0

---

Ønsker man at få et kommatotal som resultat af divisionen, skal et eller begge af tallene være kommatotal. Eksempelvis giver  $95.0 / 10$  kommatallet 9.5.

## 2.2.4 Kommatal

Der findes mange andre variabeltyper end heltalstypen int. Hvis man vil regne med kommatotal, bruger man typen double. En variabel af typen double erklæres med:

```
double højde;
```

De følgende afsnit bruger noget matematik, mange lærer i gymnasiet. Hvis du ikke kender så meget til matematik, gør det ikke noget. Præcis hvad der udregnes og formlerne bag det, er ikke så vigtigt i denne sammenhæng. Det vigtige er at forstå hvordan man arbejder med tal i Java.

---

1 Der er altså ikke tale om en matematisk ligning, der altid skal gælde, men om en tildeling, hvor variabelen på venstre side får tildelt værdien af udtrykket på højresiden.

2 Man kan med operatoren % finde resten af divisionen.  $13 \% 5$  giver altså 3, fordi  $13 - 5*2 = 3$ .

Her er et eksempel på beregning af en cylinders rumfang:

```
//  
// Beregning af rumfang for en cylinder  
//  
public class Cylinderberegning  
{  
    public static void main(String[] arg)  
    {  
        double radius;  
        radius = 5.0;  
  
        double højde = 12.5;  
  
        //beregner rumfang  
        double volumen = radius * radius * højde * 3.14159;  
  
        System.out.println("Cylinderens højde: " + højde);  
        System.out.println("Cylinderens radius: " + radius);  
        System.out.println("Cylinderens volumen: " + volumen);  
    }  
}  
Cylinderens højde: 12.5  
Cylinderens radius: 5.0  
Cylinderens volumen: 981.7468749999999
```

Læg mærke til, at man godt kan erklære en variabel og tildele den værdi i samme linje:

```
double højde = 12.5;
```

er altså det samme som:

```
double højde;  
højde = 12.5;
```

Her er et eksempel på en skatteberegning, der viser nogle flere fif:

```
//  
// Skatteberegning (Inspireret af Hallenberg og Sestoft, IT-C, København)  
//  
public class Skatteberegning  
{  
    public static void main(String[] arg)  
    {  
        double indkomst = 300000;  
        double ambi, pension, bundskat;  
  
        ambi = indkomst * 0.08;  
        pension = indkomst * 0.01;  
        indkomst = indkomst - (ambi + pension);  
        bundskat = (indkomst - 33400) * 0.07;  
  
        System.out.println("AMBI: " + ambi);  
        System.out.println("Særlig pensionsopsparing: " + pension);  
        System.out.println("Bundskat: " + bundskat);  
    }  
}  
AMBI: 24000.0  
Særlig pensionsopsparing: 3000.0  
Bundskat: 16772.0
```

Udregninger sker normalt fra venstre mod højre, men ligesom i den almindelige matematik kan man påvirke udregningsrækkefølgen ved at sætte parenteser:

```
bundskat = (indkomst - 33400) * 0.07;
```

## 2.2.5 Matematiske funktioner

Dette afsnit handler om de matematiske funktioner som sinus, cosinus, kvadratrods osv. som man lærer om i f.eks. den matematiske gren af gymnasiet. Kender du ikke disse begreber så spring let hen over afsnittet, da de *ikke* er nødvendige for at lære at programmere.

Funktionerne kaldes i Java med `Math.sin(x)` for sinus, `Math.cos(x)` for cosinus, `Math.pow(x, y)` for potens, `Math.sqrt(x)` for kvadratrods, `Math.sqr(x)` for  $x^y$  osv., hvor  $x$  og  $y$  er variable, faste tal eller beregningsudtryk.

Vi kan f.eks. lave en tabel over værdierne af kvadratrods-funktionen `Math.sqrt()` for  $x=0$  til  $x=10$  med programmet (senere, i afsnit 2.5 om løkker vil vi se en smartere måde).

```
public class Kvadratrods
{
    public static void main(String[] arg)
    {
        System.out.println("kvadratrods af 0 er " + Math.sqrt(0));
        System.out.println("kvadratrods af 1 er " + Math.sqrt(1));
        System.out.println("kvadratrods af 2 er " + Math.sqrt(2));
        System.out.println("kvadratrods af 3 er " + Math.sqrt(3));
        System.out.println("kvadratrods af 4 er " + Math.sqrt(4));
        System.out.println("kvadratrods af 5 er " + Math.sqrt(5));
        System.out.println("kvadratrods af 6 er " + Math.sqrt(6));
        System.out.println("kvadratrods af 7 er " + Math.sqrt(7));
        System.out.println("kvadratrods af 8 er " + Math.sqrt(8));
        System.out.println("kvadratrods af 9 er " + Math.sqrt(9));
        System.out.println("kvadratrods af 10 er " + Math.sqrt(10));
    }
}

kvadratrods af 0 er 0.0
kvadratrods af 1 er 1.0
kvadratrods af 2 er 1.4142135623730951
kvadratrods af 3 er 1.7320508075688772
kvadratrods af 4 er 2.0
kvadratrods af 5 er 2.23606797749979
kvadratrods af 6 er 2.449489742783178
kvadratrods af 7 er 2.6457513110645907
kvadratrods af 8 er 2.8284271247461903
kvadratrods af 9 er 3.0
kvadratrods af 10 er 3.1622776601683795
```

Her er et program, der udregner længden af den skrå side (hypotenusen) af en retvinklet trekant ud fra længden af dens to lige sider (kateter)  $a$  og  $b$  (kvadratrods af  $a^2+b^2$ ):

```
public class Trekant
{
    public static void main(String[] arg)
    {
        double a, b, hypotenusen;
        a = 3;
        b = 4;
        hypotenusen = Math.sqrt(a*a + b*b);
        System.out.println("En retvinklet trekant med sider "+a+" og "+b);
        System.out.println("har hypotenusen "+hypotenusen);
    }
}

En retvinklet trekant med sider 3.0 og 4.0
har hypotenusen 5.0
```

Her er et program, der udregner, hvor meget 1000 kroner med 5 % i rente i 10 år bliver til:

```
public class Rentesregning
{
    public static void main(String[] arg)
    {
        System.out.println("1000 kr med 5 % i rente på 10 år giver "
            + 1000*Math.pow(1.05,10) + " kroner.");
    }
}

1000 kr med 5 % i rente på 10 år giver 1628.8946267774422 kroner.
```

Ud over de almindelige matematiske funktioner findes også `Math.random()`, der giver et tilfældigt tal mellem 0 og 0.999999...

## 2.2.6 Kald af metoder

`Math.sqrt()`, `Math.sin()` og de andre matematiske funktioner og andre kommandoer, f.eks. `System.out.println()`, kaldes under et *metode*.

En metode er en navngiven programstump, der kan gøre et eller andet eller beregne en værdi. F.eks. gør `System.out.println()` det, at den skriver tekst på skærmen og `Math.sqrt()` beregner en kvadratrods. Når en metode nævnes i teksten, skriver vi altid "()" bagefter, så man kan se, at det er en metode.

Nedenstående linje indeholder et *metodekald*:

```
hypotenuse = Math.sqrt(a*a + b*b);
```

`Math.sqrt` er navnet på metoden og man kalder det, der står inde i "()", for argumentet eller parameteren.

Et metodekald er en nævnelse af en metodes navn efterfulgt af de rigtige parametre. Parametrene er omgivet af parenteser.

---

**Ved et metodekald kan man som parameter indsætte ethvert udtryk, der giver et resultat af den rigtige type**

---

Alt, der giver et resultat af den rigtige type, er altså tilladt: Konstanter, variabler, regneudtryk og resultatet af et andet metodekald:

```
double v, x;
x = Math.sqrt(100);           // konstant som parameter
x = Math.sqrt(x);             // variabel som parameter
x = Math.sin(Math.sqrt(0.3)); // værdi af andet metodekald som parameter
```

Ved et kald uden parametre skal man stadig have parenteserne med. Her er et eksempel på et metodekald af `Math.random()`, som er en metode, der skal kaldes uden parametre:

```
double tilfældigtTal;
tilfældigtTal = Math.random();
```

Vi vil i kapitel 4 se, hvad der sker, når computeren udfører et metodekald samt lære, hvordan man kan lave sine egne metoder.

## 2.2.7 Logiske variabler

En boolesk<sup>1</sup> variabel (eng.: boolean), også kaldet en logisk variabel, kan indeholde værdien sand eller falsk. Den bruges mest til at huske, om noget er sandt eller ej, men kan også repræsentere noget, der kun har to tilstande, f.eks. om en lampe er tændt eller slukket.

Variabeltypen hedder boolean og den erklæres med f.eks.:

```
boolean detErForSent;
```

En boolesk variabel kan kun sættes til værdierne true eller false. F.eks.:

```
detErForSent = false;
```

På højre side af lighedstegnet kan stå et logisk udtryk, dvs. et udsagn, der enten er sandt eller falsk, f.eks. "klokken er over 8" (her forestiller vi os, at vi har variablen klokken).

```
detErForSent = klokken > 8;
```

Udtrykket `klokken > 8` undersøges af Java, ved at indsætte værdien af variablen i regneudtrykket og derefter afgøre, om udsagnet er sandt. Hvis f.eks. klokken er lig 7, står der `7>8`, det er ikke sandt og `detErForSent` får værdien false. Hvis klokken er lig 10, står der `10>8`, det er sandt og `detErForSent` får værdien true.

---

<sup>1</sup> Opkaldt efter den britiske matematiker George Boole, 1815-64.

## 2.2.8 Opgaver

- 1) Skriv et program, som ud fra længde og bredde på et rektangel udskriver dets areal.
- 2) Skriv et program, som for ligningen  $y=3*x*x+6*x+9$  udskriver værdien af  $y$  for  $x=7$ .
- 3) Skriv et program, som omregner et beløb fra dollar til euro (f.eks. kurs 95).
- 4) Skriv et program, som udskriver tre tilfældige tal (lavet med `Math.random()`), deres sum og gennemsnittet.
- 5) Hvad skriver følgende program ud? Hvis du kan regne det ud, *uden* at køre programmet, har du forstået idéen i tildelinger.

```
public class Tildelinger
{
    public static void main(String[] arg)
    {
        int a, b, c, d;
        a = 5;
        b = 6;
        c = 7;
        d = 8;
        System.out.println("a er "+a+", b er "+b+", c er "+c+" og d er "+d);

        a = b + d;
        d = c + a;
        System.out.println("a er "+a+", b er "+b+", c er "+c+" og d er "+d);

        b = a;
        d = c;
        System.out.println("a er "+a+", b er "+b+", c er "+c+" og d er "+d);
    }
}
```

## 2.3 Betinget udførelse

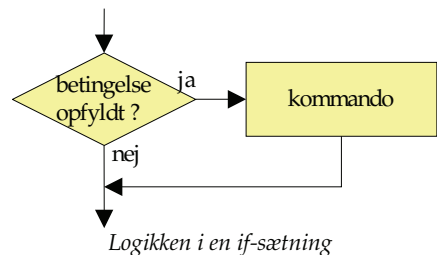
Indtil nu har vores programmer været fuldstændig forudsigelige. Vi har bedt computeren om at udføre den ene kommando efter den anden uanset udfaldet af de tidligere kommandoer.

I programmer kan man påvirke programudførelsen, ved at indføre betingelser, der fortæller, at en del af programmet kun skal gennemløbes, hvis betingelsen er opfyldt.

Det består af et udtryk, der enten er sandt eller falsk og noget, der afhænger af dets sandhedsværdi (se rutediagrammet til højre).

Alle kender betingelser fra deres dagligdag, f.eks.:

- hvis du er over 18, er du myndig.
- hvis din alkoholpromille er større end 0.5, så lad bilen stå.
- hvis den koster mindre end 500 kr, så køb den!



## I Java er syntaksen

```
if (betingelse) kommando;
```

For eksempel:

```
if (alder >= 18) System.out.println("Du er myndig");
if (alkoholpromille > 0.5) System.out.println("Lad bilen stå");
if (pris < 500) System.out.println("Jeg køber den!");
if (alder == 18) System.out.println("Du er præcis atten år.");
if (alder != 18) System.out.println("Du er ikke atten.");
```

Udtrykkene i parenteserne er logiske udtryk (eller booleske udtryk). På dansk er sætningen "over 18" tvetydig: skal man være OVER 18, dvs. 19, for at være myndig? Java har derfor to forskellige sammenligningsoperatore:  $a \geq b$  undersøger, om  $a$  er større end eller lig med  $b$ , mens  $a > b$  undersøger om  $a$  er større end  $b$ . I appendiks afsnit 2.11.6 findes en oversigt over sammenligningsoperatorene.

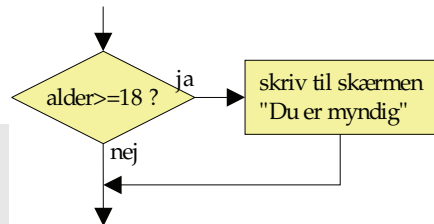
Herunder et komplet eksempel på et program, der afgør, om man er myndig. Programkoden, udtrykt på dansk, kunne være: hvis alder er større end 18, så skriv "Du er myndig". I Java skriver man:

```
public class Alder
{
    public static void main(String[] arg)
    {
        int alder;
        alder = 15;

        if (alder >= 18) System.out.println("Du er myndig.");

        System.out.println("Du er " + alder + " år.");
    }
}
```

Du er 15 år.



Rutediagram for Alder

Kommandoen `System.out.println("Du er myndig")` bliver kun udført, hvis betingelsen (`alder >= 18`) er sand. I dette tilfælde er `alder` lig 15 og der bliver ikke skrevet noget ud.

Hvis vi ændrer i programmet, så `alder` er 18, er betingelsen (`alder >= 18`) sand og vi får:

```
Du er myndig.
Du er 18 år.
```

Programudførelsen fortsætter under alle omstændigheder efter betingelsen, så uafhængigt af udfaldet, vil den sidste linje blive udført.

### 2.3.1 Indlæsning fra tastaturet

Et program bliver selvfølgelig først rigtig sjovt, hvis brugeren kan påvirke dets udførelse, f.eks. ved at programmet kan bede brugeren om at indtaste sin alder. Det kan gøres med:

```
public class AlderMedTastaturindlaesning
{
    public static void main(String[] arg)
    {
        java.util.Scanner tastatur = new java.util.Scanner(System.in); // forbered

        System.out.println("Skriv din alder herunder og tryk retur:");
        int alder;
        alder = tastatur.nextInt(); // læs et tal fra tastaturet

        if (alder >= 18) System.out.println("Du er myndig.");
        System.out.println("Du er " + alder + " år.");
    }
}
```

Programmet vil nu stoppe op og vente på, at der bliver indtastet noget efterfulgt af tryk på retur-tasten. Den øverste linje (med kommentaren *forbered*) skal kun forekomme én gang øverst i programmet (i kapitel 3 vil vi se nærmere på, hvad der sker i de to specielle linjer).



Du kan også få et grafisk indtastningsvindue, som brugeren kan udfylde til at dukke op (se afsnit 2.12.1).

## 2.3.2 if-else

Hvis vi ønsker at gøre én ting, hvis betingelsen er sand og en anden ting, hvis betingelsen er falsk, kan vi føje en else-del til vores if-sætning. Denne del vil kun blive udført, hvis betingelsen er falsk. Syntaksen er:

```
if (betingelse) kommando1;  
else kommando2;
```

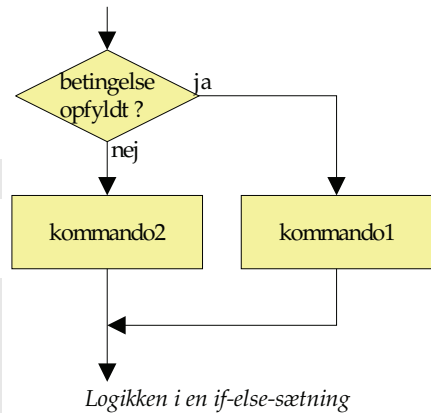
Eksempelvis:

```
public class Alder2  
{  
    public static void main(String[] arg)  
    {  
        int alder;  
        alder = 15;  
  
        if (alder >= 18)  
            System.out.println("Du er myndig.");  
        else  
            System.out.println("Du er ikke myndig.");  
  
        System.out.println("Du er " + alder + " år.");  
    }  
}
```

```
Du er ikke myndig.  
Du er 15 år.
```

Ændrer vi `alder = 15` til `alder = 18`, er betingelsen (`alder >= 18`) sand og vi får i stedet udskriften:

```
Du er myndig.  
Du er 18 år.
```



## Kædede if-else-sætninger

Bemærk at man med fordel kan kæde if-else-sætninger sammen. F.eks.:

```
if (alder >= 18) System.out.println("Du er myndig.");  
else if (alder >= 13) System.out.println("Du er teenager og ikke myndig.");  
else if (alder >= 2) System.out.println("Du er et barn og ikke myndig.");  
else System.out.println("Du er et spædbarn!");
```

## 2.3.3 Opgaver

- 1) Skriv Alder2 om til at indeholde flere aldergrænser.
  - 2) Lav et veksleprogram fra dollar til euro. Det skal påregne en kommission på 2 %, dog mindst 0,5 euro.
  - 3) Skriv et program, der beregner porto for et brev. Inddata er brevets vægt (i gram). Ud-data er prisen, for at sende det som A-post i Danmark.
  - 4) Skriv et program, som finder det største og det mindste af tre tal.
- Afprøv programmerne med forskellige værdier, ved at indlæse værdierne fra tastaturet.

## 2.4 Blokke

En blok er en samling af kommandoer. Den starter med en blokstart-parentes { og slutter med en blokslut-parentes}.<sup>1</sup>

---

### En blok grupperer flere kommandoer, så de udføres samlet

---

Blokke bruges blandt andet, hvis man vil have mere end førstkommande linje udført i en betingelse. Herunder udføres to kommandoer, hvis betingelsen er opfyldt og to andre kommandoer, hvis betingelsen ikke er opfyldt:

```
public class Alder3
{
    public static void main(String[] arg)
    {
        int alder;
        alder = 15;

        if (alder >= 18)
        {
            System.out.println("Du er " + alder + " år.");           // blokstart
            System.out.println("Du er myndig.");                     // blokslut
        }
        else
        {
            System.out.println("Du er kun " + alder + " år.");       // blokstart
            System.out.println("Du er ikke myndig.");                // blokslut
        }
    }
}
```

---

Du er kun 15 år.  
Du er ikke myndig.

### 2.4.1 Indrykning

Læg mærke til, hvordan programkoden i blokkene i ovenstående eksempel er rykket lidt ind. Det gør det lettere for programmøren at overskue koden, så han/hun kan se, hvilken {-parentes der hører sammen med hvilken }-parentes<sup>2</sup>.

---

#### Det er god skik at bruge indrykning i en blok

#### Indrykning gør programmet meget nemmere at overskue

---

Her er det samme program uden indrykning. Det er sværere at overskue nu (man kunne måske komme til at tro, at de nederste to linjer bliver udført uafhængigt af if-sætningen):

```
public class Alder3UheldigIndrykning{
public static void main(String[] arg)
{int alder;
alder = 15;
if (alder >= 18) { System.out.println("Du er " + alder + " år.");
System.out.println("Du er myndig");
} else {
System.out.println("Du er kun " + alder + " år.");
System.out.println("Du er ikke myndig");
}}}
```

De fleste udviklingsværktøjer har funktioner til at rykke flere linjers kode ind og ud. Man gør det oftest, ved at markere teksten og trykke på Tab (tabulator-tasten til venstre for Q).

---

<sup>1</sup> De, som kender Pascal, vil genkende { som BEGIN og } som END.

<sup>2</sup> 30-50 % af de problemer, nybegyndere har med at få deres programmer til at virke, skyldes dårlig indrykning, som gør koden uoverskuelig.

## 2.5 Løkker

En løkke er en gentaget udførelse af en kommando igen og igen. Hvor mange gange løkken udføres, afhænger af et logisk udtryk.

### 2.5.1 while-løkken

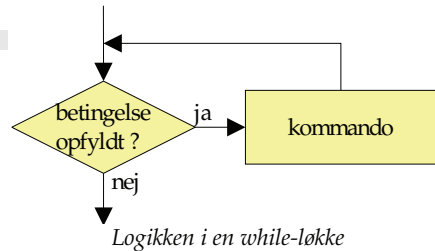
while-løkken har formen:

```
while (betingelse) kommando;
```

Kommandoen udføres igen og igen, så længe betingelsen er opfyldt. Dvs. før kommandoen udføres, undersøges betingelsen og det kontrolleres, at den er opfyldt (se rutediagrammet til højre).

Oftest vil man udføre mere end én kommando og anvender derfor en blok til samle kommandoerne:

```
while (betingelse) {  
    kommando1;  
    kommando2;  
    ...  
}
```



Her er et eksempel:

```
public class Alder4  
{  
    public static void main(String[] arg)  
    {  
        int alder;  
        alder = 15;  
  
        while (alder < 18)  
        {  
            System.out.println("Du er "+alder+" år. Vent til du bliver ældre.");  
            alder = alder + 1;  
            System.out.println("Tillykke med fødselsdagen!");  
        }  
  
        System.out.println("Nu er du "+alder+" år og myndig.");  
    }  
}
```

```
Du er 15 år. Vent til du bliver ældre.  
Tillykke med fødselsdagen!  
Du er 16 år. Vent til du bliver ældre.  
Tillykke med fødselsdagen!  
Du er 17 år. Vent til du bliver ældre.  
Tillykke med fødselsdagen!  
Nu er du 18 år og myndig.
```

Før løkken starter, har alder en startværdi på 15. Under hvert gennemløb tælles den en op. På et tidspunkt, når alder er talt op til 18, er betingelsen ikke mere opfyldt og programudførelsen fortsætter efter løkken.

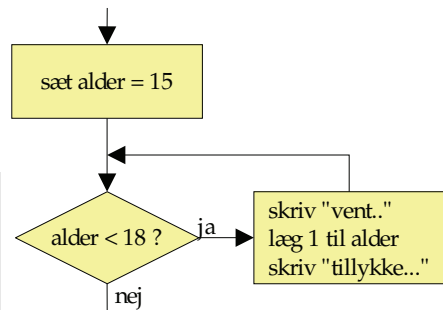
Med en løkke kan vi lave Kvadratrod-programmet nemmere: I stedet for at skrive den samme kommando igen og igen, kan vi lave en løkke.

Sammenlign det følgende program med det oprindelige Kvadratrod-program fra afsnit 2.2.5.

```
public class Kvadratrod2
{
    public static void main(String[] arg)
    {
        int n;
        n = 0;

        while (n <= 10)
        {
            System.out.println("kvadratroden af "+n+" er " + Math.sqrt(n));
            n = n + 1;
        }
    }
}
```

```
kvadratroden af 0 er 0.0
kvadratroden af 1 er 1.0
kvadratroden af 2 er 1.4142135623730951
kvadratroden af 3 er 1.7320508075688772
kvadratroden af 4 er 2.0
kvadratroden af 5 er 2.23606797749979
kvadratroden af 6 er 2.449489742783178
kvadratroden af 7 er 2.6457513110645907
kvadratroden af 8 er 2.8284271247461903
kvadratroden af 9 er 3.0
kvadratroden af 10 er 3.1622776601683795
```



Rutediagram for noget af Alder4

En tællevariabel er en variabel, der tælles op i en løkke, indtil den når en bestemt øvre grænse, hvorefter løkken afbrydes.

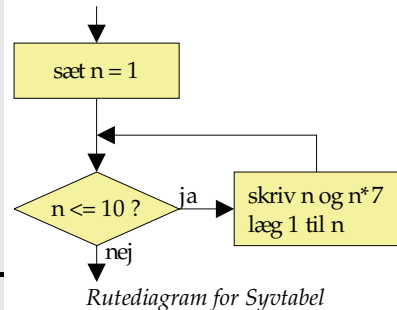
I eksemplerne ovenfor brugte vi alder og n som tællevariable.

Herunder udskriver vi 7-tabellen, ved hjælp af tællevariablen n:

```
public class Syvtabel
{
    public static void main(String[] arg)
    {
        int n;
        n = 1;

        while (n <= 10)
        {
            System.out.println(n+" : "+ 7*n);
            n = n + 1;
        }
    }
}
```

```
1 : 7
2 : 14
3 : 21
4 : 28
5 : 35
6 : 42
7 : 49
8 : 56
9 : 63
10 : 70
```



Rutediagram for Syvtabel

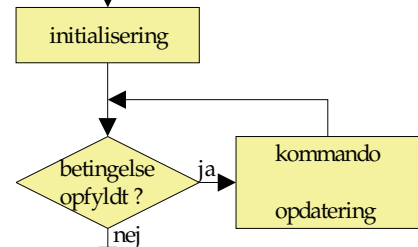
Tællevariabel-formen er den mest almindelige for løkker, men man kan sagtens komme ud for andre former for løkker. Der kan f.eks. godt indgå et regneudtryk i betingelsen.

## 2.5.2 for-løkken

for-løkken er specielt velegnet til løkker med en tællevariabel. Den har formen

```
for (initialisering; betingelse; opdatering) kommando;
```

- *initialisering* er en (evt.: erklæring og) tildeling af en tællevariabel,  
f.eks. `alder = 15`
- *betingelse* er et logisk udtryk, der angiver betingelsen for, at løkken skal fortsætte med at blive udført,  
f.eks. `alder < 18`
- *opdatering* er ændringen i tællevariablen,  
f.eks. `alder = alder + 1`

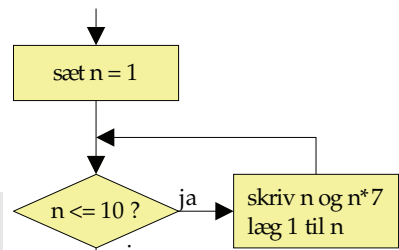


Strukturen i en for-løkke

Det kan indenad læses som "for *startværdi*, så længe *betingelse* udfør: *kommando* og *opdatering*, f.eks. "for `alder = 15`, så længe `alder < 18` udfør: *Skriv "du er.."* og *tæl alder 1 op*".

En for-løkke og en while-løkke supplerer hinanden. De har præcis samme funktion, men for-løkken er mere kompakt og bekvem, når man ønsker at lave en almindelig løkke, der udføres et bestemt antal gange. Dette program gør det samme som Syvtabel-eksemplet, men med en for-løkke:

```
public class Syvtabel2
{
    public static void main(String[] arg)
    {
        int n;
        for (n=1; n<=10; n=n+1)
            System.out.println(n+ " : "+ 7*n);
    }
}
```



Rutediagram for Syvtabel2  
(samme som for Syvtabel)

Programmerer er dovne væsner og bruger ofte for-løkken til optælling, fordi der skal skrives mindre end i en while-løkke.

Man ser også ofte, at de bruger operatoren ++ til at tælle en variabel op i en løkke: "`alder++`" svarer altså til "`alder=alder+1`", men med mindre skrivearbejde.

Tilsvarende findes --, som tæller en variabel én ned, f.eks. `alder--`.

## 2.5.3 Indlejrede løkker

En betingelse eller en løkke kan stå ethvert sted i en metode og altså også inden i en anden løkke eller en betingelse.

Herunder har vi syvtabelen igen, men denne gang "brækker" programmet sig, når det når op på 6. Efter 8 skriver det "ved ikke", i stedet for at regne resultatet ud.

```
public class Syvtabel3
{
    public static void main(String[] arg)
    {
        for (int n=1; n<=10; n++)    // n++ gør det samme som n=n+1
        {
            if (n == 6) System.out.println("puha, nu bliver det svært.");

            if (n < 8)    System.out.println(n+ " : "+ 7*n);
            else System.out.println(n+ " : (ved ikke)");
        }
    }
}
1 : 7
2 : 14
3 : 21
4 : 28
5 : 35
puha, nu bliver det svært.
6 : 42
7 : 49
8 : (ved ikke)
9 : (ved ikke)
10 : (ved ikke)
```

Man kan lave løkker i løkker. Herunder udregner vi  $n \cdot 7$  ved at lægge 7 sammen n gange:

```
public class Syvtabel4
{
    public static void main(String[] arg)
    {
        for (int n=1; n<=10; n=n+1)
        {
            int sum = 0;
            for (int j=0; j<n; j++) sum = sum + 7;

            System.out.println(n+ " : "+ sum);
        }
    }
}
1 : 7
2 : 14
3 : 21
4 : 28
5 : 35
6 : 42
7 : 49
8 : 56
9 : 63
10 : 70
```

## 2.5.4 Uendelige løkker

Hvis programmøren ikke er omhyggelig, kan han komme til at lave en løkke, hvor betingelsen vedbliver at være sand. Så bliver programudførelsen i løkken i al evighed (eller indtil brugeren afbryder programmet).

Lad os f.eks. sige, at programmøren er kommet til at skrive '-' i stedet for '+' i opdateringen af `n` i `while`-løkken fra Syvtabel-programmet. Nu vil computeren tælle nedad:

```
public class SyvtabelFejl
{
    public static void main(String[] arg)
    {
        for (int n=1; n<=10; n=n-1)
            System.out.println(n+ " : "+ 7*n);
    }
}
```

---

```
1 : 7
0 : 0
-1 : -7
-2 : -14
-3 : -21
-4 : -28
```

... og så videre i det uendelige. Løkken vil aldrig stoppe, fordi `n` vedbliver at være mindre end<sup>1</sup> 10.

En anden faldgrube er, at komme til at sætte et semikolon efter en `while`-løkke:

```
while (n <= 10);
```

Oversætteren vil tro, at der ikke er nogen kommando, der skal udføres og blot undersøge betingelsen igen og igen og igen og igen... Da `n` ikke ændrer sig, vil programmet aldrig stoppe.

Det er programmørens ansvar at sikre, at betingelsen i en løkke på et tidspunkt ikke mere opfyldes, så programmet ikke går i uendelig løkke<sup>2</sup>.

## 2.5.5 Opgaver

Prøv at køre hvert af de foregående eksempler og forvis dig om, at du forstår dem.

Mange værktøjer understøtter trinvis gennemgang til fejlfinding (eng.: debugging). Prøv i dit udviklingsværktøj og hold øje med variablerne (gøres nok med F8-tasten "step over").

- 1) Lav et program, der tæller nedad fra 10 til 1.
- 2) Lav et program, der udregner værdien af  $1+2+3+ \dots +20$  med en løkke.
- 3) Lav et program, der udskriver 2-tabellen, 3-tabellen, .. op til 10-tabellen.
- 4) Skriv et program, som for ligningen  $y=3*x+6*x+9$  udskriver værdierne af  $y$  for  $x=0$ ,  $x=1$ ,  $x=2$ ,  $x=3$  ...  $x=10$ . Ret det derefter til at skrive ud for  $x=0$ ,  $x=10$ ,  $x=20$ ,  $x=30$ ... $x=100$ .
- 5) Lav spillet "Gæt hvilket tal jeg tænker på": Lav et program, der husker et tal fra 1 til 20, som brugeren skal gætte. Her er et forslag til dialogen med brugeren:

```
Gæt et tal: 8
Tallet jeg tænker på er højere.
Gæt et tal: 13
Tallet jeg tænker på er lavere.
Gæt et tal: 11
Det er det rigtige tal! Du brugte 3 forsøg.
```

Vink: Et tilfældigt tal mellem 1 og 20 kan fås med `(int) (Math.random()*20 + 1)`

- 
- 1 På et tidspunkt når `n` til -2147483648. Variablens 32-bit kapacitet vil være opbrugt og `n` vil faktisk skifte til 2147483647, som er større end 10, hvorved løkken vil stoppe, men der er næppe nogen, der gider vente så længe!
  - 2 hvilket er noget ganske andet end uendelig lykke!

## 2.6 Værditypekonvertering

I Java har alle variabler en bestemt type gennem hele deres levetid. Det sætter nogle begrænsninger for, hvilke værdier man kan tildele en variabel. Når man først har vænnet sig til det, er det en stor hjælp, fordi oversætteren på denne måde ofte fanger fejl i programmerne (desuden gør det, at computeren hurtigere kan udføre beregninger).

I Java kan man f.eks. ikke lægge en double-værdi ind i en int-variabel:

```
int x;  
x = 2.7; // Oversætterfejl: Possible loss of precision: double, required: int.
```

Forsøger man, vil man få en oversætter-fejl. Årsagen til, at vi i Java ikke kan gemme 2.7 i `x`, kan forstås på to måder, der begge er rigtige og gyldige.

- 1) `x` har kun plads i lageret til at gemme hele tal.
- 2) `x` er erklæret som en int og skal derfor blive ved med at være en int. I de efterfølgende beregninger kan det have stor betydning, om `x` har en kommadel. Programmøren skal derfor kunne se på, hvordan `x` er erklæret og derefter være helt sikker på, hvilke værdier `x` kan indeholde.

For at kunne gemme 2.7 i `x` bliver man derfor nødt til at lave 2.7 om til en int-værdi. Det kaldes at typekonvertere værdien (eng.: cast). Dette er ikke helt uden problemer:

- Der er åbenlyst et informationstab, da kommadelen af værdien må fjernes.
- Derudover kunne double-værdien være et meget stort tal (f.eks. 5 mia. i stedet for 2.7), som ikke kan rummes i en int (der kun kan rumme tal fra -2 mia. til +2 mia).
- Konverteringen kan foretages på flere måder. Skal man afrunde korrekt op til 3 eller nedrunde til 2?

Af disse årsager bliver man i nogle tilfælde nødt til eksplicit at fortælle oversætteren, at den skal foretage en konvertering af værdien til en anden type.

### 2.6.1 Eksplicit typekonvertering

Man konverterer en værdi til en anden type, ved at skrive det eksplicit med:

```
int x;  
x = (int) 2.7;
```

Inde i parentesen skriver man typen, som værdien lige til højre skal konverteres til.

Denne form for typekonvertering runder altid ned<sup>1</sup> til nærmeste hele tal<sup>2</sup>.

---

1 Med metoden `Math.round(x)` kan man få den normale afrunding, hvor 3.5 rundes op til 4 og 3.4999 rundes ned til 3.

2 Man skal være opmærksom på, at hvis det konverteres fra et meget stort tal, kan det være, at den nye type ikke kan repræsentere tallet. F.eks. vil konvertering fra 10000000000000.0 til int ikke give det forventede, da det største tal, int kan rumme, er 2147483647.

Ved typekonverteringen fra int til float mistes også noget præcision (de mindst betydende cifre).



## 2.6.2 Implicit typekonvertering

Implicit (værdi-)typekonvertering betyder, at oversætteren selv laver konverteringen, uden at programmøren behøver at skrive noget særligt om, at den skal gøre det.

```
double y;  
y = 4; // OK: Implicit værdi-typekonvertering.
```

Selvom 4 er en int-værdi, kan *y* godt indeholde den, da den svarer til double-værdien 4.0. Denne form for konvertering er således ikke nær så problematisk, som i det tidligere eksempel.

En tommelfingerregel i Java er, at når modtagertypen kan indeholde hele intervallet af mulige værdier for afsendertypen, kan den være implicit. I appendiks 2.11.2 sidst i dette kapitel findes en tabel over typerne.

## 2.6.3 Misforståelser omkring typekonvertering

Bemærk, at det kun er *værdien*, der bliver konverteret. Variablen bliver ikke ændret.

```
int x;  
double y;  
y = 2.7;  
x = (int) y; // punkt A  
System.out.println(x);  
System.out.println(y); // y er upåvirket af typekonverteringen
```

---

```
2  
2.7
```

Man kunne måske fristes til at tro, at i punkt A konverteres variabelen *y* til en variabel af typen *int*, men det ville så betyde, at den sidste linje i uddata skulle være 2. Men husk at:

---

**En variabels type er altid, som den er erklæret – den kan ikke ændre type**

---

Det, der sker i ovenstående, er, at *y*'s værdi (2.7) læses, en konverteret værdi (2) beregnes og denne værdi lægges ind i *x*.

En anden misforståelse er at tro, at oversætteren kan se, at noget er lovligt ud fra de øvrige programlinjer, f.eks.:

```
int x;  
double y;  
y = 4.0;  
x = y; // Fejl - her stopper oversætteren med "Possible loss of precision"
```

I ovenstående tilfælde kunne man tro, at man kan bruge implicit typekonvertering, fordi oversætteren kan se, at *y* altid er 4.0 og at der derfor ikke går information tabt. Men så klog er oversætteren ikke. Når den skal afgøre, om den kan lave implicit typekonvertering, kigger den *kun* på typerne af variabler og værdier. Den skeler ikke til resten af programmet<sup>1</sup>.

---

1 Den ser ikke engang, om en konstant værdi uproblematisk kan konverteres: `int x; x=4.0; //Fejl`

## 2.7 Fejl

Som sagt udfører computeren programmet instruktion for instruktion som en kokebogsopskrift. Computeren forstår ikke programmet, men udfører blot det, programmøren (kokebogsforfatteren) har skrevet.

### 2.7.1 Indholdsmæssige (logiske) fejl

Da maskinen ikke forstår programmet, kan den heller ikke rette op på fejlene i programmørenes opskrift eller forstå, hvad programmøren "mener" med det, han skrev. Man kan altså sagtens komme til at lave et program, der gør noget andet end det, programmøren har tilsigtet:

```
public class ProgramMedFejl
{
    public static void main (String[] arg)
    {
        System.out.println("Hej Verdne!");
        int sum = 2 - 2;
        System.out.println("2 og 2 er "+sum);
    }
}
```

```
Hej Verdne!
2 og 2 er 0
```

Dette eksempel har en stavfejl og en forkert udregning. I afsnit 2.5.4 om uendelige løkker så vi en anden fejl, der gjorde, at programmet aldrig stoppede. Et andet eksempel kunne være et skatteprogram, der glemmer at tage højde for bundfradraget.

### 2.7.2 Sproglige fejl

Mens computeren ikke har mulighed for, at finde indholdsmæssige fejl i programmerne, kan den godt finde sproglige og syntaksmæssige problemer, dvs. hvis kildekoden gør brug af ukendte variabler eller metoder, eller ikke er gyldig i forhold til sprogets syntaks (den formelle definition af, hvordan man skriver javakode).

Hvis der er sproglige fejl i kildekoden, kan den ikke oversættes til bytekode, så man kan altså overhovedet ikke komme til at prøve sit program. De følgende instruktioner er alle forkerte og vil blive fanget under oversættelsen af programmet. Ofte kan fejlmeddelelsen overraske lidt, men med lidt øvelse kan man lære at forstå den "firkantede" måde, som computeren "tænker" på:

```
System.out.println("Hej verden!);
```

Her mangler en slut-" til at markere, hvor strengen stopper. Oversætteren skriver *unclosed character literal*. Den kan ikke regne ud, at strengen slutter lige før ')

```
System.out.pintln("Hej verden!");
```

Kaldet til `println` er stavet forkert. Oversætteren skriver *method pintln(java.lang.String) not found in class java.io.PrintStream*. Den kan ikke finde ud af, at man mener `println` (med r) i stedet for `pintln`.

```
system.out.println("Hej verden!");
```

System er stavet forkert (med småt). Oversætteren skriver *cannot access class system.out; neither class nor source found for system.out*. Den skelner mellem store og små bogstaver og kan ikke se, at man mener System (med stort) i stedet for system.

```
System.out.println(Hej verden!);
```

Der mangler " til at markere, hvor strengen starter og slutter. Oversætteren skriver *' )' expected* og peger lige efter Hej. Den forstår ikke at "Hej verden!" er en tekststreng, når "-tegnene mangler og mener derfor, at 'Hej' og 'verden!' skal behandles adskilt.

Når man skal finde en fejl, gælder det om at nærlæse fejlmeddelelsen og programkoden omkring stedet, hvor fejlen er og at huske, at computeren følger faste regler, men ikke forstår, hvad der foregår. F.eks. er den sidste fejlmeddelelse *' )' expected* ikke særlig sigende, da fejlen formentlig er, at der mangler "-tegn.

Det kan være banaliteter, der er årsag til sprogfejl. Det giver ofte anledning til sprogfejl, at folk glemmer, at der er forskel på store og små bogstaver.

---

### Java skelner altid mellem store og små bogstaver

Det er god stil konsekvent at skrive klassenavne med stort og variabler og metoder med småt

---

## 2.7.3 Køretidsfejl

Visse fejl opstår først ved udførelsen af programmet. Selvom alting er syntaktisk korrekt, opstår der alligevel en undtagelse fra den normale programudførelse.

Herunder ses et program, der stopper på grund af division med 0.

```
public class ProgramMedFejl2
{
    public static void main (String[] arg)
    {
        int a,b,c;

        a = 5;
        b = 6;

        c = b/(a-5);
        System.out.println("c = "+c);
    }
}
```

---

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ProgramMedFejl2.main(ProgramMedFejl2.java:10)
```

Køretidsfejl forårsager, at der opstår en undtagelse (eng.: exception), som, hvis den ikke håndteres, stopper programudførelsen (populært: programmet 'går ned').

Dette vil blive behandlet grundigere i kapitel 14 om undtagelser.

## 2.8 Test dig selv

Hvis du kan svare på spørgsmålene herunder, kan du være rimelig sikker på, at du har forstået alt det væsentlige i dette kapitel. Brug papir og blyant og skriv et par linjers svar for hvert spørgsmål. I næste afsnit kan du finde nogle vejledende svar.

- 1) Hvad er kommentarer? Hvordan skrives de?
- 2) Hvordan kommer man fra at skrive et program til at køre det?
- 3) Hvis en klasse (et program) hedder Xyz, hvad skal kildetekstfilen så hedde?
- 4) Hvad er en variabel?
- 5) Hvad bruges dens navn til?
- 6) Hvad bruges dens type til?
- 7) Beskriv nogle variabeltyper.
- 8) Hvordan giver man en variabel en værdi?
- 9) Hvad med typekonvertering?
- 10) Hvordan skrives en if-else-sætning (betinget udførelse)?
- 11) Hvad gør { og }?
- 12) Hvad er en løkke? Hvordan laver man en? Giv et eksempel.
- 13) Hvilke slags fejl findes der?

## 2.9 Resumé

I slutningen af de fleste kapitler kommer et resumé. Hvis du forstår resuméet, har du forstået alt det væsentlige i dette kapitel (men det er også nødvendigt at programmere selv).

- 1) Kommentarer er tekst, som forklarer mennesker, hvad der sker i programmet. De påvirker ikke programudførelsen. De skrives med `//` eller med `/*` og `*/`
- 2) Programmer skrives som kildetekst. Før programmet kan udføres, skal det oversættes til binær kode (kaldet bytekode). En javafortolker udfører den binære kode.
- 3) Hvis klassedefinitionen hedder `Xyz`, ligger kildeteksten i filen `Xyz.java`
- 4) En variabel bruges til at gemme en værdi i. En variabel har et navn og en type. Variabler skal erklæres, inden de kan benyttes. Man erklærer derfor ofte variabler i toppen af programmet.
- 5) Variablens navn bruges til at identificere variabelen i vores program.
- 6) Variablens type bestemmer, hvilke værdier variabelen kan indeholde.
- 7) En variabel kan f.eks. være af typen heltal (`int`), kommatotal (`double`) eller sand/falsk (`boolean`).
- 8) En variabel kan tildeles en værdi ved at bruge operatoren `=`. En variabels værdi kan også ændres ved at bruge `=`. På højre side af `=` kan f.eks. stå et beregningsudtryk.
- 9) Konvertering fra en type til en anden sker uden videre, hvis den nye type kan indeholde alle den gamle types værdier. Konvertering, hvor der muligvis mistes vigtig information, skal ske eksplicit ved at skrive den nye type i parentes, f.eks. (`int`) 5.7. Der rundes altid nedad.
- 10) Betinget udførelse har formen: `if (betingelse) { ... } else { ... }`
- 11) `{` og `}` samler flere kommandoer i en blok.
- 12) Løkker er dele af programmet, der gentages igen og igen, indtil en betingelse ikke mere er opfyldt, f.eks. (`i < 10`). De kan laves med `while` eller `for`.
- 13) Der findes tre slags fejl: De, som oversætteren kan finde (sprogfejl), de, som opstår under kørslen og forhindrer programmet i at fortsætte (køretidsfejl) og indholdsmæssige fejl, som kun ses af et menneske med sund fornuft (logiske fejl).

### 2.9.1 Gode råd om programmering

- Gennemtænk problemstillingen, inden du sætter dig til computeren.
- Formulér problemet eller formålet med programmet.
- Overvej mulige løsningsstrategier. De fleste problemer kan løses på mere end én måde.
- Lav en skitse til programmet i pseudokode (på papiret med danske ord). Du kan også tegne flowdiagrammer (der beskriver rækkefølgen tingene sker i).
- Det er ikke altid, man kan tænke hele programmet igennem på forhånd. Ved mere komplicerede programmer må man skifte mellem kodning og refleksion over koden.
- Når du sidder ved computeren, så skriv ganske få linjer ad gangen og afprøv dem. På den måde er det lettere at se problemet, hvis (når!) programmet ikke virker.
- Brug `System.out.println(...)` flittigt til at se, om programmet gør som forventet.
- Lav konsekvent indrykning i dine programmer (se afsnit 2.4.1).

## 2.10 Opgaver

Oversæt og kød et af dine egne programmer fra kommandolinjen (se afsnit 2.1.3).

### 2.10.1 Befordringsfradrag

Lav et program, som udregner befordringsfradraget (det, der kan trækkes fra i skat ud fra, hvor langt der er mellem arbejde og hjem).

- 1) Udregn og udskriv fradraget pr. dag fra 25 til 75 km på hver sin linje.
- 2) Udregn og udskriv fradraget pr. dag fra 25 til 150 km på hver sin linje.
- 3) Udregn og udskriv fradraget pr. dag fra 10 til 150 km på hver sin linje. Kun hver 10. km udskrives (10km, 20km, 30km...). Reglerne for fradraget for år 2000 var følgende:

første 24 km	intet fradrag
25 - 100 km	154 øre pr. km
over 100 km	77 øre pr. km

### 2.10.2 Kurveprogram

Denne opgave benytter begreber, som man lærer i f.eks. den matematiske gren af gymnasiet. Kender du ikke i forvejen disse begreber, bør du springe opgaven over, de er ikke nødvendige for at lære at programmere.

- 1) Skriv et program, der udskriver grafen over kvadratrodfunktionen (`Math.sqrt()`).  
Vink: Når du vil skrive en "\*" uden linjeskift, kan du bruge `System.out.print("*")` (dvs. uden `'\n'`). Når du vil skifte linje, kan du bruge `System.out.println()` uden parametre.
- 2) Lav kurveprogrammet om, så det i stedet viser kurven over polynomiet  $0.2x^2 + 0.5x + 2$ . Lav programmet, så det er nemt at se, hvor man skal rette, for at ændre funktionen, intervalstart, intervalslut, skalering og forskydning af y-aksen. Dvs. lav det til variabler og brug kommentarer, til at markere stederne i programmet.
- 3) Lav om på kurvetegningsprogrammet, så kurven ikke er udfyldt, men tegnes som en streg.
- 4) Eventuelt: Udvid kurveprogrammet til at udregne det totale antal af stjerner, der skrives ud (udregn integralet af funktionen numerisk ved at summere arealet under grafen). Er det nemmest at gøre løbende, mens stjernerne tegnes, eller bagefter? Hvordan ville du gøre på den ene og på den anden måde?

## 2.11 Appendiks

Dette afsnit sætter det, du har lært i kapitlet, i system og kan senere bruges som opslagsværk. Enkelte steder står der noget, som ikke er gennemgået endnu, men som er med for helhedens skyld.

### 2.11.1 Navngivningsregler

Variabler og metoder bør have lille startbogstav.

Eksempler: `n`, `alder`, `tal`, `talDerSkalUndersøges`, `main()`, `println()`, `sqrt()`.

Klasser bør have stort startbogstav.

Eksempler: `HejVerden`, `Cylinderberegning`, `Syvtabel2`

Består navnet af flere ord, stryger man normalt mellemrummene og lader hvert af de efterfølgende ord starte med stort (nogen bruger også understreg `_` som mellemrum).

- Navnet kan bestå af A-Å, a-å, 0-9, \$ og `_`
- Det må ikke starte med et tal. Det kan have en vilkårlig længde.
- Lovlige navne: `peter`, `Peter`, `$antal`, `var2`, `J2EE`, `dette_er_en_test`
- Ulovlige navne: `7eleven`, `dette-er-en-test`, `peter#`

Da visse styresystemer endnu ikke understøtter æ, ø og å i filnavne, bør man undgå disse i klassenavne.

### 2.11.2 De simple typer

Her er en oversigt over alle de simple variabeltyper i Java.

Type	Art	Antal bit	Mulige værdier	Standardværdi
byte	heltal	8	-128 til 127	0
short	heltal	16	-32768 til 32767	0
int	heltal	32	-2147483648 til 2147483647	0
long	heltal	64	-9223372036854775808 til 9223372036854775807	0
float	kommatal	32	$\pm 1.40239846\text{E}-45$ til $\pm 3.40282347\text{E}+38$	0.0
double	kommatal	64	$\pm 4.94065645841246544\text{E}-324$ til $\pm 1.79769313486231570\text{E}+308$	0.0
char	unicode	16	\u0000 til \uffff (0 til 65535)	\u0000
boolean	logisk	1	true og false	false

De vigtigste er `int`, `double` og `boolean`. I enkelte tilfælde bliver `long` og `char` også brugt, mens `byte`, `short` og `float` meget sjældent bruges.

### 2.11.3 Værditypekonvertering

Konvertering til en anden type sker automatisk (implicit) i tilfælde, hvor der ikke mistes information (forstået på den måde, at intervallet af de mulige værdier udvides), dvs.

- fra `byte` til `short`, `int`, `long`, `float` eller `double`
- fra `short` til `int`, `long`, `float` eller `double`
- fra `int` til `long`, `float` eller `double`
- fra `long` til `float` eller `double`
- fra `float` til `double`.

Den anden vej, dvs. hvor der muligvis mistes information, fordi intervallet af mulige værdier indsnævres, skal man skrive en eksplicit typekonvertering.

Det gøres ved at skrive en parentes med typenavnet foran det, der skal konverteres:

```
int x;  
double y;  
y = 3.8;  
x = (int) y
```

Her skæres kommadelen af 3.8 væk og x får værdien 3.

Eksplicit typekonvertering sikrer, at programmøren er bevidst om informationstab (glemmes det, kommer oversætteren med fejlen: *possible loss of precision: double, required: int*).

Det skal ske

- fra double til float, long, int, short, char eller byte
- fra float til long, int, short, char eller byte
- fra long til int, short, char eller byte
- fra int til short, char eller byte
- fra short til char eller byte
- fra byte til char
- fra char til short eller byte.

Der kan ikke typekonverteres til eller fra boolean.

## 2.11.4 Aritmetiske operatorer

Operator	Brug	Forklaring
+	a + b	a lagt sammen med b
-	a - b	b trukket fra a
*	a * b	a gange b
/	a / b	a divideret med b
%	a % b	rest fra heltalsdivision af a med b
-	-a	den negative værdi af a
++	a++	a = a+1; værdi før optælling
++	++a	a = a+1; værdi efter optælling
--	a--	a = a-1; værdi før nedtælling
--	--a	a = a-1; værdi efter nedtælling

Operatorerne giver altid samme type som operanderne, der indgår. Det skal man være specielt opmærksom på for / (divisions) vedkommende, hvor resten mistes ved heltalsdivision (f.eks er  $4/5 = 0$ ). Resten kan findes med % (f.eks er  $4\%5 = 4$ , mens  $6\%5 = 1$ ). Ønskes division med kommatall, skal mindst en af operanderne være et kommatall (f.eks er  $4.0/5 = 0.8$ ).

Operatoren ++ tæller en variabel op med én : a++ svarer til a=a+1. Tilsvarende er a-- det samme som a=a-1.



## 2.11.5 Regning med logiske udtryk

u1 og u2 er to logiske udtryk eller logiske variabler

Operator	Brug	Forklaring
&&	u1 && u2	både u1 og u2 er sandt
	u1    u2	u1 eller u2 er sandt
!	! u1	negation af u1

**Operator &&** udtrykker, at både 1. og 2. udtryk skal være sandt:

1. udtryk	2. udtryk	1. udtryk && 2. udtryk
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

F.eks. er udsagnet ( $a > 5$  &&  $a < 10$ ) sandt, hvis a er større end 5 og a er mindre end 10.

**Operator ||** udtrykker, at 1. eller 2. udtryk skal være sandt.

1. udtryk	2. udtryk	1. udtryk    2. udtryk
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

F.eks. er udsagnet ( $a > 5$  ||  $a == 0$ ) sandt, hvis a er større end 5, eller a er 0.

**Operator !** Udtrykker, at udtrykket skal **negeres**, dvs. at (!u1) er sandt, hvis u1 er falsk og falsk, hvis u1 er sandt, f.eks. er udsagnet (!( $a > 5$ )) sandt, hvis der ikke gælder, at a er større end 5 (det er det samme som ( $a \leq 5$ )).

I visse andre programmeringssprog skrives AND for &&, OR for || og NOT for !

## 2.11.6 Sammenligningsoperatorer

Operator	Brug	Forklaring
>	$a > b$	a større end b
>=	$a \geq b$	a større end el. lig med b
<	$a < b$	a mindre end b
<=	$a \leq b$	a mindre end el. lig med b
==	$a == b$	a er lig med (identisk med) b
!=	$a != b$	a forskellig fra b

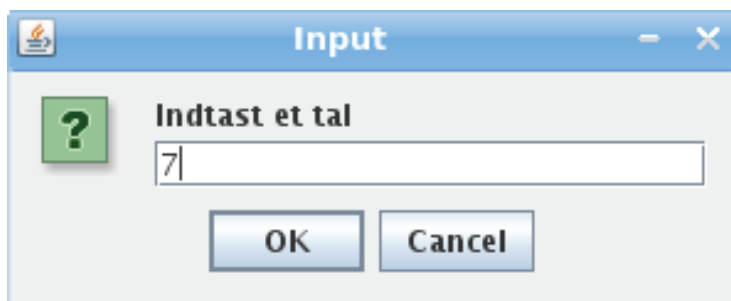
## 2.12 Avanceret

Mange kapitler afsluttes af et afsnit med lidt sværere stof, der ikke forudsættes i resten af bogen, men som kan være nyttigt at kende til senere. Er du begynder i programmering, anbefaler vi, at du lader stoffet "falde til ro" og venter et par uger med at læse denne del.

I modsætning til den almindelige tekst, forudsætter de avancerede afsnit ofte noget, der først bliver forklaret *senere* i bogen. Du kan derfor være nødt til at vende tilbage til disse afsnit, for at få det fulde udbytte.

### 2.12.1 Grafiske indtastningsvinduer

I stedet for `java.util.Scanner` kan du bruge `javax.swing.JOptionPane.showInputDialog()` til at få et grafisk indtastningsvindue, som brugeren kan udfylde:



Det gøres således:

```
public class BenytIndtastningsvindue
{
    public static void main(String[] arg)
    {
        String str = javax.swing.JOptionPane.showInputDialog("Indtast et tal");
        int svar = Integer.parseInt(str);

        System.out.println("Du indtastede "+svar+", det dobbelte er: " + 2*svar);

        str = javax.swing.JOptionPane.showInputDialog("Indtast et kommatal");
        double svar2 = Double.parseDouble(str);

        System.out.println("Du indtastede "+svar+", det halve er: " + 0.5*svar2);
    }
}
```

---

Du indtastede 7, det dobbelte er: 14

Du indtastede 12.4, det halve er: 6.2

Metoden `showInputDialog()` åbner vinduet og venter indtil brugeren har trykket på 'OK' eller 'Cancel'.

Trykkede brugeren på 'OK' returnerer den en streng med svaret. Denne streng laves om til et tal med kaldet `Integer.parseInt()`. Har du brug for at læse et kommatal skal du i stedet kalde `Double.parseDouble()`.

Trykkede brugeren på 'Cancel' returnerer `showInputDialog()` værdien `null`, som får programmet til at stoppe i den efterfølgende linje.

## 2.12.2 Formaterer tal og datoer

Vil man at skrive noget ud pænt i kolonner, er det nemmest at bruge `System.out.printf()`. I denne metode angiver man en formateringsstreng og derefter de data, der skal formateres:

```
import java.util.*;

public class UdskrivFormateret
{
    public static void main(String[] args)
    {
        for (int n = 1; n < 5; n++)
            System.out.printf("kvadratrod( %d ) er %5.3f / %2$e\n", n, Math.sqrt(n));

        System.out.printf(Locale.US, "kvadratrod %0+3d er %.9f\n", 8, Math.sqrt(8));
        Date d = new Date(); // repræsenterer dags dato
        System.out.printf("I dag er %1$tA den %1$te. %1$tB år %1$tY kl. %1$tT\n", d);
    }
}

kvadratrod( 1 ) er 1,000 / 1.000000e+00
kvadratrod( 2 ) er 1,414 / 1.414214e+00
kvadratrod( 3 ) er 1,732 / 1.732051e+00
kvadratrod( 4 ) er 2,000 / 2.000000e+00
kvadratrod +08 er 2,828427125
I dag er tirsdag den 25. september år 2007 kl. 14:42:18
```

Formateringsstrengen har nogle formatspecificeringer med syntaksen (ting i [ ] er valfrie):

`%[argumentindeks$][flag][bredde][.præcision]konverteringstype`

De mest almindelige konverteringstyper for tal er:

- `%d` decimalt heltal
- `%f` flydende kommatal
- `%e` eksponentiel notation

I første `printf`-sætning bliver `n` formateret som decimalt heltal (`%d`) og `Math.sqrt(n)` som 5-cifret flydende kommatal med 3 decimaler (`%5.3f`) og som eksponentiel notation (`%e`). I sidstnævnte har vi måttet angive argumentindekset 2 (så der står `%2$e`) for at genbruge andet argument (som er `Math.sqrt(n)`).

Man ser at den lokale (danske) formatering af tal benyttes. Ønsker man i stedet at bruge amerikansk formatering, angiver man `Locale.US` som første argument. Den anden `printf`-sætning er et eksempel på dette. Her ser man også brug af flagene `0`, for at udfylde med foranstilte `0`'er og `+` for at skrive også positive med fortegn (`%0+5d`).

I den sidste `printf`-sætning formateres et `Date`-objekt (dato og tid). Her er nogle af typerne:

- `%tA` giver dagens fulde navn (f.eks. onsdag), `%ta` giver en 3-bogstavs forkortelse
- `%tB` giver månedens fulde navn (f.eks. januar), `%tb` giver en 3-bogstavs forkortelse
- `%te` giver dagen i måneden, `%td` evt. med foranstillet `0`.
- `%tY` giver året med 4 cifre, `%ty` med 2 cifre
- `%tt` giver klokkelæt på dagen

## 2.12.3 Primaltal

Herunder er vist et program, der afgør, om et tal er et primaltal. Primaltal har den egenskab, at kun 1 og tallet selv går op i det. Vi kan derfor lave en løkke, der løber fra 2 til tallet, der skal undersøges (herefter kaldt `talDerSkalUndersøges`) og tjekke hvert mellemliggende tal, om det går op i `talDerSkalUndersøges`.

Hvordan ser man, om et tal går op i et andet tal? Jo, man dividerer de to tal og ser, om der er en rest med `%` (restdivisionsoperatoren, se afsnit 2.11.4).

Med andre ord, hvis

```
talDerSkalUndersøges % faktor > 0
```

er sandt, går faktor ikke op i `talDerSkalUndersøges`. Vi kan derfor lave en løkke, der løber opad, indtil vi finder et tal, der ikke giver nogen rest:

```
while (talDerSkalUndersøges % faktor > 0) faktor = faktor+1;
```

Hvis vi når helt op til `talDerSkalUndersøges`, må det være et primaltal:

```
public class Primaltal
{
    public static void main(String[] arg)
    {
        int talDerSkalUndersøges = 15;
        int faktor = 2;

        while (talDerSkalUndersøges % faktor > 0) faktor = faktor+1;

        if (faktor < talDerSkalUndersøges)
        {
            System.out.println(talDerSkalUndersøges + " er IKKE et primaltal,");
            System.out.println("for det har faktoren "+faktor);
        }
        else
            System.out.println(talDerSkalUndersøges + "er et primaltal.");
    }
}
```

---

```
15 er IKKE et primaltal,
for det har faktoren 3
```

## 2.12.4 Klassemetoder

Indtil nu har vi kun defineret `main`-metoden og haft al vores programkode der. Ofte kan det være nyttigt at opdele programkoden i mindre stumper med hvert sit navn. Disse stumper – kaldet metoder – bør indeholde kode, der logisk hører sammen, f.eks. en beregning eller opgave. Metoder bliver behandlet i kapitel 4 og 7, men lad os se på dem allerede nu.

Herunder definéer vi metoden `hils()`, der hilser på person med et bestemt navn:

```
public class Metoder1
{
    public static void hils(String navn)
    {
        System.out.println("Kære "+navn+", du aner ikke hvor glad jeg er for at");
        System.out.println("møde en med "+navn.length()+" bogstaver i sit navn!");
    }

    public static void main(String[] arg)
    {
        hils("Bo");
        hils("Jacob");
        hils("Christoffer");
    }
}
```

---

```
Kære Bo, du aner ikke hvor glad jeg er for at
møde en med 2 bogstaver i sit navn!
Kære Jacob, du aner ikke hvor glad jeg er for at
møde en med 5 bogstaver i sit navn!
Kære Christoffer, du aner ikke hvor glad jeg er for at
møde en med 11 bogstaver i sit navn!
```

En metode kan have flere kommaseparerede parametre og kan returnere én variabel. Ovenfor fik hils() parameteren navn (af type String) overført.

I nedenstående eksempel er der to metoder ud over main(): En til at gange et tal med 10 og en til at dividere et tal med et andet tal.

Metoden gangMedTi() får en variabel tal (af type int) som parameter og returnerer en int.

Metoden dividér() får tæller og nævner overført og returnerer en double med resultatet.

```
public class Metoder2
{
    //      returtype      metodenavn      (parametertype variabelnavn)
    public static int      gangMedTi      (int      tal)
    {
        int resultat;
        resultat = tal*10;           // lav mellemregning
        return resultat;           // returnér resultatet
    }

    public static double dividér (double tæller, double nævner)
    {
        return tæller / nævner; // returnér resultatet, uden en mellemregning
    }

    public static void main(String[] arg)
    {
        for( int x = 1; x<=5; x++)
        {
            int tidobbelt = gangMedTi(x);
            System.out.print( "Det tidobbelte er " + tidobbelt );

            double tredjedel = dividér(tidobbelt, 3);
            System.out.println( " og en tredjedel af dette er " + tredjedel );
        }
    }
}
```

```
Det tidobbelte er 10 og en tredjedel af dette er 3.3333333333333335
Det tidobbelte er 20 og en tredjedel af dette er 6.666666666666667
Det tidobbelte er 30 og en tredjedel af dette er 10.0
Det tidobbelte er 40 og en tredjedel af dette er 13.333333333333334
Det tidobbelte er 50 og en tredjedel af dette er 16.666666666666668
```

En metode, der returnerer en værdi, skal indeholde nøgleordet "return", som får programudførslen til at hoppe tilbage til kalderen (i dette tilfælde main()) med resultatet. Hvis en metode ikke skal returnere noget resultat, erklæres den med returtype "void".

Man kan godt undlade at benytte returværdien fra en metode (så vil resultatet bare blive smidt væk). Følgende definition af main() gør ingenting:

```
public static void main(String[] arg)
{
    for( int x = 1; x<=10; x++)
    {
        gangMedTi(x);
    }
}
```

## Eksempel: Finde primtal

Herunder har vi primtalsprogrammet igen, men denne gang undersøger vi primtal i intervallet 10 til 20. Hver gang vi finder et primtal, tæller vi antalPrimtal op, så vi til sidst kan skrive præcist, hvor mange primtal, der er i intervallet.

```
public class Metoder3Primtal
{
    public static boolean erPrimtal(int talDerSkalUndersøges)
    {
        int faktor = 2;

        while (talDerSkalUndersøges % faktor > 0) faktor++;

        if (faktor < talDerSkalUndersøges)
        {
            System.out.println(talDerSkalUndersøges + " har faktoren "+faktor);
            return false;
        } else {
            System.out.println(talDerSkalUndersøges + " er et primtal.");
            return true;
        }
    }

    public static void main(String[] arg)
    {
        int antalPrimtal = 0;

        int tal;
        for (tal = 10; tal<20; tal++)
        {
            boolean erPrim = erPrimtal(tal);
            if ( erPrim ) antalPrimtal = antalPrimtal + 1;
        }

        System.out.println("Antal primtal i alt: " + antalPrimtal);
    }
}
```

---

```
10 har faktoren 2
11 er et primtal.
12 har faktoren 2
13 er et primtal.
14 har faktoren 2
15 har faktoren 3
16 har faktoren 2
17 er et primtal.
18 har faktoren 2
19 er et primtal.
Antal primtal i alt: 4
```

## 2.12.5 Kombination af logiske operatorer

Som beskrevet i appendiks 2.11.5 kan flere logiske udtryk kombineres til ét udtryk med && (og), || (eller) og ! (ikke). Linjen

```
if (0<=n && n<=100 && n%2==0) System.out.println("n er OK");
```

vil udskrive "n er OK", hvis n er lige og mellem 0 og 100.

Man negerer et udtryk med ! foran:

```
if (!(0<=n && n<=100 && n%2==0)) System.out.println("n er ikke OK");
```

Man kan lege en del med logisk aritmetik. Her er et par måder at skrive det *samme* udtryk:

```
if (!(0<=n && n<=100 && n%2==0)) ...

if (!(0<=n) || !(n<=100) || !(n%2==0)) ...

if ( 0>n || n>100 || n%2!=0) ...
```

## 2.12.6 Alternativudtryk

Med `?` kan man angive to alternativer i et udtryk. For eksempel vil `n` efter linjen

```
n = k>100 ? 200 : k*2
```

være 200, hvis `k` var større end 100 og ellers være det dobbelte af `k`.

Følgende kode skriver enten "n er lige" eller "n er ulige":

```
System.out.println("n er "+ (n%2==0 ? "lige" : "ulige") );
```

Alternativudtryk kan kombineres til mere end to muligheder:

```
System.out.println("n er "+ (n<0? "negativ" : n>0 ? "positiv" : "nul"));
```

## 2.12.7 Brug af `++` og `--` (optælling og nedtælling)

I stedet for at skrive

```
n = n + 1;
```

kan man også tælle en variabel en op (eng.: increment) med

```
n++;
```

eller

```
++n;
```

På samme måde med nedtælling (eng.: decrement): `n--` og `--n` svarer til `n=n-1`.

Optællings- og nedtællingsoperatorene er især nyttige i løkker. Koden

```
n=0;
while (n++ < 9) System.out.print(n);
```

skriver 123456789 ud og værdien af `n` er bagefter 10.

Forskellen på `n++` og `++n` viser sig, når de bruges i et udtryk. `n++` giver variabelens værdi *før* optælling, mens `++n` giver værdien *efter* optællingen. Koden

```
n=0;
while (++n < 9) System.out.print(n);
```

skriver således 12345678 ud og værdien af `n` er bagefter 9.

## 2.12.8 Brug af `+=` og `-=`

I stedet for at skrive

```
n = n + 10;
```

kan man også skrive

```
n += 10;
```

Det samme gælder de andre regneudtryk, f.eks. `-=` og `*=`.

## 2.12.9 do-while-løkken

Mens `while`-løkken først tester betingelsen og derpå udfører kommandoerne:

```
while (betingelse)
{
    kommando;
}
```

forholder det sig anderledes med `do-while()`-løkken. Her udføres kommandoen først én gang og derefter bruges betingelsen til at afgøre, om den skal udføres igen:

```
do
{
    kommando;
}
while (betingelse);
```

Her er Syvtabel skrevet om til at bruge en do-while()-løkke:

```
public class SyvtabelDoWhile
{
    public static void main(String[] arg)
    {
        int n = 1;

        do
        {
            System.out.println(n+ " : " + 7*n);
            n = n + 1;
        }
        while (n < 11);
    }
}
```

---

```
1 : 7
2 : 14
3 : 21
4 : 28
5 : 35
6 : 42
7 : 49
8 : 56
9 : 63
10 : 70
```

## 2.12.10 break

break bryder ud af en løkke (er der flere, er det den inderste) og afslutter den. Eksempel:

```
for (int n=0; n<4; n++) {
    System.out.print(" A"+n);
    if (n == 2) break;
    System.out.print(" B"+n);
}
```

---

A0 B0 A1 B1 A2

## 2.12.11 continue

continue afbryder dette gennemløb af løkken og går til næste gennemløb. Eksempel:

```
for (int n=0; n<4; n++) {
    System.out.print(" A"+n);
    if (n == 2) continue;
    System.out.print(" B"+n);
}
```

---

A0 B0 A1 B1 A2 A3 B3

## 2.12.12 break og continue med navngivne løkker

En løkke kan være navngivet og da handler break eller continue om denne løkke:

```
ydre:
for (int n=0; n<6; n++) {
    System.out.print(" A"+n);
    for (int i=3; i<4; i++) {
        if (n == i) break ydre;
    }
    System.out.print(" B"+n);
}
```

---

A0 B0 A1 B1 A2 B2 A3

Her afbryder break den ydre løkke (normalt ville det være løkken med i, der blev afbrudt).



Tilsvarende med continue:

```
ydre:
for (int n=0; n<6; n++) {
    System.out.print(" A"+n);
    for (int i=3; i<4; i++) {
        if (n == i) continue ydre;
    }
    System.out.print(" B"+n);
}
```

---

A0 B0 A1 B1 A2 B2 A3 A4 B4 A5 B5[

## 2.12.13 switch

En switch kan bruges, når der skal vælges mellem en række forskellige stykker kode afhængig af et heltal eller tegn (long, int, short, byte, char).

```
int ugedag = 2;
switch (ugedag)
{
    case 1:
        System.out.print("mandag");
        break;
    case 2:
        System.out.print("tirsdag");
        break;
    case 3:
        System.out.print("onsdag");
        break;
    case 4:
    case 5:
        System.out.print("torsdag eller fredag");
        break;
    default:
        System.out.print("weekend");
}
```

Læg mærke til brugen af break, til at hoppe ud af switch-sætningen.

En switch-sætning kan altid omskrives til en række if-else-if-sætninger. Strengt kan ikke anvendes i en switch (så switch er mindre populær i Java end i så mange andre sprog).

## 2.12.14 Komma i for-løkker

Med komma kan man gruppere flere kommandoer eller udtryk i en for-løkke. Følgende

```
for (y=10,x=0; x<5; y++,x++) System.out.println(x+" "+y);
```

gør præcis det samme som

```
y=10;
for (x=0; x<5; x++)
{
    System.out.println(x+" "+y);
    y++;
}
```

---

0 10  
1 11  
2 12  
3 13  
4 14



# 3 Objekter

Indhold:

- Objekter og klasser
- Oprettelse af objekter og konstruktører
- Brug af objekters variabler og metoder
- Punkter, rektangler, strenge, lister

Kapitlet forudsættes i resten af bogen.

Forudsætter kapitel 2 Basal programmering.

## 3.1 Objekter og klasser

Hidtil har vi kun brugt de *simple typer* (som int, boolean og double). Et javaprogram vil ofte udføre mere komplekse opgaver og dermed have brug for *objekter*. Et objekt repræsenterer en eller anden (ofte fysisk) ting og indeholder sammensatte data om denne ting, f.eks. et hus-objekt (med adresse, telefonnummer, antallet af døre og vinduer ...), en bil, en person, en bankkonto, en selvangivelse, en ordre, et dokument ...

Objekter kan klassificeres i forskellige kategorier, kaldet *klasser*. F.eks. kunne man sige, at alle hus-objekter tilhører Hus-klassen. Hus-klassen er en beskrivelse af alle slags huse.

Næsten alt er repræsenteret som objekter i Java og der findes tusindvis af foruddefinerede klasser til ting, som man ofte har brug for som programmør såsom: tekststreng, datoer, lister, filer og mapper, vinduer, knapper, menuer, netværksforbindelser, hjemmeside-adresser, billeder, lyde ...

Et objekt indeholder data, der beskriver det, som objektet repræsenterer. Et Fil-objekt har oplysninger om den fil, det repræsenterer: Navn, placering, type, dato for oprettelse og indhold. Et Person-objekt har måske variabler for fornavn, efternavn, CPR-nummer.

Et objekt kan også indeholde navngivne programstumper, som kan udføres ved at give objektet besked om det. Disse programstumper kaldes *metoder* og kan opfattes som spørgsmål eller kommandoer, som man bruger til at undersøge og manipulere indholdet af objektet med.

Et Fil-objekt har måske metoden "omdøb()", der ændrer filens navn, et Person-objekt kan måske give personens alder med metoden "hvadErDinAlder()".

---

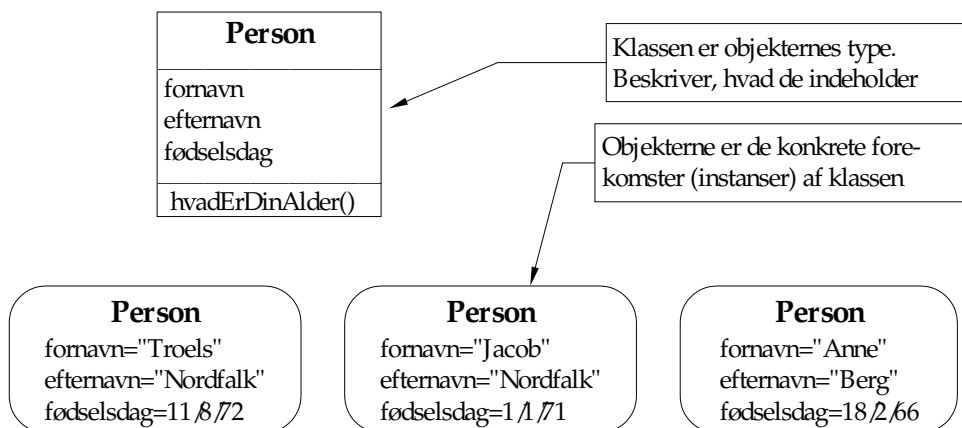
**Et objekt kan indeholde metoder og data (variabler)**

**En metode kan ændre på objektets data, når den udføres**

---

Ligesom med de simple typer afhænger det af objektets type, dets klasse, hvad man kan gøre med det. Ordet "klasse" skal forstås i betydningen "kategori, gruppe". Alle objekter kan klassificeres som værende af en bestemt type (klasse), f.eks. Streng, Dato, Fil, Knap.

Objekter af samme klasse forstår de samme beskeder (kommandoer og spørgsmål) og indeholder samme slags data. Objekter af klassen Person indeholder f.eks. begge et navn, men navnene (data) i de to person-objekter, kan være forskellige.



---

## Et objekts type kaldes dets klasse

### Klassen bestemmer, hvilke metoder og data et objekt har

---

Vi tegner klasser, som vist her til højre.

Øverst er klassens navn, dernæst data og nederst metoderne.

Dette er en del af UML-notationen (Unified Modelling Language), en notation, der ofte bliver anvendt i forbindelse med objektorienteret programmering.

I dette kapitel vil vi bruge objekter fra foruddefinerede klasser. Vi har valgt at kigge nærmere på nogle klasser, der er velegnede til at illustrere ideerne, nemlig Point, Rectangle, String, Date og ArrayList.

String og ArrayList er nok de mest brugte klasser overhovedet og er næsten uundværlige i praktisk programmering.

Klassenavn:	<b>Fil</b>
variabler:	navn placering oprettelsesdato indhold
metoder:	omdøb() slet()

## 3.2 Punkter (klassen Point)

Det første objekt, vi vil arbejde med, er Javas Point-objekt, der repræsenterer et *punkt*. I Java indeholder et punkt to heltalsvariable, nemlig en x- og en y-koordinat. Vi vil senere bruge Point-klassen, når vi kommer til programmering af grafik.

### 3.2.1 Erklæring og oprettelse

For at kunne arbejde med et objekt, har man brug for en variabel, der refererer til objektet. En variabel af typen Point (der refererer til et punkt) erklæres ved at skrive

```
Point p;
```

Ligesom med de simple typer skriver man typen (Point) efterfulgt af variabelnavnet.

Nu har vi defineret, at p er en variabel til objekter af typen Point og vi kan lave et nyt Point-objekt, som vi sætter p til at pege på<sup>1</sup>:

```
p = new Point();
```

Vi skriver altså new og klassens navn (Point) efterfulgt af parenteser med startværdier for objektet. Her giver vi ingen startværdier og parentesen er derfor tom.

---

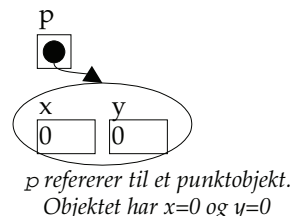
#### Et objekt oprettes med new

Når et objekt oprettes, sørger det for at sætte dets data til nogle fornuftige startværdier

---

I dette tilfælde vil punktet starte med at have koordinaterne (0,0) og situationen er, som vist på figuren til højre: p peger hen på et objekt, der har en x- og y-variabel, som begge er sat til 0.

Man kan sige, at hver gang vi anvender new-operatoren, bruger vi klassen som en slags støbeform, til at skabe et nyt objekt med.



---

<sup>1</sup> p er ikke objektet selv. p er kun en reference (pegepind, pointer) til objektet. Endnu har p ikke nogen værdi (man siger, at den er **null**, dvs. referencen peger ingen steder hen).

## 3.2.2 Objektvariabler

Vi kan undersøge objektet `p`'s variabler. Her erklærer vi en anden variabel, `a`, ...

```
int a;
```

... og gemmer `p`'s x-koordinat i variabelen:

```
a = p.x;
```

`p`'s x-koordinat får man fat i, ved at skrive *p punktum x*. Vi kan derefter udskrive `a`:

```
System.out.println("a: "+a);
```

Man kan også udskrive koordinaterne direkte, uden at bruge `a` som mellemvariabel:

```
System.out.println("x-koordinat: "+p.x);  
System.out.println("y-koordinat: "+p.y);
```

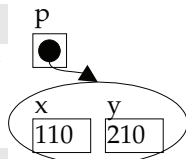
Vi kan også ændre `p`'s koordinater:

```
p.x = 110;  
p.y = 210;
```

Variablerne `x` og `y` i `Point`-objektet kan behandles fuldstændig som andre variabler af typen `int`, når vi bare husker at skrive "`p`." foran. F.eks. kan man tælle x-koordinaten op med 5:

```
p.x = p.x + 5;
```

`x` og `y` kaldes objektvariabler, fordi de tilhører objektet `p`.



Efter tildeling af `p.x` og `p.y`

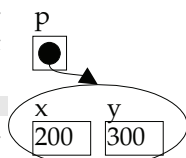
## 3.2.3 Metodekald

I stedet for at ændre objektet udefra kan vi bede objektet selv udføre ændringen. Metoden `move()` flytter punktet til et bestemt koordinatsæt, dvs. den ændrer `x`- og `y`-koordinaten.

```
p.move(200,300);
```

Man siger, at man foretager et metodekald på objektet, som `p` refererer til og man skriver: *p punktum metodenavn parenteser*.

Efter metodekaldet til `move()`, har `x`- og `y`-koordinaterne ændret sig i det objekt, som `p` peger på.



Efter kald af `move(200,300)`

**Et objekt kan indeholde metoder**

**Et metodekald på et objekt kan ændre objektets data**

Til højre er `Point`-klassen illustreret i UML-notationen.

Herunder er nogle af de metoder, som punktobjekter forstår (en oversigt over klassen kan findes i appendiks, afsnit 3.9.1).

Nogle af `Point`-klassens metoder

klassenavn:	<b>Point</b>
variabler:	<code>x</code> <code>y</code>
metoder:	<code>move(x,y)</code> <code>translate(x,y)</code>

**move(int x, int y)**

Sætter punktets koordinater

**translate(int x, int y)**

Rykker punktets koordinater relativt i forhold til, hvor det er

Først står navnet på metoden med fed, f.eks.: **move**.

Derefter står parametrene adskilt af komma, f.eks.: (int `x`, int `y`).

For hver parameter er angivet en type og et navn. Typen angiver, hvilke værdier man kan kalde metoden med og bruges til at kontrollere, at man har kaldt den med en værdi af den rigtige type. Navnet i beskrivelsen bruges kun til at minde om, hvad parameteren betyder.

Bemærk, at kaldet derfor ser anderledes ud end beskrivelsen:

```
p.move(200,300);           // korrekt
p.move(int 200, int 300);  // sprogfejl: parametertyper angivet ved kald.
p.move(x=200, y=300);     // sprogfejl: parameternavne angivet ved kald.
```

---

**I parenteserne i metodekaldet giver man oplysninger til objektet om, hvordan man vil have metoden udført**

**Oplysningerne kaldes parametre (eller argumenter)**

---

I kaldet til move() ovenfor gav vi oplysningerne 200 og 300 som parametre.

## 3.2.4 Eksempel

Her er et eksempel på tingene, vi har vist ovenfor:

```
import java.awt.*; // Point-klassen skal importeres fra pakken java.awt

public class BenytPunkter
{
    public static void main(String[] arg)
    {
        Point p;
        p = new Point();

        int a;

        a = p.x;

        System.out.println("a: "+a);

        System.out.println("x-koordinat: "+p.x);
        System.out.println("y-koordinat: "+p.y);

        p.x = 110;
        p.y = 210;
        System.out.println("x-koordinat: "+p.x);
        System.out.println("y-koordinat: "+p.y);

        p.move(200,300);
        System.out.println("x-koordinat: "+p.x);
        System.out.println("y-koordinat: "+p.y);

        p.x = p.x + 5;
        System.out.println("x-koordinat: "+p.x);
        System.out.println("y-koordinat: "+p.y);

        p.translate(-10,20);
        System.out.println("x-koordinat: "+p.x);
        System.out.println("y-koordinat: "+p.y);
    }
}

a: 0
x-koordinat: 0
y-koordinat: 0
x-koordinat: 110
y-koordinat: 210
x-koordinat: 200
y-koordinat: 300
x-koordinat: 205
y-koordinat: 300
x-koordinat: 195
y-koordinat: 320
```

## 3.2.5 Import af standardklasser

Øverst i kildeteksten "importerer" vi alle klasser i pakken java.awt:

```
import java.awt.*;
```

Dette fortæller oversætteren, hvor den skal lede efter definitionen af klasserne, vi bruger i programmet. I dette tilfælde er det for at oversætteren skal kende til Point-klassen (der findes i pakken java.awt).

En pakke er en samling klasser med beslægtede funktioner. AWT står for "Abstract Window Toolkit" og java.awt indeholder forskellige nyttige klasser til at tegne grafik på skærmen, herunder klasser til at repræsentere punkter og rektangler.

Lige nu er det nok at vide, at de fleste klasser skal importeres, før de kan bruges (hvis du er meget nysgerrig, kan du læse den første del af kapitel 6 om pakker allerede nu).

## 3.3 Rektangler (klassen Rectangle)

Vi vil nu gå videre til nogle lidt mere udviklede objekter af klassen Rectangle. Den bruges sjældent i praksis (så du behøver ikke lære dens metoder udenad), men den er velegnet til at illustrere ideer omkring oprettelse af objekter (konstruktører) og metodekald med returværdi.

Et rektangel-objekt består af en x- og y-koordinat og en højde og bredde. Disse objektvariabler (data) hedder *x*, *y*, *width* og *height*.

En variabel med navnet *r* af typen Rectangle erklæres med:

```
Rectangle r;
```

Ligesom med Point skal vi have lavet et rektangel-objekt, som *r* refererer til, ellers peger *r* ingen steder hen (den har værdien null):

```
r = new Rectangle();
```

Som med de simple typer kan man godt erklære variablen og initialisere den i samme linje:

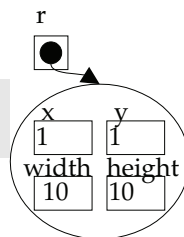
```
Rectangle r = new Rectangle();
```

Det Rectangle-objekt, der oprettes, får *x*, *y*, *width* og *height* sat til 0.

Vi kan ændre dette til (1,1,10,10) med:

```
r.x = 1;  
r.y = 1;  
r.width = 10;  
r.height = 10;
```

Det er besværligt, hvis man skal bruge fire linjers programkode på, at sætte et objekts værdier, når man opretter det. Det kan gøres kortere.



### 3.3.1 Konstruktører

Når man vil oprette et objekt med bestemte startværdier, kan det gøres ved, at benytte en *konstruktør*, hvor startværdierne kan angives.

For eksempel kan et rektangel oprettes med:

```
r = new Rectangle(1,1,10,10);
```

De fire parametre i parenteser fortæller, at det rektangel, som skal skabes, som start skal have det øverste venstre hjørne i (1,1) og en bredde og en højde på 10.

Det er i virkeligheden en slags metodekald, vi her foretager, så det er ikke nogen tilfældighed, hvis det ligner.



---

Når man skaber et nyt objekt med new, kaldes en konstruktør

Konstruktøren opretter objektet og initialiserer dets data (variabler)

Nogle konstruktører har parametre, der beskriver, hvordan objektet skal oprettes (hvilke værdier dets variabler skal have)

---

Herunder er beskrevet tre konstruktører for Rectangle – dvs. tre måder, rektangler kan oprettes på.

*Nogle af Rectangle-klassens konstruktører*

```
Rectangle()  
    opretter et rektangel i (0,0), hvis bredde og højde er 0  
Rectangle(int bredde, int højde)  
    opretter et rektangel i (0,0) med den angivne bredde og højde  
Rectangle(int x, int y, int bredde, int højde)  
    opretter et rektangel i (x,y) med den angivne bredde og højde
```

Point-klassens konstruktører er beskrevet i appendikset afsnit 3.9.1. Vi kan f.eks. bruge den, der har to parametre:

```
Point p;  
p= new Point(8,6); // skaber et Point med koordinaterne (8,6)
```

## 3.3.2 Metoder

Vi vil nu lave et lille program, der tjekker, om punktet *p* ligger inde i rektanglet *r*. Vi erklærer en variabel, *inde*, af type boolean, som vi kan bruge til at gemme resultatet af vores undersøgelse i.

```
boolean inde;
```

Objekter af klassen Rectangle har en metode, *contains()*, som kan fortælle, om et punkt ligger inde i rektanglet:

```
inde = r.contains(p);
```

Det, der sker, er, at vi kalder metoden *contains()* – svarende til spørgsmålet ”indeholder du *p*?” – på rektanglet *r*. Vi giver *p* med som parameter, således at rektanglets metode ved, at det lige præcis er punktet *p*, som skal undersøges. Metoden bliver udført og foretager nogle beregninger, som vi ikke kan se og til sidst kommer den ud med et svar. Dette svar returneres til os og bliver gemt i variabelen *inde*. Modsat tilfældet med Point-objekters *move()*- og *translate()*-metoder er rektanglers indhold uændret af kald af *contains()*.

---

**Ikke alle metoder på et objekt ændrer på det**

**Nogle metoder giver et svar tilbage (returnerer et resultat)**

---

Prøv at sammenligne det med kaldet til *Math.sqrt()*, som vi så i forrige kapitel:

```
hypotenuse = Math.sqrt(x*x + y*y);
```

Det er samme mekanisme: Vi spørger *Math.sqrt()* om, hvad kvadratroden af  $x^2+y^2$  er og svaret, som metoden giver tilbage, gemmer vi i variabelen *hypotenuse*.

### 3.3.3 Metoders returtype

Ligesom parametre skal være af den rette type, gælder for resultatet af et metode-kald at:

---

**En metode giver et resultat af en bestemt type, når den bliver udført**

**Dette kaldes metodens returtype**

---

Math.sin() har returtypen double, mens contains() på et rektangel-objekt har returtypen boolean. Det er derfor, vores variabel inde også skulle have typen boolean.

Hvis punktet var inde i rektanglet, så vil vi skrive det på skærmen:

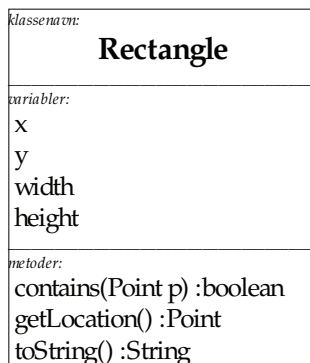
```
if (inde==true)
{
    System.out.println("p er inde i r");
}
```

Herunder ses nogle af Rectangle-klassens metoder. Foran metode-navnene står returtyperne. I kursiv står spørgsmålene, som de svarer til. En mere komplet oversigt over klassen kan findes i appendiks i afsnit 3.9.2.

*Nogle af Rectangle-klassens metoder.*

boolean <b>contains</b> (Point p)	<i>"indeholder du p?"</i>
returnerer true, hvis p er inden for rektanglet, ellers false.	
Point <b>getLocation</b> ()	<i>"hvad er din placering?"</i>
returnerer et Point-objekt, der har samme koordinater som rektanglets øverste venstre hjørne.	
String <b>toString</b> ()	<i>"hvordan vil du beskrive dig selv?"</i>
giver en beskrivelse af rektanglet med (x,y)-koordinater og mål som en streng.	

Her er Rectangle illustreret i UML-notation.



Returtyperne skrives her efter metodenavnet. Ofte vil vi af hensyn til overskueligheden undlade returtyperne (ligesom vi nogle gange undlader parametertyperne).

Herunder ses et samlet eksempel med to punkter. Det andet punkt, `p2`, undersøger vi direkte i en if-sætning uden at bruge en mellemvariabel.

```
import java.awt.*;

public class BenytRektangler
{
    public static void main(String[] arg)
    {
        Point p, p2;
        Rectangle r;

        p = new Point();
        p2 = new Point(6,8);
        r = new Rectangle(1,1,10,10);

        boolean inde;
        inde = r.contains(p);

        if (inde==true)
        {
            System.out.println("p er inde i r");
        }

        if (r.contains(p2))
        {
            System.out.println("p2 er inde i r");
        }
    }
}
```

p2 er inde i r

### 3.3.4 Metoders parametre

Her er et eksempel, der beregner afstanden (distancen) mellem punktet `p` og rektanglet `r`'s øverste venstre hjørne. Det foregår ved, at vi spørger `r`: `getLocation()` – "*hvad er din position?*". Svaret bruger vi som parameter til et spørgsmål til `p`: `distance(svaret fra r)` – "*Hvad er din afstand til (svaret fra r)?*"

```
double afstand;
afstand = p.distance(r.getLocation());
```

---

**Ved et metodekald beregnes først alle parametrene og derefter udføres metoden**

---

Dvs. først beregnes parameteren, `getLocation()` kaldes altså på `r`. Den returnerer et punkt som er `r`'s (x,y) og derefter kaldes `distance()` på `p` med dette Point-objekt som parameter.

Man kunne også bruge en mellemvariabel og skrive:

```
Point rpunkt;
rpunkt = r.getLocation(); // rpunkt er r's øverste venstre hjørne
afstand = p.distance(rpunkt);
```

Det er i starten lettere at læse kode med mellemvariabler, men når eksemplerne bliver mere indviklede, bliver antallet af mellemvariabler for stort. Man skal øve sig i selv at forestille sig, at der er nogle mellemregninger med mellemvariabler.

## 3.4 Tekststrengene (klassen String)

Vi kommer nu til de mest brugte objekter, nemlig tekststrengene (af typen String). En variabel, der refererer til strengene, erklæres ved at skrive

```
String s;
```

Nu har vi defineret, at *s* er en variabel, der kan referere til objekter af typen String, men den peger endnu ikke på nogen konkret streng.

Lad os tildele *s* en værdi. Så er situationen som vist på figuren til højre:

```
s = "Ude godt";
```

Vi kan bruge *s* i vores program, f.eks. til at skrive ud på skærmen:

```
System.out.println("Strengen s indeholder: "+s);
```

Vi kan spørge streng-objektet om forskellige ting. For eksempel kan vi kalde `length()`, der svarer til spørgsmålet "hvor lang er du?":

```
int strengensLængde;  
strengensLængde = s.length(); // strengen svarer med tallet 8  
System.out.println("s er "+strengensLængde+" tegn lang");  
s er 8 tegn lang
```

Vi kunne også springe mellemvariablen over og skrive:

```
System.out.println("s er "+s.length()+" tegn lang");
```

Metoden `toUpperCase()` svarer til spørgsmålet "hvordan ser du ud med store bogstaver?":

```
System.out.println("s med store bogstaver: "+s.toUpperCase());  
s med store bogstaver: UDE GODT
```

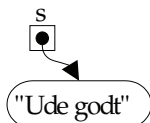
Herunder ses nogle af metoderne, man kan kalde på strengene. I kursiv til højre står spørgsmålene, som de svarer til.

*Nogle af String-klassens metoder. En mere fuldstændig oversigt kan findes i afsnit 3.9.3.*

String <b>replace</b> (String søgetekst, String erstatning)	<i>"hvad hvis x erstattes med y i strengen?"</i>
giver en ny streng, som er identisk med denne streng, bortset fra at alle steder i strengen hvor teksten <i>søgetekst</i> står er blevet erstattet med teksten <i>erstatning</i> .	
String <b>substring</b> (int startindeks)	<i>"hvad er delstrengen fra x?"</i>
giver en ny streng, som er en del af denne streng. Delstrengen starter ved <i>startindeks</i> og går til slutningen.	
String <b>substring</b> (int startindeks, int slutindeks)	<i>"hvad er delstrengen fra x til y?"</i>
giver en ny streng, som er en del af denne streng. Delstrengen starter ved <i>startindeks</i> og slutter ved <i>slutindeks</i> (til og med <i>slutindeks</i> -1).	
String <b>toLowerCase</b> ()	<i>"hvordan ser du ud med små bogstaver?"</i>
giver en ny streng, som er identisk med denne streng, bortset fra at alle store bogstaver er erstattet med små.	
String <b>toUpperCase</b> ()	<i>"hvordan ser du ud med store bogstaver?"</i>
giver en ny streng, som er identisk med denne streng, bortset fra at alle små bogstaver er erstattet med store.	
boolean <b>equals</b> (String str)	<i>"er det samme indhold?"</i>
giver sand, hvis denne streng indeholder den samme tegnsekvens som <i>str</i> , ellers falsk.	
int <b>length</b> ()	<i>"hvad er din længde?"</i>
giver længden af (antal tegn i) strengen.	
int <b>indexOf</b> (String str)	<i>"hvor er delstrengen x?"</i>
giver indeks på første forekomst af <i>str</i> som delstreng. Er <i>str</i> ikke en delstreng, returneres -1.	



*s refererer ikke til noget*



*s refererer nu til en streng*

Herunder ses et eksempel, hvor nogle af metoderne er afprøvede:

```
// BenytStreng.java
// Viser brugen af String-klassen og dens metoder.
public class BenytStreng
{
    public static void main(String[] arg)
    {
        String s;
        s = "Ude godt";
        System.out.println("Strengen s indeholder: "+s);
        System.out.println("s er "+s.length()+" tegn lang");
        System.out.println("s med store bogstaver: "+s.toUpperCase());
        System.out.println("Tekst på plads nummer 4 og frem: "+s.substring(4));
        System.out.println("Det første g er på plads nummer: "+s.indexOf("g"));
    }
}
```

---

Strengen s indeholder: Ude godt  
s er 8 tegn lang  
s med store bogstaver: UDE GODT  
Tekst på plads nummer 4 og frem: godt  
Det første g er på plads nummer: 4

### 3.4.1 Streng-objekter er uforanderlige

De fleste objekter tillader, at deres data ændres, enten ved at man direkte har adgang til deres variabler eller gennem kald af metoder.

Når vi skal ændre i et Point-objekt, f.eks. så dets x og y er (1,1), skriver vi:

```
p.move(1,1); // p forandres
```

Sådan er det ikke med strenge: Metoderne i strenge er indrettet sådan, at de ikke ændrer i objektet. I stedet returneres et nyt streng-objekt, som indeholder resultatet af ændringen.

Kalder man derfor en ændringsmetode på en streng, bliver strengen *ikke* ændret:

```
s.replace("godt","fint"); // objektet s peger på forandres ikke
```

replace() giver en ny streng tilbage til os, hvor alle steder med "godt" er erstattet med "fint", men den bliver smidt væk med det samme, da vi ikke bruger returværdien.

I stedet skal vi huske returværdien:

```
String s2;
s2 = s.replace("godt","fint");// s forandres ikke, men s2 husker resultatet
```

Nu bliver resultat-strengen gemt vha. s2 (den streng s peger på er som sagt uforandret).

Her ses samlet et eksempel på, hvordan man bruger streng:

```
// BenytStreng2.java
public class BenytStreng2
{
    public static void main (String[] arg)
    {
        String s1, s2, s3, s4;
        s1 = "Ude godt, men hjemme bedst.";
        s1.toUpperCase(); // lav store bogstaver, men smid resultatet væk
        s2 = s1.toUpperCase(); // lav store bogstaver fra s1, gem det i s2
        s3 = s2.replace("E", "X"); // erstat E med X på s2 og gem resultatet i s3
        s4 = s3.substring(4, 16); // tag delstreng på s3 og gem resultatet i s4

        System.out.println ("s1: " + s1); // s1 er stadig med små bogstaver
        System.out.println ("s2: " + s2); // s2 har ikke fået ændret E til X
        System.out.println ("s3: " + s3); // s3 er ikke delstrengen på plads 4-16
        System.out.println ("s4: " + s4); // s4 er delstrengen på plads 4-16
    }
}
```

---

s1: Ude godt, men hjemme bedst.  
s2: UDE GODT, MEN HJEMME BEDST.  
s3: UDX GODT, MXN HJXMMX BXDST.  
s4: GODT, MXN HJ

Variablerne s1, s2, s3 og s4 får tildelt en reference (pegepind) til hvert sit strengobjekt og derefter ændrer deres indhold sig ikke, uanset hvilke metoder der kaldes på objekterne.

Bemærk, at selvom streng-objekterne i sig selv er uforanderlige, kan streng-*variabler* godt ændres:

```
s = s.replace('d','f'); // sæt s til at referere resultatet af replace()
```

Forskellen mellem en metode, der ændrer på det objekt, den bliver kaldt på og en metode, der returnerer en værdi, kan være svær at forstå i starten, men det kommer i takt med, at du programmerer selv.

### 3.4.2 Man behøver ikke bruge new til String-objekter

De andre klasser, vi har set indtil nu, har vi brugt til at skabe nye objekter med. Når vi skulle lave et nyt Point-objekt, kaldte vi dens konstruktør vha. new, f.eks.:

```
Point p;  
p = new Point(0,0);
```

Lige netop med strenge behøves det ikke. Her skriver man typisk:

```
String s;  
s = "Ude godt";
```

Man *kan* godt skrive:

```
s = new String("Ude godt");
```

I det sidste tilfælde skabes et nyt String-objekt, som også indeholder teksten "Ude godt", så der i lageret er *to* strenge med samme indhold, hvilket er unødvendigt. Netop fordi strenge ikke kan ændres, når de først er skabt, har man aldrig brug for kopier. Hvorfor skulle man lave en kopi, når den alligevel altid vil være helt den samme som originalen?

### 3.4.3 Navnesammenfald for metoder

I tabellen over Strings metoder er der en, der er nævnt to gange; substring(). Den findes i to varianter: **substring**(int startindeks) og **substring**(int startindeks, int slutindeks).

Hvilken variant der kaldes i BenytStrenge2.java ved tildelingen af *s4*, kan man se ud fra hvilke parameterlister, der passer sammen. I dette tilfælde den metode med to parametre.

Så længe der er forskel på antallet af parametre, er det simpelt nok, ellers må man kigge på typerne af parametrene.

### 3.4.4 At sætte strenge sammen med +

Operatoren + bruges ikke kun til at lægge tal sammen. Hvis enten højre- eller venstre-siden er en streng, bliver + opfattet som: *"konverter begge sider til strenge og sæt dem i forlængelse af hinanden til en samlet streng"*.

Hvis man f.eks. skriver:

```
Point p;  
p=new Point(1,1);  
System.out.println("Svaret er: "+p);
```

```
Svaret er: java.awt.Point[x=1,y=1]
```

Sker der i computeren noglunde følgende:

```
String s1;  
String s2;  
String s3;  
s1 = "Svaret er: ";  
s2 = p.toString(); // toString() er en metode alle objekter har  
s3 = s1 + s2;  
System.out.println(s3);
```

Metoden toString() laver en streng-repræsentation af et objekt. Alle objekter har en toString()-metode og oversætteren sætter kode ind, der kalder toString(), hvis den møder et + mellem en streng og en anden slags objekt.

Alle de simple typer kan også laves om til strenge med +:

```
int i;  
i = 42;  
System.out.println("Svaret er: "+i);
```

Java kigger ikke på indholdet af strengene, så "2" (som streng) + 3 (som tal) giver "23" (som streng).

Man kan altså bruge operatoren + til et lille trick: For at få noget repræsenteret som en streng, kan man sammensætte det med en tom streng:

```
String s;  
int i;  
i = 42;  
s = ""+i; // nu refererer s til strengen "42"
```

Man kan derimod ikke skrive:

```
s = i; // sprogfejl: konverterer ikke automatisk fra int til String.
```

...eller...

```
i = s+1; // sprogfejl: konverterer ikke automatisk fra String til int.
```

... selvom s er "42".

## 3.4.5 Sammenligning

Umiddelbart kunne man fristes til at sammenligne to strenge med == ligesom med de simple typer. Det går ofte (men ikke altid!) godt:

```
s1 = "Hej verden";  
s2 = s1;  
if (s1 == s2) System.out.println("s1 og s2 er ens."); // forkert!  
else System.out.println("s1 og s2 er IKKE ens.");  
s1 og s2 er ens.
```

Imidlertid sammenligner == ikke *indholdet af* objekterne, men *referencerne til* (adresserne på), objekterne. Sammenligningen s1==s2 er kun sand, fordi s1 og s2 peger på det samme objekt.

Derfor vil s1==s2 altid være falsk, hvis s1 og s2 refererer til to objekter forskellige steder i hukommelsen, også selvom de har samme indhold:

```
s1 = "Hej verden";  
s2 = "Hej " + s1.substring(4); // giver "Hej "+"verden" = "Hej verden"  
if (s1 == s2) System.out.println("s1 og s2 er ens."); // forkert!  
else System.out.println("s1 og s2 er IKKE ens.");  
s1 og s2 er IKKE ens.
```

I stedet bør man kalde equals()-metoden, dvs. spørge et af objekterne "har du samme indhold som dette objekt?" og give det andet objekt som parameter:

```
if (s1.equals(s2)) System.out.println("s1 og s2 er ens."); // korrekt  
else System.out.println("s1 og s2 er IKKE ens.");  
s1 og s2 er ens.
```

Dette gælder i virkeligheden ikke kun strenge. Alle objekter har en equals()-metode, som kan bruges til at afgøre, om indholdet af to objekter er ens og den bør bruges fremfor ==.

---

**Sammenligning af adresser på objekter sker med ==**

**Sammenligning af objekters indhold sker med equals()-metoden**

---

## 3.4.6 Opgaver

1. Skriv et program, der finder positionen af det første mellemrum i en streng.  
(vink: Brug metoden `indexOf(" ")`)
2. Skriv et program, der fjerner det første ord i en sætning (indtil første mellemrum).
3. Skriv et program, der tæller antallet af mellemrum i en tekst.
4. Skriv et program, der fjerner den første forekomst af ordet "måske" fra en tekst.  
Ændr derefter programmet, så det fjerner alle forekomster af ordet (brug en løkke).
5. Skriv et program, der finder og fjerner alle forekomster af ordet "måske" fra en tekst, uanset om det er skrevet med store eller små bogstaver.
6. Skriv et program, der undersøger, om en tekst er et palindrom, dvs. med samme stavning forfra og bagfra (som f.eks. "regninger", "russerdressur", "vær dog god ræv").  
(vink: Træk de enkelte tegn ud af strengene med `substring(n,n+1)` og husk, at strengobjekter skal sammenlignes med `.equals()`-metoden: `s1.equals(s2)`).
7. Udvid programmet til at tage højde for store/små bogstaver, tegnsætning og mellemrum, sådan at de følgende palindromer også genkendes: "Selmas lakserøde garagedøre skal samles" og "Åge lo, da baronesse Nora bad Ole gå".

## 3.5 Lister (klassen ArrayList)

En ArrayList er en liste af objekter, der bliver nummereret efter et indeks.

ArrayList
<code>add(objekt :Object)</code> <code>add(indeks :int, objekt :Object)</code> <code>size() :int</code> <code>get(indeks) :Object</code> <code>toString() :String</code>

Konstruktører og metoder er beskrevet i appendiks, afsnit 3.9.6.

---

### En ArrayList er en liste af objekter

---

En liste oprettes normalt med viden om hvilken slags objekter, den skal indeholde. F.eks.:

```
ArrayList<String> liste;  
liste = new ArrayList<String>();
```

opretter en liste af strenge. Man tilføjer elementer til listen med `liste.add()`, f.eks.:

```
liste.add("æh");
```

der tilføjer strengen "æh" til listen.

Man kan sætte ind midt i listen med `liste.add( indeks , objekt )`, f.eks.:

```
liste.add(0, "øh");
```

der indsætter "øh" på plads nummer 0, sådan at listen nu indeholder først "øh" og så "æh". Alle elementerne fra og med det indeks, hvori man indsætter, får altså rykket deres indeks et frem.

Man aflæser et element i listen med `liste.get( indeks )`.

Med `liste.size()` får man antallet af elementer i listen, i dette tilfælde 2.



## Metoder

**void add ( element )**

Føj *element* til listen. *element* skal være af ArrayList-objektets type.

**void add ( int indeks , element )**

Indsætter *element* i listen lige før plads nummer *indeks*. Første element er på plads nummer 0.

**void remove ( int indeks )**

Sletter elementet på plads nummer *indeks*.

**int size ()**

returnerer antallet af elementer i listen.

**Object get ( int indeks )**

Returnerer en reference til objektet på plads nummer *indeks* (regnet fra 0).

**String toString ()**

returnerer listens indhold som en streng. Dette sker ved, at konvertere hver af elementerne til en streng.

*ArrayList-klassen skal importeres med `import java.util.*`; før den kan bruges*

Her er et lille eksempel:

```
import java.util.*;

public class BenytArrayList
{
    public static void main(String[] arg)
    {
        ArrayList<String> liste; // opret liste-variabel
        liste = new ArrayList<String>(); // opret liste-objekt

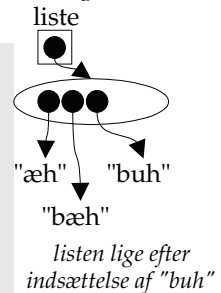
        liste.add("æh");           // føj til listen
        liste.add("bæh");
        liste.add("buh");

        System.out.println("Listen har elementerne "+liste.toString());

        liste.add(2,"og");
        System.out.println("Nu har listen "+liste); // .toString() kaldes implicit

        liste.remove(0);
        System.out.println("Nu har listen "+liste+" og størrelsen "+liste.size());
        System.out.println("På plads nummer 1 er: "+liste.get(1));

        for (String element : liste) { // gennemløb alle elementerne i listen
            System.out.println("Et element i listen: "+ element);
        }
    }
}
```



```
Listen har elementerne [æh, bæh, buh]
Nu har listen [æh, bæh, og, buh]
Nu har listen [bæh, og, buh] og størrelsen 3
På plads nummer 1 er: og
Et element i listen: bæh
Et element i listen: og
Et element i listen: buh
```

Vi indsætter først tre (referencer til) strenge i listen (se figur). Dernæst lægges "og" ind på plads nummer 2, dvs. efter "bæh" og før "buh". Til sidst fjernes "æh" på plads nummer 0.

Lister kan blive vilkårligt lange. De sørger selv for at reservere mere hukommelse, hvis det bliver nødvendigt.

## 3.5.1 Gennemløb af lister

Læg mærke til, hvordan man kan gennemløbe en liste:

```
for (String element : liste) {  
    System.out.println("Et element i listen: " + element);  
}
```

Denne specielle syntaks til for-løkken (foreach) er beregnet til at gennemløbe alle elementerne i en liste. Den har den generelle form:

```
for (Elementtype elementvariabel : liste) kommando;
```

## 3.5.2 Eksempel: Eventyrfortælling

Her er et program, der kombinerer forskellige personer og handlinger til et lille eventyr:

```
import java.util.*;  
  
public class Eventyr  
{  
    public static void main(String[] arg)  
    {  
        ArrayList<String> personer = new ArrayList<String>(); // liste af strenge  
        personer.add("De tre små grise");  
        personer.add("Ulven");  
        personer.add("Rødhætte");  
  
        ArrayList<String> handler = new ArrayList<String>();  
        handler.add("slikker sig om munden");  
        handler.add("får en idé!");  
        handler.add("gemmer sig i skoven");  
  
        for (int i=0; i<5; i++) {  
            int personNummer = (int) (Math.random()*personer.size());  
            String person = personer.get( personNummer );  
            String handling = handler.get( (int)(Math.random()*handler.size()) );  
            System.out.println(person + " " + handling);  
        }  
    }  
}  
  
Ulven får en idé!  
Ulven slikker sig om munden  
Rødhætte gemmer sig i skoven  
De tre små grise slikker sig om munden  
Ulven gemmer sig i skoven
```

## Opgave

Udvid programmet, så eventyret bliver mere interessant.

## 3.5.3 Eksempel: Punkter i en liste

I det næste eksempel lægges tre Point-objekter ind i en liste og listen gennemløbes, for at finde punktet med den mindste afstand til (0,0) (også kaldet origo).

```

import java.awt.*;
import java.util.*;

public class MindsteAfstand
{
    public static void main(String[] arg)
    {
        ArrayList<Point> punktliste;    // punkt-liste
        Point origo, p1, p2, p3;
        double minDist = 10000;
        punktliste = new ArrayList<Point>();
        origo = new Point(0,0);
        p1 = new Point(0,65);
        p2 = new Point(50,50);
        p3 = new Point(120,10);

        punktliste.add(p1);
        punktliste.add(p2);
        punktliste.add(p3);

        for (Point p : punktliste)
        {
            double dist;

            dist = origo.distance(p);
            if (dist<minDist)
            {
                minDist=dist;
            }
        }

        System.out.println("Den mindste afstand mellem punkterne "
            + punktliste + " og (0,0) er "+minDist);
    }
}

```

```

Den mindste afstand mellem punkterne [java.awt.Point[x=0,y=65],
java.awt.Point[x=50,y=50], java.awt.Point[x=120,y=10]] og (0,0) er 65.0

```

### 3.5.4 Historisk bemærkning: Lister før JDK 1.5

Bruger du en ældre udgave af Java (eller skal dit program også virke i ældre udgaver af Java), er du nødt til at arbejde med lister, sådan som man gjorde, før JDK 1.5 kom.

Før JDK 1.5 kom i 2004, måtte man bruge en tællevariabel, der løb fra 0 til listens størrelse (liste.size()) til at gennemløbe listen element for element. Programkoden så ofte ud som:

```

for (int n=0; n<liste.size(); n++) {
    String element = (String) liste.get(n);
    System.out.println("Element nummer "+n+" i listen: " + element);
}

```

```

Element nummer 0 i listen: bæh
Element nummer 1 i listen: og
Element nummer 2 i listen: buh

```

Hvis man har brug for at kende elementets nummer (her er det i variablen n), kan den gamle måde stadig være den bedste.

Før JDK 1.5 var man også henvist til at oprette lister uden at angive elementernes type:

```

ArrayList liste;    // opret liste-variabel uden <String>-elementtype
liste = new ArrayList(); // opret liste-objekt uden <String>-elementtype

```

I stedet måtte man foretage en eksplicit typekonvertering, når man tog noget ud af listen:

```

for (int i=0; i<liste.size(); i++) {
    String element = (String) liste.get(i);
    System.out.println("Element nummer "+i+" i listen: " + element);
}

```

### 3.5.5 Eksempel: Blanding af kort med ArrayList

I det følgende bruges nogle af de metoder, der er nævnt i appendikset om ArrayList (afsnit 3.9.6). Dette program blander nogle spillekort (beskrevet som strenge<sup>1</sup>) i en liste:

```
import java.util.*;

public class BlandKort
{
    public static void main(String[] arg)
    {
        ArrayList<String> bunke = new ArrayList<String>();

        // Opbyg bunken
        for (int n=2; n<9; n++)
        {
            bunke.add("runder"+n); // ruder
            bunke.add("klør"+n);   // klør
            bunke.add("spar"+n);   // spar
        }

        System.out.println("Før blanding: "+bunke); // bunke.toString() kaldes
        int antalKort = bunke.size();

        // Bland bunken
        for (int n=0; n<100; n++)
        {
            int nr = (int) (Math.random() * antalKort); // find en tilfældig plads

            String kort = bunke.get(nr);                // tag et kort ud
            bunke.remove(nr);

            nr = (int) (Math.random() * antalKort);
            bunke.add(nr, kort);                        // sæt det ind et andet sted
        }

        System.out.println("Efter blanding: "+bunke);
    }
}

Før blanding: [runder2, klør2, spar2, ruder3, klør3, spar3, ruder4, klør4, spar4,
runder5, klør5, spar5, ruder6, klør6, spar6, ruder7, klør7, spar7, ruder8, klør8,
spar8]
Efter blanding: [spar3, klør3, ruder7, spar5, spar2, ruder5, ruder6, klør6,
spar6, klør5, klør8, ruder2, ruder4, klør7, ruder3, spar8, spar4, ruder8, klør4,
klør2, spar7]
```

Blandingen udføres ved 100 gange at tage et kort fra en tilfældig plads (med get()), fjerne det (med remove()) og flytte det til en anden tilfældig plads (med add() på den nye plads).

---

<sup>1</sup> I et rigtigt program ville de enkelte kort nok være repræsenteret med objekter fra en Kort-klasse, hvor hvert Kort-objekt har objektvariablerne farve og værdi.

## 3.6 Datoer (klassen Date)

Date-klassen repræsenterer et punkt i tiden (en dato og et klokkeslæt).

Date
konstruktører: Date() Date(tid_i_millisekunder :long) Date(år, måned, dag, time, minut) metoder: getTime() :long setTime(tid_i_millisekunder :long) after(andenDato :Date) :boolean getDate() :int setDate(dag :int) getMonth() :int setMonth(måned :int) ...

For at oprette et dato-objekt, der repræsenterer dags dato og tid, skriver vi:

```
Date netopNu;  
netopNu = new Date(); // dags dato og tidspunkt lige nu
```

new-operatoren er som bekendt bindeleddet mellem en klasse (f.eks. Date) og et objekt (en konkret dato, f.eks. 24/12 2000 kl. 18:37).

For at oprette en dato, der repræsenterer et andet tidspunkt, kan vi bruge en af de andre konstruktører, der får årstal, måned (regnet fra 0), dag, time og minut. Undertegnede er født den 1. januar 1971, så min fødselsdag kunne oprettes med:

```
jacobsFødselsdag = new Date(71,0,1,12,00); // 1.januar 1971 kl. 12:00
```

I appendiks 3.9.5 er et udvalg af Date-klassens konstruktører og metoder beskrevet.

Eksemplet nedenfor regner på min fødselsdato og finder ud af, hvornår jeg var halvt så gammel som nu.

```
// Datoer.java
// Viser brugen af Date-klassen og dens metoder.

import java.util.*; // Date-klassen er i pakken java.util

public class Datoer
{
    public static void main (String[] arg)
    {
        Date netopNu;
        Date jacob;

        netopNu = new Date(); // dags dato og tidspunkt lige nu
        jacob = new Date(71,0,1,12,00); // 1. januar 1971 kl. 12:00

        System.out.println("Dags dato: "+netopNu.toString());
        System.out.println("Jacob blev født "+jacob); // .toString() implicit

        // Lad os regne Jacobs alder ud (i millisekunder)
        long nuMs;
        long jacobMs;
        long alderMs;

        nuMs = netopNu.getTime();
        jacobMs = jacob.getTime();
        alderMs = nuMs - jacobMs;

        // Hvornår var han halvt så gammel?
        jacob.setTime(nuMs - alderMs/2);
        System.out.println("Jacob var halvt så gammel "+jacob);
    }
}

Dags dato: Tue Sep 25 06:07:46 CEST 2007
Jacob blev født Fri Jan 01 12:00:00 CET 1971
Jacob var halvt så gammel Sun May 14 09:33:53 CEST 1989
```

Kaldet `jacob.setTime(...)` ændrer objektet, så Jacobs fødselsdag blev glemt.

Man kan gøre meget mere med datoer end vist her:

- `GregorianCalendar`-klassen repræsenterer vort kalendersystem (det gregorianske / juli-anske) og har alle de kalenderfunktioner, man kunne ønske sig. Med den kan man arbejde med ugedage, måneder, år, tidszoner, sommertid etc.
- Med `DateFormat`-klassen kan man formatere og udskrive datoer langt pænere end med `toString()` og på alle mulige sprog (bl.a. på dansk). Klassen kan også gå den anden vej: Analysere en tekststreng og finde frem til datoen den repræsenterer.

### 3.6.1 Opgaver

1. Ret `Datoer`-programmet sådan, at Jacobs fødselsdato ikke går tabt (opret et tredje objekt, i stedet for at ændre i `jacob`-objektet).
2. Skriv et program, der udskriver datoen for i morgen, om en uge og om et år.
3. Skriv et program, der ud fra en persons fødselsdato udskriver alle fødselsdage, som personen har fejret indtil nu (lav f.eks. en `while`-løkke, hvor du tæller år frem fra fødselsdatoen og brug metoden `before()`, til at tjekke, om du er nået frem til nu).

## 3.7 Test dig selv

Hvis du kan svare på spørgsmålene herunder, kan du være rimelig sikker på, at du har forstået alt det væsentlige i dette kapitel. Brug papir og blyant og skriv et par linjers svar til hvert spørgsmål. I resuméet i næste afsnit kan du finde nogle vejledende svar.

- 1) Hvad er et objekt?
- 2) Hvad er en klasse?
- 3) Hvordan oprettes en variabel, der kan referere til et objekt?
- 4) Hvordan oprettes et objekt?
- 5) Hvordan kan man ændre på et objekt (dets data)?
- 6) Hvad er en metode?
- 7) Hvad er en returtype?
- 8) Hvad er en parameter?
- 9) Hvad er forskellen på `==` og `equals()`?
- 10) Hvad er der specielt ved strenge i forhold til de andre objekter, du har set?

## 3.8 Resumé

I slutningen af de fleste kapitler kommer et resumé. Hvis du forstår resuméet (og har lavet nogle opgaver), har du fat i det væsentlige i dette kapitel.

- 1) Et objekt er en samling af data og programkode, der repræsenterer en ting i programmet, f.eks. et punkt, et rektangel, en streng, en liste, en dato.
- 2) En klasse er en beskrivelse af en type objekter. Alle strenge tilhører således `String`-klassen og alle punkter `Point`-klassen. Klassen kan opfattes, som en abstrakt beskrivelse af objekterne, en slags skabelon eller støbeform, ud fra hvilken vi kan oprette konkrete objekter. Objekter fra samme klasse har de samme variabler og metoder.
- 3) En variabel oprettes (defineres) ved at skrive dens type og dens navn, f.eks. `Point p`;
- 4) Et objekt oprettes i hukommelsen ved at skrive `new` og typen efterfulgt af parenteser med eventuelle parametre, f.eks. `new Point(10,10)`;
- 5) Et objekt har variabler, der husker dets data. Nogle objekter (f.eks. `Point` og `Rectangle`) giver adgang til at ændre disse variabler udefra, men for de fleste objekters vedkommende skal man kalde metoder, der ændrer deres data.
- 6) En metode er en kommando eller et spørgsmål til et objekt.
- 7) Returtypen på en metode angiver typen af svar, som kalderen får, f.eks. `int` eller `String`. Metoder med returtypen `'void'` er kommandoer, der ikke returnerer noget.
- 8) Når man kalder en metode, skal man ofte give nogle ekstra oplysninger med kaldet. De kaldes parametre og angives i parenteser efter metodenavnet.
- 9) Forskellen på `==` og `equals()` er, at `(x==y)` undersøger, om variablerne `x` og `y` refererer til det samme objekt, mens `(x.equals(y))` undersøger, om `x` og `y`'s objekter indeholder de samme data.
- 10) Streng-objekter opfører sig ikke som de fleste andre slags objekter: De ændrer sig ikke, når de først er skabt, deres metoder giver i stedet nye strenge tilbage, der er modificerede versioner af strengen. Man behøver i øvrigt ikke bruge `new String("xx")`, fordi en tekst i `""`-tegn allerede er et streng-objekt (og det er aldrig nødvendigt med en kopi).
- 11) Et vigtigt supplement til en lærebog er javadokumentationen over klasserne i standardbiblioteket. Den kan findes på <http://java.sun.com/javase/6/docs/api/>, men følger også med de fleste udviklingsværktøjer.

## 3.9 Appendiks

Her finder du en oversigt over de vigtigste klasser og deres vigtigste metoder. På adressen <http://java.sun.com/javase/6/docs/api/> findes en komplet oversigt.

### 3.9.1 Klassen Point

Point repræsenterer et punkt med en x- og y-koordinat.

*java.awt.Point – punkter med en x- og y-koordinat – skal importeres med import java.awt.\*;*

#### Variable

int x	x-koordinaten
int y	y-koordinaten

#### Konstruktører

Point()  
Opretter et punkt i (0,0).

Point(int x, int y)  
Opretter et punkt i (x,y).

Point(Point p)  
Opretter et punkt med samme (x,y)-koordinater som p.

#### Metoder

void **move**(int x, int y)  
Sætter punktets koordinater.

void **translate**(int x, int y)  
Rykker punktets koordinater relativt i forhold til, hvor det er.

double **distance**(Point etAndetPunkt)  
Giver afstanden fra punktet til *etAndetPunkt*.

boolean **equals**(Object obj)  
Undersøger, om punktet har samme koordinater som *obj*. Returnerer true, hvis det er tilfældet og false, hvis obj ikke er et punkt eller hvis det har andre koordinater.

String **toString**()  
giver en strengrepræsentation af punktet med (x,y)-koordinater, f.eks. java.awt.Point[x=0,y=0]



## 3.9.2 Klassen Rectangle

Rectangle repræsenterer et todimensionalt rektangel.

*java.awt.Rectangle – todimensionalt rektangel – skal importeres med import java.awt.\*;*

### Variable

int x	x-koordinat på øverste venstre hjørne
int y	y-koordinat på øverste venstre hjørne
int width	bredden
int height	højden

### Konstruktører

Rectangle()

opretter et rektangel i (0,0), hvis bredde og højde er 0.

Rectangle(int bredde, int højde)

opretter et rektangel i (0,0) med den angivne bredde og højde.

Rectangle(int x, int y, int bredde, int højde)

opretter et rektangel i (x,y) med den angivne bredde og højde.

Rectangle(Point p)

opretter et rektangel i *p*, hvis bredde og højde er 0.

### Metoder

void **add** (Point p)

udvider rektanglet sådan, at det også omfatter punktet *p*.

void **translate** (int x, int y)

rykker rektanglets koordinater relativt i forhold til, hvor det er.

boolean **contains** (Point p)

returnerer true, hvis *p* er inden for rektanglet, ellers false.

boolean **intersects** (Rectangle r)

returnerer true, hvis rektanglet og *r* overlapper.

Rectangle **intersection** (Rectangle r)

undersøger overlappet (fællesmængden, snitmængden) mellem rektanglet og *r* og returnerer et rektangel, der repræsenterer det fælles overlap.

Rectangle **union**(Rectangle r)

returnerer et rektangel, der repræsenterer foreningsmængden, dvs. det mindste rektangel, der indeholder både *r* og dette rektangel.

boolean **equals**(Object obj)

Undersøger, om rektanglet har samme koordinater og mål som *obj*. Returnerer true, hvis det er tilfældet og false, hvis *obj* ikke er et rektangel, eller hvis det har andre koordinater eller mål.

String **toString**()

giver en strengrepræsentation af rektanglet med (x,y)-koordinater og mål.

### 3.9.3 Klassen String

Strengene er specielle ved, at de ikke kan ændres, når de først er oprettet.

*java.lang.String – tekststreng*

<code>char <b>charAt</b> (int indeks)</code>	returnerer tegnet på det angivne <i>indeks</i> . Indeks tæller fra 0.
<code>String <b>replace</b> (String søgetekst, String erstatning)</code>	Giver en ny streng, som er identisk med denne streng, bortset fra at alle steder i strengen, hvor teksten <i>søgetekst</i> står, er blevet erstattet med teksten <i>erstatning</i> .
<code>String <b>substring</b> (int startindeks)</code>	Returnerer en ny streng, som er en del af denne streng. Delstrengen starter ved <i>startindeks</i> og går til slutningen.
<code>String <b>substring</b> (int startindeks, int slutindeks)</code>	Returnerer en ny streng, som er en del af denne streng. Delstrengen starter ved <i>startindeks</i> og slutter ved <i>slutindeks</i> (til og med <i>slutindeks</i> -1).
<code>String[] <b>split</b> (String regulærtUdtryk)</code>	Deler strengen op efter skilletegnene i <i>regulærtUdtryk</i> og returnerer et streng-array med bidderne. Et par eksempler på brug kan findes sidst i afsnit 3.10.5 og i afsnit 15.3.1.
<code>String <b>toLowerCase</b> ()</code>	returnerer en ny streng, som er identisk med denne streng, bortset fra at alle store bogstaver er erstattet med små.
<code>String <b>toUpperCase</b> ()</code>	returnerer en ny streng, som er identisk med denne streng, bortset fra at alle små bogstaver er erstattet med store.
<code>String <b>trim</b> ()</code>	returnerer en ny streng, som er identisk med denne streng, bortset fra at alle blanktegn, tabulatortegn, linjeskift etc. er fjernet fra begge ender af strengen.
<code>int <b>length</b> ()</code>	returnerer længden af (antal tegn i) strengen.
<code>int <b>indexOf</b> (String str)</code>	returnerer indekset på den første forekomst af <i>str</i> som delstreng. Hvis <i>str</i> ikke er en delstreng, returneres -1.
<code>int <b>lastIndexOf</b> (String str)</code>	returnerer indekset på den sidste forekomst af <i>str</i> som delstreng. Hvis <i>str</i> ikke er en delstreng, returneres -1.
<code>boolean <b>startsWith</b> (String str)</code>	returnerer sand, hvis denne streng starter med de samme tegn som <i>str</i> , ellers falsk.
<code>boolean <b>endsWith</b> (String str)</code>	returnerer sand, hvis denne streng slutter med de samme tegn som <i>str</i> , ellers falsk.
<code>boolean <b>equals</b> (String str)</code>	returnerer sand, hvis denne streng har samme indhold som <i>str</i> , ellers falsk.
<code>boolean <b>equalsIgnoreCase</b> (String str)</code>	returnerer sand, hvis denne streng har samme indhold som <i>str</i> , ellers falsk. Der skelnes ikke mellem store og små bogstaver.

To strenge *s1* og *s2* sammenlignes, ved at kalde *s1.equals(s2)*, ikke med *s1==s2* (der sammenligner objektreferencer, j.v.f. afsnit 3.4.5).

### 3.9.4 Specialtegn i strenge

Visse tegn kan man ikke skrive direkte i tekststrenger i kildeteksten. De er opført herunder:

Ønsket tegn	I kildeteksten skrives
Tabulator	\t
Linjeskift (eng.: newline)	\n
Vognretur (eng.: carriage return). Bruges sjældent	\r
Bak (eng.: backspace). Bruges sjældent	\b
Anførelsestegn "	\"
Apostrof '	\'
En bagstreg \ (eng.: backslash)	\\
Unicode-tegn nummer XXXX	\uXXXX

Unicode-tegn skrives som fire hexadecimale cifre (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F). F.eks. kan de græske bogstaver findes på "\u0390" til "\u03F3" (det græske pi skrives med "\u03C0") og de kyrilliske tegn findes fra "\u0400" til "\u047F". Unicode-tegntabellen kan findes på <http://unicode.org>

Eksempler:

```
System.out.println("Jacob\n\n\n\tNordfaaa\b\blk");
```

Jacob

Nordfalk

```
System.out.println("\t\"At være eller ikke være\n\tDet er spørgsmålet.\");
```

```
System.out.println("\t\t\t\t\t\t\t\t\t\tShakespeare");
```

"At være eller ikke være  
det er spørgsmålet."

Shakespeare

## 3.9.5 Klassen Date

Date repræsenterer en tid, dvs. en dato og et klokkeslæt. Se eksempler på brug i afsnit 3.6.

*java.util.Date – et tidspunkt – skal importeres med import java.util.\*;*

### Konstruktører

Date ()

opretter et Date-objekt, som repræsenterer tidspunktet, da det blev oprettet.

Date (long tid\_i\_millisekunder)

opretter et Date-objekt, som repræsenterer et tidspunkt, der er *tid\_i\_millisekunder* efter 1. januar 1970 kl 00:00:00 GMT.

Date (int år, int måned, int dag, int timer, int minutter)

(frarådes)

opretter et Date-objekt med det givne tidspunkt. Bemærk, at *år* regnes fra år 1900 (1997 svarer til *år=97*) og *måned* regnes fra 0 (januar svarer til *måned=0*).

Date (String dato)

(frarådes)

opretter et Date-objekt, som repræsenterer det tidspunkt, *dato* indeholder.

### Metoder

long **getTime** ()

returnerer antal millisekunder siden 1. januar 1970 kl. 00:00:00 GMT repræsenteret af dette Date-objekt.

void **setTime** (long tid\_i\_millisekunder)

ændrer dette Date-objekt til at repræsentere et tidspunkt, der er *tid\_i\_millisekunder* efter 1. januar 1970 kl 00:00:00 GMT.

boolean **after** (Date hvornår)

undersøger, om denne dato er efter *hvornår*-datoen.

boolean **before** (Date hvornår)

undersøger, om denne dato er før *hvornår*-datoen.

String **toString** ()

returnerer en strengrepræsentation af formen: ugedag mm dd tt:mm:ss åååå (f.eks. Man 5. juli 15:23:18 2000). Denne metode kaldes automatisk, hvis man forsøger at lægge en dato sammen med en streng med +-operatoren.

int **getDate** ()

(frarådes)

returnerer dagen i måneden repræsenteret af dette objekt.

void **setDate** (int dag).

(frarådes)

ændrer dagen i måneden til *dag*. Sættes den til en dag uden for denne måned, opfattes det som om måneden skal ændres tilsvarende, f.eks. vil setDate(32) gå ind i næste måned.

Tilsvarende findes **getYear()**, **setYear()**, **getMonth()**, **setMonth()**, **getHours()**, **setHours()**, **getMinutes()**, **setMinutes()**, **getSeconds()** og **setSeconds()**.

(frarådes)

(frarådes): Disse metoder blev frarådet fra JDK version 1.1, fordi de ikke understøtter andre kalendere end det gregorianske kalendersystem, der bruges i den vestlige verden. Det betyder dog ikke det store for europæiske programmer.

## 3.9.6 Klassen ArrayList

ArrayList er en liste af andre objekter. Se afsnit 3.5 for eksempler på brug af ArrayList.

*java.util.ArrayList – en liste af objekter – skal importeres med import java.util.\*;*

### Konstruktører

ArrayList<Elementtype> ()

opretter en tom liste med elementer af klassen *Elementtype*. <Elementtype> kan udelades.

### Metoder

void **add**( Elementtype element )

føjer *element* til slutningen af listen. Hvis Elementtype var angivet i konstruktøren skal *element* være af denne type. Elementtyper som Integer eller Double tillader også *autoboxing* af den tilsvarende simple type (int hhv. double), dvs. liste=new ArrayList<Integer>() tillader liste.add(5).

void **add**( int indeks , element )

indsætter *element* i listen lige før plads nummer *indeks*. Første element er på plads nummer 0.

Elementtype **remove**( int indeks )

fjerner elementet på plads nummer *indeks* og returnerer elementet, der blev fjernet.

boolean **isEmpty**()

returnerer sand, hvis listen er tom (indeholder 0 elementer).

int **size**()

returnerer antallet af elementer i listen.

Elementtype **get**(int indeks)

returnerer en reference til objektet på plads nummer *indeks*. Har du udeladt <Elementtype> i konstruktøren skal du huske at lave en typekonvertering af resultatet som beskrevet i afsnit 3.5.4.

boolean **contains**( objekt )

returnerer sand, hvis *objekt* findes i listen.

int **indexOf**( objekt )

returnerer indekset på første forekomst af *objekt* i listen. Hvis den ikke findes, returneres -1.

String **toString** ()

returnerer listens indhold som en streng. Dette sker ved at konvertere hver af elementerne til en streng.

## Autoboxing

Elementtyper som Integer eller Double tillader også *autoboxing* af den tilsvarende simple type. Det vil f.eks. sige, at man kan have en liste af den simple type int med:

```
ArrayList<Integer> liste = new ArrayList<Integer>();  
  
liste.add(8); // tilføj element '8' på første plads  
  
int nummer = liste.get(0); // aflæs 8-tallet i listen
```

Et eksempel findes i afsnit 3.10.2.

Bemærk at selvom det kunne virke logisk, kan man altså *ikke* skrive f.eks. ArrayList<int> for at oprette en liste af int, man *skal* angive en objekttype som f.eks. ArrayList<Integer>.

## 3.10 Avanceret

Når det foregående i kapitlet er nogenlunde forstået, kan du læse videre for nogle flere fif.

Da de avancerede afsnit i slutningen af hvert kapitel er beregnet til at blive læst senere, kan der godt blive brugt nogle begreber, der først behandles senere i bogen.

### 3.10.1 Sætte strenge sammen (klassen `StringBuilder`)

Hvis man skal manipulere meget med strenge, bliver det hæmmende og langsommeligt, at `String`-objekter ikke tillader, at man ændrer i dem. Derfor findes der en speciel klasse, `StringBuilder`, hvor man godt kan ændre i indholdet.

`StringBuilder`-objekter kan blive vilkårligt lange. De sørger selv for, at reservere mere hukommelse, hvis det bliver nødvendigt.

*java.lang.StringBuilder – tekststrenge, der kan ændres i.*

#### Konstruktører

`StringBuilder ()`

opretter en tom `StringBuilder`.

`StringBuilder (String str)`

opretter et `StringBuilder`-objekt med samme tegnsekvens som *str*.

`StringBuilder (int startkapacitet)`

opretter en `StringBuilder`, der har reserveret plads til et antal tegn på forhånd.

#### Metoder

`StringBuilder append ( ... )`

tilføjer parameteren i slutningen (og returnerer `StringBuilder`-objektet selv).

Parameteren kan være `int`, `float`, `double`, `boolean` eller andre simple typer eller `String`.

`StringBuilder insert (int pos, ... )`

indsætter parameteren på position *pos* (og returnerer `StringBuilder`-objektet selv).

`char charAt (int indeks)`

returnerer tegnet på plads nummer *indeks*.

`void setCharAt (int indeks, char tegn)`

sætter tegnet på plads nummer *indeks* til at være *tegn*.

`StringBuilder reverse ()`

sætter alle i omvendt rækkefølge (og returnerer `StringBuilder`-objektet selv).

`String toString ()`

returnerer bufferen som en streng. Denne metode kaldes automatisk, hvis man forsøger at lægge objektet sammen med en streng med `+`-operatoren.

`int length ()`

returnerer længden af strengen (i antal tegn).

`void setLength (int nyLængde)`

sætter længden af strengen til *nyLængde*.

Bemærk, at equals() er en del af den interne mekanik i Java og den findes på alle objekter.

---

### Selvom metoden findes, kan equals() *ikke* bruges til at sammenligne indholdet af to StringBuilder-objekter eller et StringBuilder-objekt og en streng

---

To StringBuilder-objekter sb1 og sb2 sammenlignes, ved at sammenligne strengene, som de repræsenterer: sb1.toString().equals( sb2.toString() ).

I eksemplet herunder omformulerer vi et citat og skriver det derefter baglæns. Bemærk, at vores StringBuilder-objekt sb ændrer indhold undervejs.

```
// Viser brugen af StringBuilder-klassen og dens metoder.
public class BenytStringBuilder {
    public static void main (String[] arg)
    {
        StringBuilder sb;
        sb = new StringBuilder("At være");
        sb.append(" eller ");
        sb.append("ikke være");
        System.out.println("sb er nu: "+sb);
        sb.insert(8,"I FRED ");
        System.out.println("sb er nu: "+sb);
        sb.reverse();
        System.out.println("sb er nu: "+sb);
        sb.setLength(2);
        System.out.println("sb er nu: "+sb);
    }
}

sb er nu: At være eller ikke være
sb er nu: At være I FRED eller ikke være
sb er nu: eræv ekki relle DERF I eræv tA
sb er nu: er
```

## Optimering

Skal du sætte mange strenge sammen, bør du bruge en StringBuilder, da du så undgår, at oprette de mange midlertidige String-objekter, der ellers ville opstå. Følgende program demonstrerer den enorme hastighedsforskel, en StringBuilder kan gøre.

```
// Demonstrerer hastighedsforskellen mellem String og StringBuilder
// ved sammensætning af mange strenge
public class HastighedsforskelMedStringBuilder
{
    public static void main (String[] arg)
    {
        long tid1 = System.currentTimeMillis();
        String s = "";
        for (int i=0; i<10000; i++) s = s + "x";    // her oprettes 10000 objekter

        long tid2 = System.currentTimeMillis();
        System.out.println("Antal sekunder med String: " + (tid2-tid1)*0.001 );

        StringBuilder sb = new StringBuilder(10000); // reservér plads til 10000 tegn
        for (int i=0; i<10000; i++) sb.append("x"); // her ændres i det samme objekt
        s = sb.toString();

        long tid3 = System.currentTimeMillis();
        System.out.println("Antal sek med StringBuilder: " + (tid3-tid2)*0.001 );
    }
}

Antal sekunder med String: 1.067
Antal sek med StringBuilder: 0.0080
```

Her sparer man altså over en faktor 100 i kørselstid (0.008 sekunder i stedet for 1 sekund).

Besparselsen skyldes, at der oprettes færre objekter (1 enkelt StringBuilder i stedet for 10000 strenge). Oprettelse (og oprydning af) objekter er en forholdsvis tidskrævende operation.

## 3.10.2 Standardmetoder til at arbejde med lister

Der findes faktisk en metode, der kan udføre blandingen af kortene i afsnit 3.5.5 for os:

```
// Bland bunken
Collections.shuffle(bunke)
```

Man kan spare rigtig meget tid, ved at kende til mulighederne i standardbiblioteket. Her er et eksempel, der viser nogle andre af faciliteterne i standardbiblioteket:

```
import java.util.*;
public class StandardmetoderTilLister
{
    public static void main(String[] args)
    {
        ArrayList<Integer> liste = new ArrayList<Integer>();
        liste.add(8); // tilføj ét element
        Collections.addAll(liste, 1, 2, 3, 4, 5, 6, 3, 3, 3, 9); // tilføj mange elem
        System.out.println("Listen indeholder: " + liste);

        Collections.sort(liste);
        System.out.println("Listen sorteret: " + liste);

        Collections.reverse(liste);
        System.out.println("Listen baglæns: " + liste);

        Collections.shuffle(liste);
        System.out.println("Listen blandet igen: " + liste);

        System.out.println("Største tal i listen: " + Collections.max(liste));

        System.out.println("Antal 3-taller : " + Collections.frequency(liste, 3));

        ArrayList<Integer> liste2 = new ArrayList<Integer>();
        liste2.addAll(liste); // tilføj alle elementer fra liste til liste2
        System.out.println("Kopien liste2 indeh: " + liste2);

        liste.subList(0, 5).clear(); // fjern element 0 til 5 fra liste
        System.out.println("Liste uden elem 0-5: " + liste);

        liste2.removeAll(liste); // fjern element 5 til 9 fra liste2
        System.out.println("Liste2 u. andre elem: " + liste2);
    }
}
```

```
Listen indeholder: [8, 1, 2, 3, 4, 5, 6, 3, 3, 3, 9]
Listen sorteret: [1, 2, 3, 3, 3, 3, 4, 5, 6, 8, 9]
Listen baglæns: [9, 8, 6, 5, 4, 3, 3, 3, 3, 2, 1]
Listen blandet igen: [5, 3, 8, 1, 3, 4, 9, 2, 3, 6, 3]
Største tal i listen: 9
Antal 3-taller : 4
Kopien liste2 indeh: [5, 3, 8, 1, 3, 4, 9, 2, 3, 6, 3]
Liste uden elem 0-5: [4, 9, 2, 3, 6, 3]
Resterende unikke : [5, 8, 1]
```

## 3.10.3 Lister af simple typer (autoboxing)

Eksemplet ovenfor viser, hvordan man kan lave lister med simple typer som int og double. Egentligt arbejder ArrayList kun med objekter og tillader ikke simple typer som elementer. Java konverterer automatisk mellem en simpel type og den tilsvarende klasse. Bemærk at lister skal erklæres med objekttypen (f.eks. Integer), ikke den simple type (f.eks. int):

```
ArrayList<Integer> liste; // rigtigt
ArrayList<int> liste; // FORKERT
```

En anden mulighed er at bruge arrays, se kapitel 8.

## 3.10.4 Andre slags lister og mængder

Udover ArrayList findes en række andre smarte liste- og mængdeklasser og metoder, der er uvurderlige hvis man arbejder meget med større datamængder. De er beskrevet i kapitel 1 i bogen "Videregående programmering i Java", der kan læses på <http://javabog.dk/VP/>.



### 3.10.5 Nøgleindekserede tabeller (klassen HashMap)

En hashtabel er en tabel, der afbilder nogle nøgler til værdier. Hashtabeller er nyttige som associative afbildninger – når man vil indekserer nogle objekter ud fra f.eks. navne.

Før vi går videre, så bemærk lige, at lister *går fra heltal til objekter*, dvs. man finder listens elementer frem ud fra et helt tal (indekset). Blandt andet har en liste metoderne:

```
void add( int indeks , element )  
    Indsætter element i listen lige før plads nummer indeks. Første element er på plads nummer 0.  
Elementtype get(int indeks)  
    returnerer en reference til objektet på plads nummer indeks.
```

Man kan sige, at elementerne i en liste indekseres (fremfindes) ud fra et *tal*.

Hashtabeller *går fra objekter til objekter* på den måde, at til hvert element knyttes et nøgle-objekt. Elementerne kan derefter findes frem ud fra nøglerne.

Man kan altså sige, at elementerne i en hashtabel indekseres (fremfindes) ud fra et *objekt*.

*java.util.HashMap – nøgleindekseret tabel af objekter*

#### Konstruktører

HashMap<Nøgletype, Elementtype> ()  
 opretter en tom tabel hvor nøglerne er af klassen *Nøgletype* og værdierne af klassen *Elementtype*.  
 <Nøgletype, Elementtype> kan udelades.

#### Metoder

void **put** (Nøgletype nøgle, Elementtype værdi)  
 føjer objektet *værdi* til hashtabellen under objektet *nøgle*.

Elementtype **get** (Nøgletype nøgle)  
 slår op under *nøgle* og returnerer den værdi, der findes der (eller null hvis nøglen ikke kendes).

Elementtype **remove**(Nøgletype nøgle)  
 fjerner indgangen under *nøgle* og returnerer værdien (eller null hvis nøglen ikke kendes).

boolean **isEmpty**()  
 returnerer sand, hvis tabellen er tom (indeholder 0 indgange).

int **size**()  
 returnerer antallet af indgange.

boolean **containsKey**(Nøgletype nøgle)  
 returnerer sand, hvis *nøgle* findes blandt nøglerne i tabellen.

boolean **containsValue**(Elementtype værdi)  
 returnerer sand, hvis *værdi* findes blandt værdierne i tabellen.

Set<Nøgletype> **keySet**()  
 giver alle nøglerne. Kan bruges til at gennemløbe alle indgangene (se nedenfor).

String **toString** ()  
 giver tabellens indhold som en streng. Dette sker ved at konvertere hver af indgangenes nøgler og værdier til strenge.

Herunder opretter vi en tabel, der holder styr på fødselsdatoer for et antal personer med deres fornavne som nøgler, med put()-metoden: put("Jacob", dato). Derefter kan indgangene hentes tilbage igen med get()-metoden: get("Jacob") giver Jacobs fødselsdato.

Sidst gennemløbes alle indgangene. Vi bruger en for-løkke til at løbe gennem tabellens nøgler (med hashtabel.keySet()), hvorefter vi slår de tilsvarende værdier op.

```

import java.util.*;
public class BenytHashMap
{
    public static void main(String[] arg)
    {
        // En tabel med strenge som nøgler og Date-objekter som værdier
        HashMap<String,Date> hashtabel = new HashMap<String,Date>();
        Date dato = new Date(71,0,1); // 1. januar 1971
        hashtabel.put("Jacob",dato);
        hashtabel.put("Troels",new Date(72,7,11)); // 11. august 1972
        hashtabel.put("Eva",new Date(73,2,5));
        hashtabel.put("Ulla",new Date(69,1,9));
        System.out.println( "tabel indeholder: "+hashtabel );

        // Lav nogle opslag i tabellen under forskellige navne
        dato = hashtabel.get("Troels");
        System.out.println( "Opslag under 'Troels' giver: "+dato);
        System.out.println( ".. og under Jacob: "+hashtabel.get("Jacob"));
        System.out.println( ".. Kurtbørge: "+hashtabel.get("Kurtbørge"));
        System.out.println( ".. Eva: "+hashtabel.get("Eva"));

        // Gennemløb af alle elementer
        for (String nøgle : hashtabel.keySet()) {
            dato = hashtabel.get(nøgle);
            System.out.println(nøgle + "'s fødselsår: "+dato.getYear());
        }
    }
}

```

---

```

tabel indeholder: {Jacob=Fri Jan 01 00:00:00 CET 1971, Troels=Fri Aug 11 00:00:00
CET 1972, Eva=Mon Mar 05 00:00:00 CET 1973, Ulla=Sun Feb 09 00:00:00 CET 1969}
Opslag under 'Troels' giver: Fri Aug 11 00:00:00 CET 1972
.. og under Jacob: Fri Jan 01 00:00:00 CET 1971
.. Kurtbørge: null
.. Eva: Mon Mar 05 00:00:00 CET 1973
Jacob's fødselsår: 71
Troels's fødselsår: 72
Eva's fødselsår: 73
Ulla's fødselsår: 69

```

En hashtabel husker ikke rækkefølgen af indgangene. Derfor er rækkefølgen, som elementerne bliver udskrevet i, ikke den samme som den rækkefølge, de blev sat ind i.

Her er en lille esperanto-dansk-ordbog. Nøglerne er esperanto og værdierne er danske ord:

```

import java.util.*;
public class BenytHashMapOrdbog
{
    public static void main(String[] args)
    {
        HashMap<String,String> ord = new HashMap<String,String>();
        ord.put("granda", "stor");
        ord.put("longa", "lang");
        ord.put("bela", "smukt");
        ord.put("estas", "er");

        String esperantotekst = "longa, granda hundo estas.... bela!";

        for (String eoOrd : esperantotekst.split("\\b")) { // split efter ordgrænser
            String da = ord.get( eoOrd ); // slå esperantoord op og få det danske ord
            if (da == null) da=eoOrd; // hvis intet fundet lader vi det stå uoversat
            System.out.print( da );
        }
    }
}

```

---

```

lang, stor hundo er.... smukt!

```

Har vi en tekst på esperanto, kan vi nu oversætte teksten ord for ord til dansk. Hvert ord slås op i hashtabellen og hvis det findes, erstattes det med det danske ord. Tegn og ord, som ikke kan findes i tabellen (såsom "hundo", der betyder "hund"), efterlades uforandret.

# 4 Definition af klasser

Indhold:

- Definere egne klasser (typer af objekter)
- Definere konstruktører, objektvariabler og metoder
- Definition af klasserne Boks, Terning, Raflebæger, Person og Konto
- Nøgleordet this

Kapitlet forudsættes i resten af bogen.

Forudsætter kapitel 3, Objekter.

Er man i gang med et større program, vil man have brug for at definere sine egne specialiserede klasser. Et regnskabsprogram kunne f.eks. definere en Konto-klasse. Ud fra Konto-klassen ville der blive skabt et antal konto-objekter, svarende til de konti, der skulle administreres.

I dette kapitel ser vi på, hvordan man selv definerer sine egne klasser. Vi minder om, at

---

**En klasse er en skabelon, som man kan danne objekter ud fra**

**Klassen beskriver variabler og metoder, der kendetegner objekterne**

**Et objekt er en konkret forekomst (instans) af klassen**

---

Når man programmerer objektorienteret, samler man data i selvstændige objekter og definerer metoder, som arbejder på disse data i objekterne.

## 4.1 En Boks-klasse

Lad os tage et eksempel på en klassedefinition:

<i>klassenavn:</i>	<b>Boks</b>
<i>variabler:</i>	<code>længde :double</code> <code>bredde :double</code> <code>højde :double</code>
<i>metoder:</i>	<code>volumen() :double</code>

Vi definerer klassen Boks, som indeholder tre variabler, nemlig `længde`, `bredde` og `højde`. Derudover definerer vi metoden `volumen()`, som arbejder på disse data. Metoden returnerer en `double` og har ingen parametre.

```
public class Boks
{
    double længde;
    double bredde;
    double højde;

    double volumen()
    {
        double vol;
        vol = længde*bredde*højde;
        return vol;
    }
}
```

### 4.1.1 Variabler

Variablerne `længde`, `bredde` og `højde` kaldes også objektvariabler, fordi hvert Boks-objekt har én af hver.

---

**Objektvariabler erklæres direkte i klassen uden for metoderne<sup>1</sup>**

---

Vi kan lave et Boks-objekt, `boksobjekt` med `new`:

```
Boks boksobjekt;
boksobjekt = new Boks();
```

---

<sup>1</sup> Vi vil senere se, at såkaldte klassevariabler (static) også erklæres her.

Nu er der oprettet et Boks-objekt i lageret, der således har en længde-, en bredde- og en højde-variabel (hver med værdien 0).

Variablen `vol` kaldes en lokal variabel, fordi den er erklæret lokalt i `volumen()`-metoden.

---

**En variabel erklæret inde i en metode kaldes en lokal variabel**

**Lokale variable eksisterer kun, så længe metoden, hvori de er erklæret, udføres**

---

Modsat længde, bredde og højde begynder `vol`-variabler altså ikke at eksistere, bare fordi vi har skabt en Boks<sup>1</sup>.

## 4.1.2 Brug af klassen

Objekter af klassen Boks kan f.eks. benyttes på følgende måde:

```
public class BenytBoks
{
    public static void main(String[] arg)
    {
        double rumfang;

        Boks boksobjekt;
        boksobjekt = new Boks();
        boksobjekt.længde = 12.3;
        boksobjekt.bredde = 2.22;
        boksobjekt.højde = 6.18;
        rumfang = boksobjekt.volumen();
        System.out.println("Boksens volume: " + rumfang);
    }
}
```

Boksens volume: 168.75108

Som det ses, er det klassen `BenytBoks`, der indeholder `main()`-metoden. Der skal være én og kun én klasse med en `main()`-metode i et program. En sådan "main-klasse" bruges ikke til at definere objekttyper med – kun til at angive, hvor programmet skal startes<sup>2</sup>.

I følgende sætninger (i klassen `BenytBoks`) sættes det nyoprettede Boks-objekts variable:

```
boksobjekt.længde = 12.3;
boksobjekt.bredde = 2.22;
boksobjekt.højde = 6.18;
```

I den efterfølgende sætning:

```
rumfang = boksobjekt.volumen();
```

kaldes metoden `volumen()` i Boks-objektet, der udregner rumfanget ud fra variableerne, som er blevet tilført data i linjerne ovenfor. Metoden returnerer en `double` – denne lægges over i variabelen `rumfang`, som udskrives.

## 4.1.3 Metodedefinition

Når vi definerer en metode, giver vi den et hoved og en krop.

**Hovedet** ligner den måde, vi tidligere har set metoder opremset på. Metodehovedet fortæller metodens navn, returtype og hvilke parametre metoden eventuelt har:

```
double volumen()
```

- 
- 1 Når programudførelsen går ind i metoden, oprettes de lokale variable. De nedlægges igen, når metoden returnerer.
  - 2 Man kan godt lave en klasse, der både har en `main()`-metode og som der oprettes objekter ud fra, men det kan være forvirrende for en begynder at blande de to ting sammen.

**Kroppen** kommer lige under hovedet:

```
{
    double vol;
    vol = længde*bredde*højde;
    return vol;
}
```

I kroppen står der, hvad der skal ske, når metoden kaldes. Her står altså, at når metoden `volumen()` kaldes, bliver der først oprettet en lokal variabel, `vol`. Denne bliver tildelt produktet af de tre variabler `længde`, `bredde` og `højde`. Den sidste linje i kroppen fortæller, at værdien af `vol` bliver givet tilbage (returneret) til der, hvorfra metoden blev kaldt.

---

### En metodekrop udføres, når metoden kaldes

---

Variablerne `længde`, `bredde` og `højde`, som kroppen bruger, er dem, der findes i netop det objekt, som `volumen()`-metoden blev kaldt på.

Lad os kigge på en stump af `BenytBoks`:

```
boksobjekt.længde= 12.3;
boksobjekt.bredde= 2.22;
boksobjekt.højde= 6.18;
rumfang = boksobjekt.volumen();
```

Her får det `Boks`-objekt, som `boksobjekt` refererer til, sat sine variabler og når `volumen()` derefter kaldes på objektet, vil `længde`, `bredde` og `højde` have disse værdier. `rumfang` bliver sat til den værdi, `vol` har i return-linjen og `vol` nedlægges (da den er en lokal variabel).

---

### En return-sætning afslutter udførelsen af metodekroppen og leverer en værdi tilbage til kalderen

---

## 4.1.4 Flere objekter

Herunder opretter vi to bokse og udregner deres forskel i rumfang. Hver boks er et aftryk af `Boks`-klassen forstået på den måde, at de hver indeholder deres egne sæt variabler. Variablen `bredde` kan således have forskellige værdier i hvert objekt.

```
public class BenytBokse
{
    public static void main(String[] arg)
    {
        Boks boks1, boks2;
        boks1 = new Boks();
        boks2 = new Boks();

        boks1.længde= 12.3;
        boks1.bredde= 2.22;
        boks1.højde= 6.18;

        boks2.længde= 13.3;
        boks2.bredde= 3.33;
        boks2.højde= 7.18;

        double v1, v2;

        v1 = boks1.volumen();
        v2 = boks2.volumen();

        System.out.println("Volumenforskel: "+ (v2 - v1));
    }
}
```

---

Volumenforskel: 149.24394

Når vi kalder `volumen()` på `boks1` og `boks2`, er det således to forskellige sæt `længde`-, `højde`- og `bredde`-variabler, der bliver brugt til beregningen, når `volumen()`'s krop udføres.

## 4.2 Indkapsling

Indkapsling (eng.: encapsulation) af data og metoder i objekter betyder, at man ikke lader andre bruge objekterne helt efter eget for godt/befindende. Man gør visse dele af objekterne utilgængelige uden for klassens metoder. Herved sætter man nogle regler op for, hvordan man kan benytte objekterne.

Hvorfor overhovedet indkapsle (skjule) variable?

Indkapsling i klasser er vigtig, når programmerne bliver store og komplekse. Hvis det er muligt at ændre data i en klasse, kan det føre til situationer, som kommer ud af kontrol i store komplekse systemer. Når data er indkapslet kan de ikke ændres direkte udefra, og man må i stedet definere metoder til at ændre i data på. I metoderne kan man sikre sig mod vanvittige overgreb på data ved at tilføje logik, der sikrer, at variablene er konsistente.

I ovenstående eksempel kan man for eksempel sætte højden af en boks til et negativt tal. Spørger man derefter på volumen(), vil man få et negativt svar! Det kræver ikke meget fantasi at forestille sig, hvordan sådanne fejl kunne gøre et program ubrugeligt. Tænk for eksempel på pakkepost-omdeling, hvis et af Post Danmarks programmer påstod, at der nemt kunne være 10000 pakker på hver *minus* en kubikmeter og 10001 pakker på hver plus en kubikmeter i én postvogn... endda med flere kubikmeter til overs til anden post!

Med indkapsling opnår man, at objekterne altid er konsistente, fordi objekterne selv sørger for, at deres variable har fornuftige værdier.

Man styrer adgangen til en variabel eller metode med nøgleordene `public` og `private`<sup>1</sup>:

---

**public** betyder "adgang for alle"

**private** betyder "kun adgang fra samme klasse"

---

Herunder ses en modificeret version af eksemplet med Boks- og BenytBoks-klassen, men nu er variablene erklæret `private`.

```
public class Boks2
{
    private double længde;
    private double bredde;
    private double højde;

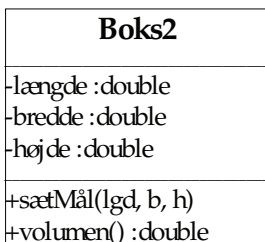
    public void sætMål(double lgd, double b, double h)
    {
        if (lgd<=0 || b<=0 || h<=0)
        {
            længde = 10.0;
            bredde = 10.0;
            højde = 10.0;
            System.out.println("Ugyldige mål. Bruger standardmål.");
        } else {
            længde = lgd;
            bredde = b;
            højde = h;
        }
    }

    public double volumen()
    {
        double vol;
        vol = længde*bredde*højde;
        return vol;
    }
}
```

---

1 Skriver man ingenting, svarer det til `public` (inden for samme pakke – se afsnit 6.9).

Figuren illustrerer klassen i UML-notationen. Bemærk, at variablerne er private, så de har et - foran, mens metoderne, som kan ses udefra (public), har et + foran:



Nu da variablerne længde, bredde og højde er erklæret private, er det ulovligt at ændre dem "udefra" i vores BenytBoks-program.

Til gengæld har vi defineret metoden sætMål(), som man skal kalde for at sætte variablerne.

Da den eneste måde at ændre data på er ved at kalde metoden sætMål(), kan vi der indlægge ønsket logik – for eksempel sikre os mod 0 (nul) eller negative værdier.

```
public class BenytBoks2
{
    public static void main(String[] arg)
    {
        Boks2 enBoks = new Boks2();

        //ulovligt: enBoks.længde= 12.3;
        //ulovligt: enBoks.bredde= 2.22;
        //ulovligt: enBoks.højde= 6.18;

        enBoks.sætMål( 2.0, 2.5, 1.5);

        System.out.println("Volumen er: " + enBoks.volumen());

        enBoks.sætMål(-2.0, 0.0, 1.0);

        System.out.println("Volumen er: " + enBoks.volumen());

        enBoks.sætMål( 2.0, 3.0 ,1.0);

        System.out.println("Volumen er: " + enBoks.volumen());
    }
}
```

---

```
Volumen er: 7.5
Ugyldige mål. Bruger standardmål.
Volumen er: 1000.0
Volumen er: 6.0
```

En anden fordel ved indkapsling er, at man bliver uafhængig af, hvordan data er repræsenteret internt. Man kunne f.eks. senere ændre Boks-klassen, så den kun lagrede volumen (beregnet allerede i sætMål()) i stedet for længde, bredde og højde.



## 4.3 Konstruktører

En konstruktør (eng.: constructor) er en speciel metode, der har samme navn som klassen. Den kaldes automatisk ved oprettelse af et objekt med 'new'-operatoren og benyttes oftest til at klare forskellige former for initialisering af det nye objekt.

Som vi så i forrige kapitel (i tilfældet med Rectangle, Point og Date), kan man have flere konstruktører for en klasse, bare parameterlisterne er forskellige.

Her kommer et eksempel<sup>1</sup> med nogle konstruktører:

```
/** En boks med en længde, bredde og højde */
public class Boks3
{
    private double længde;
    private double bredde;
    private double højde;

    /** konstruktør der opretter en standardboks */
    public Boks3()
    {
        System.out.println("Standardboks oprettes");
        længde = 10.0;
        bredde = 10.0;
        højde = 10.0;
    }

    /** en anden konstruktør der får bredde, højde og længde */
    public Boks3(double lgd, double b, double h)
    {
        System.out.println("Boks oprettes med lgd="+lgd+" b="+b+" h="+h);
        sætMål(lgd,b,h);
    }

    /** sætter boksens bredde, højde og længde */
    public void sætMål(double lgd, double b, double h)
    {
        if (lgd<=0 || b<=0 || h<=0)
        {
            System.out.println("Ugyldige mål. Bruger standardmål.");
            længde = 10.0;
            bredde = 10.0;
            højde = 10.0;
        }
        else {
            længde = lgd;
            bredde = b;
            højde = h;
        }
    }

    /** udregner boksens rumfang */
    public double volumen()
    {
        double vol = længde*bredde*højde ;
        return vol;
    }
}
```

Boks3
-længde :double -bredde :double -højde :double
+Boks3() +Boks3(lgd, b, h) +sætMål(lgd, b, h) +volumen() :double

Bemærk:

---

**En konstruktør erklæres som en metode med samme navn som klassen**

**En konstruktør har ingen returtype – ikke engang 'void'**

---

I ovenstående eksempel er der defineret to konstruktører:

```
public Boks3()
public Boks3(double lgd, double b, double h)
```

---

<sup>1</sup> Kommentarerne med `/**` og `*/` gør, at man automatisk kan generere dokumentation ud fra klassen (af samme form som standardklassernes dokumentation). Javadoc er behandlet i kapitel 2 i "Videregående programmering i Java", der kan læses på <http://javabog.dk/VP>.

Vi kan afprøve Boks3 med:

```
public class BenytBoks3
{
    public static void main(String[] arg)
    {
        Boks3 enBoks;
        enBoks = new Boks3();           // brug konstruktøren uden parametre
        System.out.println("Volumen er: " + enBoks.volumen());

        Boks3 enAndenBoks;
        enAndenBoks = new Boks3(5,5,10); // brug den anden konstruktør
        System.out.println("Volumen er: " + enAndenBoks.volumen());
    }
}
```

---

Standardboks oprettes  
Volumen er: 1000.0  
Boks oprettes med lgd=5.0 b=5.0 h=10.0  
Volumen er: 250.0

### 4.3.1 Standardkonstruktører

Når vi i de foregående eksempler (f.eks. Boks2) ikke har benyttet en konstruktør, er det fordi Java, hvis ikke en konstruktør er erklæret, selv erklærer en tom standardkonstruktør uden parametre. Dvs. Java i Boks2's tilfælde usynligt har defineret konstruktøren:

```
public Boks2()
{
}
```

Denne konstruktør har vi kaldt, hver gang vi har oprettet en boks med 'new Boks2()'.

---

**Der kaldes altid en konstruktør, når et objekt oprettes**

**Standardkonstruktøren genereres automatisk, hvis der ikke er andre konstruktører i klassen**

---

En standardkonstruktør genereres kun, hvis der ikke er andre konstruktører i klassen.

Hvis vi ikke havde defineret en konstruktør uden parametre i Boks3, ville oversætteren i BenytBoks3 brokke sig over, at denne type konstruktør ikke fandtes:

```
BenytBoks3.java:6: No constructor matching Boks3() found in class Boks3.
    enBoks = new Boks3();
```

### 4.3.2 Opgaver

- 1) Definér klassen Pyramide. Objekterne skal have variableerne side og højde (definér en konstruktør) og en metode til at udregne volumen (side\*side\*højde/4).  
Skriv en BenytPyramider, som opretter 3 pyramider og udregner volumen af dem.
- 2) Ret Boks3 til også at have variabelen massefylde og definér en ekstra konstruktør, der også får massefylden overført (den oprindelige konstruktør med lgd, b og h kan sætte massefylde til 1). Lav også metoder til at sætte massefylden, sætMassefylde(double m), og udregne vægten, vægt(). Afprøv, om det virker (test din klasse med en ændret udgave af BenytBoks3).

## 4.4 En Terning-klasse

Lad os tage et andet eksempel, en terning. Den vigtigste egenskab ved en terning er dens værdi (dvs. antallet af øjne på siden, der vender opad lige nu) mellem 1 og 6.

Terning
+værdi :int
+Terning() +kast() +toString():String

```
/** En klasse der beskriver 6-sidede terninger */
public class Terning
{
    /** antallet af øjne på den side på terningen, der vender opad lige nu */
    public int værdi;

    /** konstruktør der opretter en terning */
    public Terning()
    {
        kast(); // kald kast() der sætter værdi til noget fornuftigt
    }

    /** kaster terningen, så den får en anden værdi */
    public void kast()
    {
        // find en tilfældig side
        double tilfældigtTal = Math.random();
        værdi = (int) (tilfældigtTal * 6 + 1);
    }

    /** giver en beskrivelse af terningen som en streng */
    public String toString()
    {
        String svar = ""+værdi; // værdi som streng, f.eks. "4"
        return svar;
    }
}
```

Her er et program, der bruger et Terning-objekt til at slå med, indtil vi får en 6'er:

```
public class BenytTerning
{
    public static void main(String[] arg)
    {
        Terning t;
        t = new Terning(); // opret terning

        // Slå nu med terningen indtil vi får en sekser
        boolean sekser = false;
        int antalKast = 0;

        while (sekser==false)
        {
            t.kast();
            antalKast = antalKast + 1;
            System.out.println("kast "+antalKast+": "+t.værdi);
            if (t.værdi == 6) sekser = true;
        }

        System.out.println("Vi slog en 6'er efter "+antalKast+" slag.");
    }
}
```

---

```
kast 1: 4
kast 2: 2
kast 3: 6
Vi slog en 6'er efter 3 slag.
```

## 4.4.1 Opgaver

1. Skriv et program, der rafler med to terning-objekter, indtil der slås en 6'er.
2. Skriv et program, der rafler med fire terninger, indtil der slås tre eller fire 6'ere. Udskriv antal øjne for hver terning.
3. Skriv et program, der rafler med 12 terninger og hver gang udskriver øjnene, summen af øjnene og hvor mange 6'ere der kom. Brug ArrayList-klassen til at holde styr på terningerne.
4. Lav en Moent-klasse, der repræsenterer en mønt med 2 sider (du kan tage udgangspunkt i Terning.java). Lav metoden krone(), der returnerer true eller false. Lav et program, der kaster en mønt 100 gange og tæller antal gange, det fik krone.

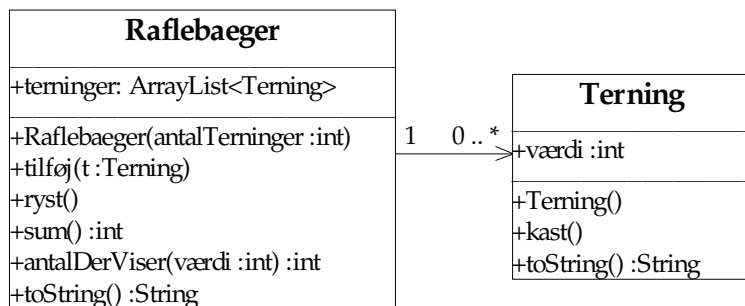
## 4.5 Relationer mellem objekter

Indtil nu har alle vore objekter haft simple typer som variable. Nu vil vi se på objekter, der har andre objekter som variable (dvs. de har referencer til andre objekter).

### 4.5.1 En Raflebæger-klasse

Når man laver et større program, bliver det ofte nødvendigt at uddelegere nogle af opgaverne fra hovedprogrammet til andre dele af programmet. I vores tilfælde kunne vi godt lave et lille terningspil direkte fra main(), men hvis vi skulle lave f.eks. et yatzy- eller matadorspil, ville det blive besværligt, at skulle holde rede på hver enkelt terning (og alle de andre objekter) på den måde. Hver gang en spiller kaster med terningerne, skal man først kaste hver enkelt terning, hvorefter man skal udregne summen (eller i Yatzy undersøge antallet af par, tre ens osv.).

En løsning er at skabe andre, mere overordnede objekter, som tager sig af detaljerne. I vores tilfælde kan man definere en Raflebæger-klasse, som er bekvem at bruge fra hovedprogrammet og som har terningerne og holder styr på dem. Her er UML-klassediagrammet:



*Et raflebæger indeholder terninger.*

*Pilen symboliserer **har-relation**: Et Raflebæger har 0 til \* (flere) terninger.*

Pilen fra raflebæger-klassen til Terning-klassen repræsenterer en **har**-relation. Med 'har' menes, at referencerne til Terning-objekterne kendes af raflebæger-objektet (her via en ArrayList). Terning-objekterne kendes ikke nødvendigvis af hovedprogrammet.

Raflebægeret har metoderne `ryst()`, der kaster alle terningerne, `sum()`, der udregner summen af terningernes værdier, og `antalDerViser()`, der fortæller, hvor mange terninger der har en given værdi (f.eks. hvor mange terninger, der viser 6 øjne).

```
import java.util.*;

public class Raflebaeger
{
    /** listen af terninger, der er i raflebægeret */
    public ArrayList<Terning> terninger;

    public Raflebaeger(int antalTerninger)
    {
        terninger = new ArrayList<Terning>();
        for (int i=0; i<antalTerninger; i++)
        {
            Terning t;
            t = new Terning();
            tilføj(t);
        }
    }

    /** lægger en terning i bægeret */
    public void tilføj(Terning t)
    {
        terninger.add(t);
    }

    /** ryster bægeret, så alle terningerne bliver 'kastet' og får en ny værdi */
    public void ryst()
    {
        for (Terning t : terninger)
        {
            t.kast();
        }
    }

    /** finder summen af alle terningernes værdier */
    public int sum()
    {
        int resultat=0;
        for (Terning t : terninger)
        {
            resultat = resultat + t.værdi;
        }
        return resultat;
    }

    /** finder antallet af terninger, der viser en bestemt værdi */
    public int antalDerViser(int værdi)
    {
        int resultat;
        resultat = 0;
        for (Terning t : terninger)
        {
            if (t.værdi==værdi)
            {
                resultat = resultat + 1;
            }
        }
        return resultat;
    }

    /** beskriver bægerets indhold som en streng */
    public String toString()
    {
        /** (listen toString() kalder toString() på hver terning)
        return terninger.toString();
        */
    }
}
```

Herunder er et lille program, der spiller med tre terninger, indtil man får netop to seksere:

```
public class BenytRaflebaeger
{
    public static void main(String[] arg)
    {
        Raflebaeger bager;
        boolean toSeksere;
        int antalForsøg;

        bager = new Raflebaeger(3);    // opret et bager med 3 terninger
        toSeksere=false;
        antalForsøg = 0;
        while (toSeksere==false)
        {
            bager.ryst();              // kast alle terningerne
            System.out.print("Bæger: " + bager + " sum: " + bager.sum());
            System.out.println(" Antal 6'ere: "+bager.antalDerViser(6)
                               + " antal 5'ere: "+bager.antalDerViser(5));
            if (bager.antalDerViser(6) == 2)
            {
                toSeksere = true;
            }
            antalForsøg++;
        }
        System.out.println("Du fik to seksere efter "+ antalForsøg+" forsøg.");
    }
}

Bæger: [4, 4, 4] sum: 12 Antal 6'ere: 0 antal 5'ere: 0
Bæger: [5, 5, 6] sum: 16 Antal 6'ere: 1 antal 5'ere: 2
Bæger: [2, 5, 6] sum: 13 Antal 6'ere: 1 antal 5'ere: 1
Bæger: [4, 2, 4] sum: 10 Antal 6'ere: 0 antal 5'ere: 0
Bæger: [6, 4, 1] sum: 11 Antal 6'ere: 1 antal 5'ere: 0
Bæger: [6, 6, 4] sum: 16 Antal 6'ere: 2 antal 5'ere: 0
Du fik to seksere efter 6 forsøg.
```

Linjen:

```
bager = new Raflebaeger(3);
```

opretter et raflebaeger med tre terninger i.

## 4.5.2 Opgaver

- 1) Skriv et program, der vha. et Raflebaeger rafler med fire terninger, indtil der slås tre eller fire 6'ere. Udskriv antal øjne for hver terning.
- 2) Skriv et program, der vha. et Raflebaeger rafler med 12 terninger og udskriver terningernes værdier, summen af værdierne og hvor mange 6'ere der kom.
- 3) Skriv et simpelt Yatzy-spil med fem terninger. Man kaster én gang og ser, om man har et par, to par, tre ens, hus (et par og tre ens, f.eks. 25225), fire ens eller fem ens.

Udvid Raflebaeger, så man kan spørge, om der er fire ens, med en fireEns()-metode:

```
public boolean fireEns()
{
    ...
}
```

Lav tilsvarende de andre metoder (toEns(), treEns(), toPar(), hus()...).

(vink: Gør flittigt brug af antalDerViser()-metoden)

Ret toString()-metoden, så den fortæller, om der var fem ens, hus eller lignende.

Lav et program (en klasse med en main()-metode), der rafler et Raflebaeger et par gange og skriver dets indhold ud. Her er et eksempel på, hvordan uddata kunne se ud:

```
1 4 4 3 4 : Tre ens
4 2 1 6 6 : Et par
2 6 2 2 6 : Hus
5 2 3 6 4 : Ingenting
2 3 4 5 4 : Et par
6 5 2 6 2 : To par
6 6 2 2 6 : Hus
```

## 4.6 Nøgleordet this

Nogle gange kan et objekt have brug for at referere til sig selv. Det gøres med nøgleordet `this`, der ligner (og bruges som) en variabel<sup>1</sup>.

---

### `this` refererer til det objekt, man er i

---

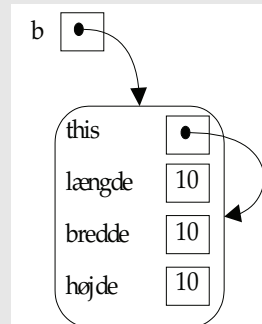
Læs igen definitionen af `Boks2`. I dens `sætMål()`-metode brugte vi andre variabelnavne for parametrene (nemlig `lgd`, `b` og `h`) end objektvariablerne (`længde`, `bredde` og `højde`). Vi kan altid få fat i objektets variabler med `this`, så vi kunne også have brugt de samme variabelnavne:

```
import java.util.*;
public class Boks2medThis
{
    private double længde;
    private double bredde;
    private double højde;

    public void sætMål(double længde, double bredde, double højde)
    {
        if (længde<=0 || bredde<=0 || højde<=0)
        {
            System.out.println("Ugyldige mål. Bruger standardmål.");
            this.længde = 10.0;
            this.bredde = 10.0;
            this.højde = 10.0;
        } else {
            this.længde = længde;
            this.bredde = bredde;
            this.højde = højde;
        }
    }

    public double volumen()
    {
        return bredde*højde*længde;
    }

    public void fjøTilListe(ArrayList l)
    {
        l.add(this);
    }
}
```



*this* virker som en variabel, inde i objektet, der peger på objektet selv (*b* er en variabel, der peger på objektet udefra)

I `sætMål()` er der nu to sæt variabler med samme navn.

Java vælger da altid den variabel, der er "tættest på", dvs. f.eks. 'længde' svarer til parametervariablen `længde`. For at få fat i objektvariablen skal vi bruge `this.længde`.

Derfor skal vi skrive:

```
this.længde = længde;
```

for at tildele objektets længde-variabel den nye værdi.

En anden anvendelse af `this` er, når et objekt har brug for at give en reference til sig selv til et andet objekt. Normalt ville vi tilføje en boks til en liste med:

```
ArrayList l = new ArrayList();
Boks2medThis b = new Boks2medThis();
l.add(b);
```

Med metoden `fjøTilListe()` kan vi i stedet for bede `b` om at tilføje sig selv:

```
b.fjøTilListe(l);
```

Vi vil senere (i `Spiller`-klassen i matadorspillet i kapitlet om nedarvning) se et eksempel på dette, hvor det er en fordel i praksis.

---

1 Man kan dog ikke tildele `this` en anden værdi.

## 4.7 Ekstra eksempler

Dette afsnit giver nogle ekstra eksempler, der repeterer stoffet i kapitlet.

### 4.7.1 En n-sidet terning

Det normale er en 6-sidet terning, men der findes også 4-, 8- 12- og 20-sidede. Klassen nedenfor beskriver en generel n-sidet terning.

```
/** En klasse der beskriver 4-, 8- 12- og 20-sidede terninger */
public class NSidetTerning
{
    /** hvor mange sider har terningen (normalt 6) */
    private int sider;

    /** den side på terningen, der vender opad lige nu */
    private int værdi;

    /** konstruktør der opretter en standardterning med 6 sider */
    public NSidetTerning ()
    {
        sider = 6;
        kast(); // sæt værdi til noget
    }

    /** konstruktør der opretter en terning med et vist antal sider */
    public NSidetTerning (int antalSider)
    {
        if (antalSider >= 3) sider = antalSider;
        else sider = 6;
        kast();
    }

    /** kaster terningen, så den får en anden værdi */
    public void kast ()
    {
        // find en tilfældig side
        double tilfældigtTal = Math.random();
        værdi = (int) (tilfældigtTal * sider + 1);
    }

    /** giver antallet af øjne på den side på terningen, der vender opad */
    public int hentVærdi ()
    {
        return værdi;
    }

    /** ændrer terningen til at vende en bestemt side opad */
    public void sætVærdi (int nyVærdi)
    {
        if (nyVærdi > 0 && nyVærdi <= sider) værdi = nyVærdi;
        else System.out.println("Ugyldig værdi");
    }

    /** giver en beskrivelse af terningen som en streng.
        Hvis den ikke har 6 sider udskrives også antal af sider */
    public String toString ()
    {
        String svar = ""+værdi; // værdi som streng, f.eks. "4"
        if (sider != 6) svar = svar+"("+sider+"s)";
        return svar;
    }
}
```

Vi har ladet antallet af sider og værdien være private og lavet metoden hentVærdi(), som kan bruges udefra. Der er også en sætVærdi()-metode, mens antallet af sider ikke kan ændres udefra, når først terningen er skabt.



NSidetTerning
-sider :int
-værdi :int
+NSidetTerning()
+NSidetTerning(antalSider :int)
+kast()
+hentVærdi() :int
+sætVærdi( :int)
+toString() :String

Her er et program til at afprøve klassen med:

```
public class BenytNSidetTerning
{
    public static void main(String[] arg)
    {
        NSidetTerning t = new NSidetTerning(); // sekssidet terning

        System.out.println("t viser nu "+t.hentVærdi()+" øjne");

        NSidetTerning t6 = new NSidetTerning(6); // sekssidet terning
        NSidetTerning t4 = new NSidetTerning(4); // firesidet terning
        NSidetTerning t12 = new NSidetTerning(12); // tolvsidet terning

        System.out.println("t4 er "+t4); // t4.toString() kaldes implicit
        t4.kast();
        System.out.println("t4 er nu "+t4);
        t4.kast();

        System.out.println("terninger: "+t+" "+t6+" "+t4+" "+t12);
        t.kast();
        t12.kast();
        System.out.println("terninger: "+t+" "+t6+" "+t4+" "+t12);

        for (int i=0; i<5; i++)
        {
            t.kast();
            t6.kast();
            t4.kast();
            t12.kast();
            System.out.println("kast "+i+": "+t+" "+t6+" "+t4+" "+t12);
            if (t.hentVærdi() == t6.hentVærdi())
            {
                System.out.println("t og t6 er ens!");
            }
        }
    }
}
```

```
t viser nu 6 øjne
t4 er 4(4s)
t4 er nu 1(4s)
terninger: 6 1 4(4s) 5(12s)
terninger: 6 1 4(4s) 3(12s)
kast 0: 3 1 4(4s) 2(12s)
kast 1: 1 6 4(4s) 11(12s)
kast 2: 1 1 4(4s) 5(12s)
t og t6 er ens!
kast 3: 3 6 4(4s) 3(12s)
kast 4: 3 2 2(4s) 6(12s)
```

## 4.7.2 Personer

Lad os lave en klasse til at repræsentere en person. Hvert person-objekt skal have et fornavn, et efternavn og en alder. Når man opretter en ny Person, skal man angive disse data, f.eks.: `new Person("Jacob","Nordfalk",30)`, så vi definerer en konstruktør med disse parametre.

Vi definerer også, at hver person har metoden `toString()`, der returnerer en streng med personens oplysninger af formen "Jacob Nordfalk (30 år)".

Desuden har vi metoden `præsentation()`, der skriver oplysningerne pænt ud til skærmen som "Jeg hedder Jacob og jeg er 30 år". Denne metode returnerer ikke noget, men skriver i stedet hilsenen ud til skærmen (personer under 5 år siger bare "agyyy!")

Til sidst kunne man forestille sig, at en person kan hilse på en anden person (metoden `hils()`). Det afhænger af alderen, hvordan man hilser. En person på over 60 år vil hilse på Jacob med "Goddag, hr. Nordfalk", mens en yngre bare vil sige "Hej Jacob".

```
import java.util.*;
public class Person
{
    public String fornavn;
    public String efternavn;
    public int alder;
    public ArrayList<Konto> konti; // bruges senere

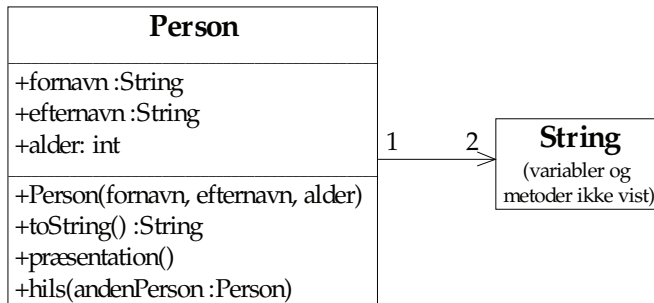
    public Person(String fornavnP, String efternavnP, int alderP)
    {
        fornavn = fornavnP;
        efternavn = efternavnP;
        alder = alderP;
        konti = new ArrayList<Konto>(); // bruges senere
    }

    public String toString()
    {
        return fornavn+" "+efternavn+" (" +alder+" år)";
    }

    public void præsentation()
    {
        if (alder < 5) System.out.println("agyyy!");
        else System.out.println("Jeg hedder "+fornavn+" og jeg er "+alder+" år.");
    }

    public void hils(Person andenPerson)
    {
        if (alder < 5) System.out.print("ma ma.. ");
        else if (alder < 60) System.out.print("Hej "+andenPerson.fornavn+" ");
        else System.out.print("Goddag, hr. "+andenPerson.etternavn+" ");
        præsentation();
    }
}
```

Bemærk, at Person-objektet har to andre objekter, nemlig to strenge. Selvom man ikke plejer at tegne strenge med i klassediagrammer, har vi alligevel taget dem med for at illustrere, at der faktisk også eksisterer en **har**-relation mellem disse to klasser.



*En Person har to String-objekter. Disse er undtagelsesvist også vist.*

Læg også mærke til, hvordan vi fra hils()-metoden kalder præsentation(). Lad os prøve at oprette tre personer og lade dem præsentere sig og derpå hilse på hinanden:

```

public class BenytPerson
{
    public static void main(String[] arg)
    {
        Person j, k, l;
        j = new Person("Jacob", "Nordfalk", 30);
        k = new Person("Kai", "Lund", 86);
        l = new Person("Lars", "Holm", 2);

        System.out.println("Vi har oprettet "+j+", "+k+" og "+l);
        j.præsentation();
        k.præsentation();
        l.præsentation();
        j.hils(k);
        k.hils(j);
        l.hils(j);
    }
}
  
```

```

Vi har oprettet Jacob Nordfalk (30 år), Kai Lund (86 år) og Lars Holm (2 år)
Jeg hedder Jacob og jeg er 30 år.
Jeg hedder Kai og jeg er 86 år.
agyyy!
Hej Kai. Jeg hedder Jacob og jeg er 30 år.
Goddag, hr. Nordfalk. Jeg hedder Kai og jeg er 86 år.
ma ma.. agyyy!
  
```

I linjen

```
x.hils(y);
```

er det x-variablens person (Jacob), der hilser på y-variablens person. Da x-variablens person er under 60, vil den uformelle hilsen "Hej Kai" blive brugt. I linjen under er det lige omvendt.

## 4.7.3 Design af klasser

I kapitel 22, Objektorienteret analyse og design kan du læse mere om, hvordan man designer sit program, d.v.s. vælger hvilke klasser, der skal defineres, hvilke metoder de skal have og hvilke relationer, der skal være mellem objekterne.

## 4.8 Test dig selv

- 1) Hvad er en klasse?
- 2) Hvad kan en klasse indeholde?
- 3) Lav en klassedefinition og beskriv syntaksen.
- 4) Hvad er en metode?
- 5) Definér en metode og beskriv syntaksen.
- 6) Hvad betyder det, at en metode returnerer noget?
- 7) Beskriv reglerne omkring return og returværdi.
- 8) Lav en metode, der finder kvadratet af et tal ( $x \cdot x$ ) og returnerer det.
- 9) Hvad er et objekt ?
- 10) Hvad er en objektvariabel?
- 11) Hvordan oprettes et objekt.
- 12) Hvad sker der, når et objekt oprettes?
- 13) Hvad er en konstruktør? Hvilken kode bør være i konstruktøren?
- 14) Hvad er standardkonstruktøren?
- 15) Hvad er `this`?

## 4.9 Resumé

- Når man definerer en ny klasse, definerer man en ny type objekter, så man kan sige, at en klasse er det samme som en objekttype.
- En klasse indeholder variabler, der indeholder data, og metoder, der beskriver opførsel, dvs. hvad der skal ske, hvis en af objektets metoder kaldes.
- Klasser bør have stort startbogstav. Metoder og variabler bør have lille startbogstav.
- En metode indeholder den programkode, der skal køres, hvis metoden kaldes. Dette kaldes metodens krop.
- En metode skal erklære, hvilken type data den returnerer, f.eks. `int`, `double` eller `String`. Det skrives lige før metodens navn. Hvis metoden ikke returnerer noget, skal der stå `'void'` (som f.eks. i `sætMål()` i `Boks`-klassen afnit 4.2). Hvis metoden returnerer noget, skal der i metodekroppen stå `'return xx'` et eller flere steder, hvor `xx` er en variabel eller et udtryk af samme type som den erklærede returtype (f.eks. i `volumen()` i `Boks`).
- Et objekt er en datastruktur med nogle metoder tilknyttet. Objektet oprettes med `new Klassenavn()`.
- Når et objekt oprettes, reserveres der plads til dets variabler og en konstruktør kaldes.
- En konstruktør er en speciel metode, der hedder det samme som klassen. Der kaldes altid netop én konstruktør, når et objekt oprettes. I konstruktøren kan man sørge for, at objektets variabler får nogle fornuftige startværdier.
- Standardkonstruktøren er en konstruktør, som Java definerer, hvis en klasse ikke har nogen konstruktører defineret overhovedet.
- `this` er et nøgleord (reserveret ord), der er en reference til objektet selv.

## 4.9.1 Formen af en klasse

Den normale form af en klasse er skitseret nedenfor.

Øverst erklæres, hvilke andre klasser der skal være kendt og navnet på klassen. Derefter kommer der en blok med selve definitionen af klassen.

Først i blokken kommer variabler, der knytter sig til objektet. Disse variabler kaldes objektvariabler og har en værdi, så længe objektet findes.

Foran variablen kan man angive, i hvor høj grad, der er adgang til variablen udefra (synligheden). Der er fire grader af adgang: `private`, `ingenting`, `protected` og `public` (disse er forklaret i afsnit 6.9).

Derefter erklæres konstruktører og til sidst de almindelige metoder.

```
// Klassenavn skal være i filen Klassenavn.java

import klasser; // f.eks. import java.util.*;
...

public class Klassenavn
{
    // mellem { og } skal definitionen af klassen stå

    // erklæring af variabler (og evt. samtidig initialisering)
    adgang type navnPåObjektvariabel;
    public int n;
    private String s;
    private String s2 = "goddag"; // samtidig initialisering

    // erklæring af konstruktører, evt. med parametre
    adgang Klassenavn(type1 parameter1, type2 parameter2, ...)
    {
        // kode der sætter objektvariablerne til deres startværdier
        ...
    }

    // eksempler på konstruktører:
    public Klassenavn()
    {
        n = 5;
        s = "hej";
    }

    public Klassenavn(int nn, String ss)
    {
        n = nn;
        s = ss;
    }

    // erklæring af metoder, evt. med parametre
    adgang returtype metodenavn(type1 parameter1, type2 parameter2, ...)
    {
        ...
    }

    // eksempler på metoder:
    public int metode1()
    {
        ...
        return 15; // noget af type int
    }

    public void metode2(int nn, String ss)
    {
        ...
    }
} // slut på definitionen af klassen
```

## 4.9.2 Formen af en metode

Metoder består af et hoved:

```
public int metode1()
```

og en krop:

```
{  
    type1 lokalVariabel1;  
    type2 lokalVariabel2;  
    ...  
    // programkode her  
}
```

### Metodehovedet

Metodehovedet har formen:

```
adgang returtype metodenavn(type1 parameter1, type2 parameter2, ...)
```

- **adgang** – ligesom med variabler kan der stå private, ingenting, protected og public.
- **returtype** – afgør, hvad metoden returnerer. Kan være void (ingen returtype) eller int, double eller en anden simpel type, String eller en anden objekttype. Hvis den er void, betyder det, at metoden ikke returnerer noget.
- **metodenavn** – skal følge de samme regler som variabelnavne.
- **parametre** – en liste af de variabeltyper, der skal overføres, når metoden kaldes (listen kan godt være tom). Disse kaldes parametervariabler og er lokale variabler. Typen kan være int, double eller en anden simpel type, String eller en anden objekttype. Variabelnavnet bestemmer navnet, der skal angives i metodekroppen for at få fat i den værdi, der blev kaldt med.

Eksempler:

```
double volumen()  
  
public void sætMål(double lgd, double b, double h)  
  
public void tilføj(Terning t)  
  
public void kast()  
  
public int antalDerViser(int værdi)  
  
public void præsentation()  
  
public Konto(Person ejer)  
  
public void overførsel(int kroner)  
  
public String toString()
```

### Metodekroppen

Kroppen af en metode starter med { og slutter med } og skal komme lige efter hovedet. I metodens krop kan man definere lokale variabler og programkode. Lokale variabler har kun en værdi i det korte tidsrum, koden i metodens krop er ved at blive udført.

Hvis der i hovedet blev angivet en anden returtype end void, skal der stå en return-sætning nederst i kroppen med noget af samme type som returtypen.

```
public int metode3()  
{  
    ...  
    return 42;  
}
```

## 4.10 Opgaver

Husk at lave små main()-programmer, der afprøver de ting, du programmerer.

- 1) Lav en klasse, der repræsenterer en bil. En bil har en farve, et antal kørte kilometer og en nypris. Definér metoderne:

```
public void kør(int antalKilometer) // opdaterer antal kørte kilometre
public double pris()                // giver den vurderede salgspris
public String toString()            // giver en beskrivelse af bilen
```

- 2) Udbyg Bil-klassen med en liste, der husker, hvilke personer, der sidder i bilen lige nu. Definér følgende metoder på Bil-klassen og afprøv klassen.

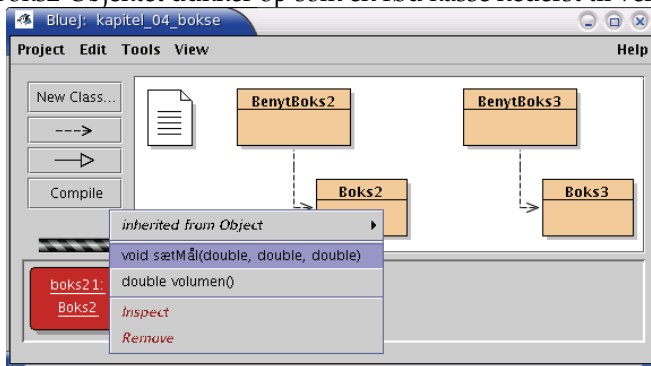
```
public void enSætterSigInd(Person p) // kaldes når person sætter sig ind i bilen
public String hvemSidderIBilen()     // giver en beskrivelse af personerne i bilen
public void alleStigerUd()           // kaldes, når alle stiger ud af bilen
```

- 3) Lav et program, der holder styr på en musiksamling. Opret en klasse, der repræsenterer en udgivelse (int år, String navn, String gruppe, String pladeselskab). Programmet skal huske listen over udgivelser og kunne udskrive den.

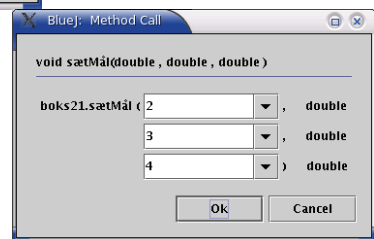
### 4.10.1 BlueJ

Udviklingsværktøjet BlueJ, der er beskrevet i afsnit 1.3.5, er meget velegnet til at forstå begreberne behandlet i dette kapitel bedre. Har du installeret BlueJ, kan du hente bogens eksempler som BlueJ-projekter (på <http://javabog.dk/OOP/kode/bluej-projekter/>).

- Pak ZIP-filen med BlueJ-projekterne ud, start BlueJ og åbn projektet kapitel\_04\_bokse. Projektet viser klasserne Boks2, BenytBoks2, Boks3 og BenytBoks3. Tryk på 'Compile'-knappen for at oversætte alle klasserne.
- Højreklik derefter på Boks2 og opret et objekt ved at vælge 'new Boks2()'. Boks2-Objektet dukker op som en rød kasse nederst til venstre:



- Højreklik på det røde Boks2-objekt og kald sætMål(). Sæt parametrene til f.eks. 2, 3 og 3 og klik OK. Undersøg objektets variabler (højreklik og vælg *Inspect*). Kald derefter volumen() igen og tjek resultatet.
- Højreklik på BenytBoks2 og vælg main() for at køre programmet.
- Åbn nu Boks3 og se på kildekoden. Skift derefter til 'Interface' for at se javadoc for klassen.



Når du har leget med Boks-klasserne som de er, så prøv at ændre i klasserne (ret i koden). Prøv derefter de andre BlueJ-projekter.

## 4.10.2 Fejlfinding

- 1) Der er 2 fejl i koden nedenfor. Find dem og ret dem. Kig eventuelt på afsnittene om formen af en klasse og formen af en metode ovenfor.

```
public class Fejlfinding1
{
    private int a = 5;
    private String b;
    private c String;
    {
    }
```

- 2) Der er 3 fejl i koden nedenfor. Find dem og ret dem.

```
public class Fejlfinding2
{
    private String b;

    public Fejlfinding2() {
        return b;
    }

    b = "Hej";

    public Fejlfinding2(String c) {
        b = c
    }
}
```

- 3) Der er 9 fejl i koden nedenfor. Find dem, ret dem og begrund rettelserne.

```
public class Fejlfinding4
{
    private int a = 5;
    private String b;

    public void x1(int y)
    {
        y = a;
    }

    a = 2;

    public Fejlfind(int a) {
        a = 4;
        String = "goddag";
    }

    public x2(int y)
    {
        a = y*2;
    }

    public int x3(y int)
    {
        b = y;
    }

    public void x4(int y)
    {
        return 5;
    }
}
```



4) Der er 8 fejl i koden nedenfor. Find dem og ret dem.

```
import java.util.*;

public class Fejlfinding3
{
    public String l = 'hej';
    public string etLangtVariabelnavn;
    public arrayList v2;
    public INT v3;
    public v5;
    public int vi = 5.3;
    public ArrayList vi3 = "xx";
    public String vi5 = new String(Hej);
}
```

5) Der er 6 fejl i koden nedenfor. Find dem og ret dem.

```
public class Fejlfinding5
{
    private int a = 5
    public void x1(int y)    {
        a = y;
    }
    public void x2(int y) // fejlmeddelelse: 'class' or 'interface' expected
    {
        a = y*2*x1;
    }
    public int x3(int y)
    {
        a = y*x2();
    }
    }

    public void x4(int y)
    {
        x4 = 8;
    }
}
```

6) Find så mange fejl du kan i koden nedenfor og ret dem.

```
public class Fejlfinding6
{
    public int m()
    {
        System.out.println("Metode m blev kaldt.");
    }

    public void m2()
    {
        String s = "Metode m2 blev kaldt."
        System.out.println(s);
        return s;
    }

    public m3()
    {
        System.out.println("Metode m3 blev kaldt.");
    }

    public void m4(int)
    {
        System.out.println("m4 fik parameter "+p);
    }

    public void m5(p1, p2, p3)
    {
        System.out.println("m5 fik "+p1+" og "+p2+" og "+p3);
        System.out.println("s er: "+s);
        String s = p2.toUpperCase();
    }
}
```

## 4.11 Avanceret

Er man uvant med objektorienteret programmering, kan det være svært at forstå relationer mellem objekter og "rolleskiftet", der sker, når man skifter fra en klasse til en anden.

### 4.11.1 Bankkonti

Lad os se nærmere på relationer mellem objekter i bankverdenen. De vigtigste egenskaber ved en bankkonto er saldoen og hvem der ejer den. Et Konto-objekt kunne altså have et Person-objekt tilknyttet, en **har**-relation, og denne person bør være kendt, når kontoen oprettes. Det er derfor oplagt, at ejeren skal angives i Konto's konstruktør. Kontoen skal også sørge for at indsætte sig selv (her bruges nøgleordet `this`) i personens liste over konti, sådan at der er konsistens mellem data i Konto-objektet og data i Person-objektet.

```
public class Konto
{
    public int saldo;
    public Person ejer;
    public Konto(Person ejeren)
    {
        ejer = ejeren;           // Sæt kontoen til at referere til personen
        ejer.konti.add(this);    // ... og personen til at referere til kontoen
        saldo = 0;
    }

    public void overførsel(int kroner)
    {
        saldo = saldo + kroner;
    }

    public String toString()
    {
        return ejer+" har "+saldo+" kr.";
    }
}
```

```
classDiagram
    class Konto {
        +saldo:int
        +ejer:Person
        +Konto(ejer:Person)
        +overførsel(kroner:int)
        +toString():String
    }
    class Person {
        +konti:List<Konto>
    }
    Konto "0..*" -- "1" Person
```

*En Konto har altid en Person tilknyttet*

Dette er et eksempel på en **har**-relation begge veje. Læg mærke til, hvordan vi fra Konto-objektet får fat i listen af ejerens konti. Den tilføjer vi så kontoen selv – `this` – til:

```
ejer.konti.add(this); // i Konto-klassen
```

Den samme handling, udført fra f.eks. `main()` i `BenytPerson`, ville se helt anderledes ud:

```
Person p = new Person();
Konto k = new Konto(); // vi "leger" at der er en tom konstruktør
p.konti.add(k);        // i en anden klasse end Konto eller Person
```

Og var koden lagt i `Person`-klassen ville den se ud som

```
Konto k = new Konto(); // vi "leger" at der er en tom konstruktør
konti.add(k);          // i Person-klassen
```

### 4.11.2 Opgaver

- 1) Udbyg `Person`-klassen med metoden `formue()`, der skal returnere summen af saldi på personens konti. Udbyg `BenytPerson` sådan, at hver person har en eller flere konti (husk at en `Konto` oprettes med en `Person` i konstruktøren).
- 2) Lav en klasse, der repræsenterer en `Postering` på en bankkonto med tekst, indsat beløb (udtræk regnes negativt) og dato. Udvid `Konto` med en liste af `posteringer` (`ArrayList` af `Postering`-objekter) og metoden `udskrivPosteringer()`, der skal udskrive `posteringerne` og løbende saldo på skærmen.
- 3) Udbyg klassen `Person`, så en person kan eje en bil. Udbyg metoden `formue()`, sådan at den husker at indregne bilens pris. Metoden skal virke både for personer med og uden bil (`Person`-objekter uden bil kan have denne variabel sat til `null`).

# 5 Nedarvning

Indhold:

- At udvide (arve fra) klasser
- Konstruktører og arv
- Polymorfi
- Object-klassen
- Større eksempel: Et Matador-spil

Kapitlet forudsættes i resten af bogen.

Forudsætter kapitel 4, Definition af klasser.

I dette kapitel vil vi se på, hvordan man kan genbruge programkode, ved at tage en eksisterende klasse og udbygge den med flere metoder og variabler (nedarvning).

## 5.1 At udbygge eksisterende klasser

Hvad gør man, hvis man ønsker en klasse, der ligner en eksisterende klasse, men alligevel ikke helt er den samme?

Svaret er: Man kan definere underklasser, der **arver** (genbruger en del af koden) fra en anden klasse og kun definerer den ekstra kode, der skal til for at definere underklassen i forhold til stamklassen (kaldet superklassen).

Arv (eng.: inheritance) er et meget vigtigt element i objektorienterede sprog. Med nedarvning kan man have en hel samling af klasser, der ligner hinanden på visse punkter, men som er forskellige på andre punkter.

### 5.1.1 Eksempel: En falsk terning

Hvis man vil snyde i terningspil, findes der et kendt trick: Bruge sine egne terninger, hvor man har boret 1'er-sidens hul ud, kommet bly i hullet og malet det pænt over, så det ikke kan ses. Sådant en terning vil have meget lille sandsynlighed for at få en 1'er og en ret stor sandsynlighed for at få en 6'er.

Herunder har vi lavet en nedarvning fra Terning (se afsnit 4.4) til en ny klasse, FalskTerning, ved at starte erklæringen med:

```
public class FalskTerning extends Terning
```

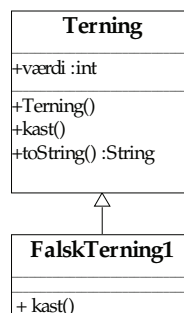
Vi har dermed automatisk overtaget (arvet) alle metoder og variabler fra Terning-klassen. Dvs. at et FalskTerning1-objekt også har en værdi-variabel og en toString()-metode.

Vi ændrer nu klassens opførsel, ved at definere en anden udgave af kast()-metoden:

```
/** En Terning-klasse for falske terninger. */
public class FalskTerning1 extends Terning
{
    /** tilsidesæt kast med en "bedre" udgave */
    public void kast()
    {
        // udskriv så vi kan se at metoden bliver kaldt
        // System.out.println("[kast() på FalskTerning1] ");

        værdi = (int) (6*Math.random() + 1);

        // er det 1 eller 2? Så lav det om til 6!
        if ( værdi <= 2 ) værdi = 6;
    }
}
```



I klassesdiagrammet til højre er nedarvningen vist med en hul pil fra FalskTerning1 til Terning.

Dette kaldes også en **er-en**-relation; FalskTerning1 **er en** Terning, da den jo har alle de egenskaber (variabler og metoder), en terning har.

*FalskTerning1 overtager egenskaber (arver) fra Terning, men definerer sin egen udgave af kast().*

Kort sagt:

---

**En klasse kan arve variabler og metoder fra en anden klasse**

**Klassen, der nedarves fra, kaldes superklassen**

**Klassen, der arver fra superklassen, kaldes underklassen**

**Underklassen kan tilsidesætte (omdefinere) metoder arvet fra superklassen ved at definere dem igen**

---

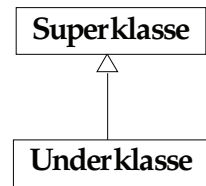
Andre steder i litteraturen er der brugt talrige betegnelser for superklasse, underklasse og tilsidesættelse. Her er et udpluk:

**Superklasse** kaldes også: Basisklasse, forældreklasse, stamklasse.

**Underklasse** kaldes også: Afledt klasse, nedarvet klasse, subklasse.

**Tilsidesætte** (eng.: override) kaldes også: omdefinere, overskrive.

I vores eksempel er superklassen Terning. Underklassen FalskTerning1 har tilsidesat metoden kast().



I det følgende program kastes med to terninger, en rigtig og en falsk:

```
public class Snydespill
{
    public static void main(String[] arg)
    {
        Terning t1 = new Terning();
        FalskTerning1 t2 = new FalskTerning1();

        System.out.println("t1: "+t1); // kunne også kalde t1.toString()
        System.out.println("t2: "+t2);

        for (int i=0; i<5; i++)
        {
            t1.kast();
            t2.kast();
            System.out.println("t1=" + t1 + " t2=" + t2);
            if (t1.værdi == t2.værdi) System.out.println("To ens!");
        }
    }
}

t1: 1
t2: 3
t1=1 t2=5
t1=1 t2=6
t1=4 t2=3
t1=6 t2=6
To ens!
t1=2 t2=6
```

Vi kan altså bruge FalskTerning1-objekter på præcis samme måde som Terning-objekter. Bemærk, hvordan t2 giver 6 meget oftere end t1.

## 5.1.2 At udbygge med flere metoder og variable

Lad os nu se på et eksempel, hvor vi definerer nogle variable og metoder i nedarvingen.

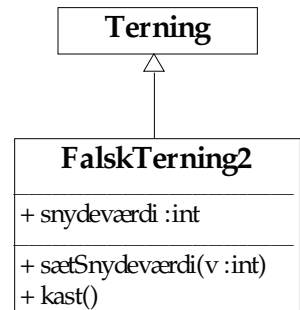
```
public class FalskTerning2 extends Terning
{
    public int snydeværdi;

    public void sætSnydeværdi(int nySnydeværdi)
    {
        snydeværdi = nySnydeværdi;
    }

    public void kast()
    {
        //System.out.println("[kast() på FalskTerning2] ");

        værdi = (int) (6*Math.random() + 1);

        // 1 eller 2? Så lav det om til snydeværdi!
        if ( værdi <= 2 ) værdi = snydeværdi;
    }
}
```



FalskTerning2 har fået en ekstra variabel, snydeværdi, og en ekstra metode, sætSnydeværdi()<sup>1</sup>, der sætter snydeværdi til noget andet.

```
public class Snydespil2
{
    public static void main(String[] arg)
    {
        FalskTerning2 t1 = new FalskTerning2();
        t1.sætSnydeværdi(4);

        for (int i=0; i<3; i++)
        {
            t1.kast();
            System.out.println("t1=" + t1);
        }
    }
}
```

---

```
t1=4
t1=4
t1=6
```

## 5.1.3 Nøgleordet super

Nogen gange ønsker man i en nedarvet klasse, at få adgang til superklassens metoder, selv om de måske er blevet tilsidesat med en ny definition i nedarvingen. F.eks. kunne det være rart, hvis vi kunne genbruge den oprindelige kast()-metode i FalskTerning.

Med super refererer man til de metoder, der er kendt for superklassen. Dermed kan vi skrive en smartere udgave af FalskTerning:

```
public class FalskTerning3 extends Terning
{
    public void kast ()
    {
        super.kast(); // kald den oprindelige kast-metode

        // blev det 1 eller 2? Så lav det om til en 6'er!
        if ( værdi <= 2 ) værdi = 6;
    }
}
```

super.kast() kalder kast()-metoden i superklassen. Derefter tager vi højde for, at det er en falsk terning.

---

<sup>1</sup> Egentlig er sætSnydeværdi() overflødig, da snydeværdi er public, men vi skal bruge den til at illustrere en pointe i næste afsnit.

## 5.1.4 Opgaver

- 1) Lav en LudoTerning, der arver fra Terning. Tilsidesæt toString() med en, der giver "\*" på en 3er og "globus" på en 4er (vink: kopiér Terning's toString()-metode over i LudoTerning og ret i den). Afprøv klassen.
- 2) Byg videre på opgave 4.10 og opret klassen Transportmiddel. Et transportmiddel har en farve, et navn, et antal tilbagelagte kilometer og en nypris. Definér metoderne  

```
public void bevæg(int antalKilometer) // opdaterer antal kilometre
public double pris() // giver den vurderede salgspris
public String toString() // giver en beskrivelse af transportmidlet
```

Opret nedarvingerne Cykel, Skib og Bil med hver sin pris()-metode.
- 3) Forestil dig en virksomhed, hvor der er forskellige slags personer: Ansatte og kunder. De ansatte er delt op i medarbejdere og ledere. Skitsér et passende klassesdiagram.
- 4) Lav klasserne til et skak-spil: Definér superklassen Brik med egenskaben farve (sort eller hvid), position *x* og position *y* (hver mellem 1 og 8). Definér også metoden  

```
public boolean kanFlytteTil(int xNy, int yNy) // om brikken kan flytte dertil
```

der (for Brik) returnerer sand, hvis positionen eksisterer (*xNy* og *yNy* er mellem 1 og 8). Definér nedarvingerne Bonde og Taarn med tilsidesat kanFlytteTil().
- 5) Lav et system til at arbejde med forskellige geometriske figurer.  
Opret klassen Figur med metoderne beregnAreal() og beregnOmkreds().  
Lav nedarvingerne Punkt, Linje (med variabelen længde), Cirkel (med variabel radius), Rektangel (med variablerne højde og bredde).

## 5.2 Polymorfe variabler

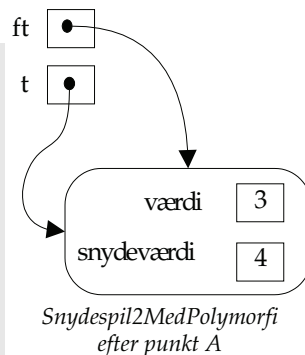
Se på følgende eksempel:

```
public class Snyderpil2medPolymorfi
{
    public static void main(String[] arg)
    {
        FalskTerning2 ft = new FalskTerning2();
        ft.sætSnydeværdi(4);

        Terning t;
        t = ft;                                     // punkt A

        for (int i=0; i<3; i++)
        {
            t.kast();
            System.out.println("t=" + t);
        }
    }
}
```

t=4  
t=6  
t=6



Hov: Terning-variablen `t` refererer nu pludselig til et `FalskTerning2`-objekt ?!

```
t = ft;
```

Der er altså ikke overensstemmelse mellem typen på venstre side (`Terning`) og typen på højre side (`FalskTerning2`). Hvad så med typesikkerheden?

### 5.2.1 Dispensation fra traditionel typesikkerhed

Typesikkerhed gør, at man ikke f.eks. kan tildele et `Point`-objekt til en `Terning`-variabel uden at få en sprogfejl under oversættelsen.

Hvis man kunne det, ville programmerne kunne indeholde mange fejl, der var svære at finde. Hvis man f.eks. et eller andet sted i et kæmpeprogram havde sat en `Terning`-variabel til at referere til et `Point`-objekt, og det var tilladt, hvad skulle der så ske, når man så (måske langt senere i en anden del af programmet) forsøgte at kalde dette objekts `kast()`-metode? Et `Point`-objekt har jo ingen `kast()`-metode. Det kunne blive meget svært at finde ud af, hvordan den forkerte tildeling fandt sted. Sagt med andre ord: Normalt skal vi være lykkelige for, at Java har denne regel om typesikkerhed.

Der er imidlertid en meget fornuftig dispensation fra denne regel:

---

**En variabel kan referere til objekter af en underklasse af variabelens type**

---

`t`-variablen har ikke ændret type (det kan variabler ikke), men den peger nu på et objekt af typen `FalskTerning2`. Dette objekt har jo alle metoder og data, som et `Terning`-objekt har, så vi kan ikke få kaldt ikke-eksisterende metoder, hvis vi bare "lader som om", den peger på et `Terning`-objekt. At `FalskTerning2`-objektet også har en objektvariabel, `snydeværdi`, og en ekstra metode, kan vi være ligeglade med, de kan bare ikke ses fra variabelen.

Dispensationen giver altså mening, fordi en nedarving (f.eks. et `FalskTerning2`-objekt) set udefra kan lade, som om det også er af superklassens type (et `Terning`-objekt). Udefra har det jo *mindst* de samme objektvariabler og metoder, da det har arvet dem.

Selvom variabelen `t` af typen `Terning` refererer til et `FalskTerning2`-objekt, kan man *kun* bruge den til at anvende metoder/variabler i objektet, som stammer fra `Terning`-klassen:

```
t.snydeværdi = 4; // sprogfejl: snydeværdi er ikke defineret i Terning
t.sætSnydeværdi(4); // sprogfejl: sætSnydeværdi() er ikke defineret i Terning
```



## 5.2.2 Polymorfi

En anden meget væsentlig detalje omkring denne dispensation er, at det er *objektets* type, ikke variabelens, der bestemmer, hvilken metodekrop der bliver udført, når vi kalder en metode:

```
t.kast(); // kalder FalskTerning2's kast,  
        // fordi t peger på et FalskTerning2-objekt.
```

Herover kalder vi altså den `kast()`-metode, der findes i `FalskTerning2`-klassen. Den kigger således ikke på variabelen `t`'s type (så ville den jo bare udføre `kast()`-metoden i `Terning`).

---

**Variablens type bestemmer, hvilke metoder der kan kaldes**

---

**Objektets type bestemmer, hvilken metodekrop der bliver udført**

---

Af samme grund kaldes det at definere en metode, som allerede findes, fordi den er arvet, for *tilsidesættelse* af metoden. Man *tilsidesætter* metodens opførsel med en anden opførsel (en anden metodekrop).

## 5.2.3 Eksempel på polymorfi: Brug af Raflebaeger

Da `FalskTerning2`-objekter også er af type `Terning`, kan de bruges i et `Raflebaeger`:

```
public class SnydeMedBaeger  
{  
    public static void main(String[] arg)  
    {  
        Raflebaeger bager = new Raflebaeger(0); // opret et bager med nul terninger  
  
        Terning t = new Terning();  
        bager.tilføj(t); // føj en almindelig terning til bageret  
  
        FalskTerning2 ft = new FalskTerning2();  
        ft.sætSnydeværdi(6);  
        bager.tilføj(ft); // tilføj() får et objekt af typen Terning,  
                        // og dermed også af typen FalskTerning2.  
  
        ft = new FalskTerning2();  
        ft.snydeværdi=6;  
        t=ft; // t bruges som mellemvariabel for sjov.  
        bager.tilføj(t);  
  
        for (int i=1; i<10; i++)  
        {  
            bager.ryst();  
        }  
    }  
}
```

I `SnydeMedBaeger` kaldes `Raflebaeger`'s `ryst()`-metode. Hvis du nu kigger i definitionen af dennes `ryst()`-metode (se afsnit 4.5.1), kan du se, at den kalder `kast()`-metoden på de enkelte objekter i "terninger"-listen:

```
public void ryst()  
{  
    for (Terning t : terninger)  
    {  
        t.kast();  
    }  
}
```

Da to af objekterne, vi har lagt ind i bageret, er af typen `FalskTerning2`, vil `Raflebaeger`'s `ryst()`-metode, når den kommer til et objekt af denne type, kalde `FalskTerning2`'s `kast()` helt automatisk. Resultatet er altså, at vi får større sandsynlighed for at få seksere.

Faktisk har vi ændret den måde, et Raflebaeger-objekt opfører sig på, helt uden at ændre i Raflebaeger-klassen! Raflebaeger ved ikke noget om FalskTerning2, men kan alligevel bruge den.

En programmør kan altså lave en Raflebaeger-klasse, som kan alt muligt smart: Kaste terninger, se hvor mange ens der er, tælle summen af øjnene, se om der er en stigende følge (eng.: straight) osv.

Når en anden programmør vil lave en ny slags terning (f.eks. en snydeterning), behøver han ikke sætte sig ind i, hvordan Raflebaeger-klassen virker og lave tilpasninger af den, for den kan automatisk bruge hans egen nye slags terning!

## 5.2.4 Hvilken vej er en variabel polymorf ?

Når følgende er muligt:

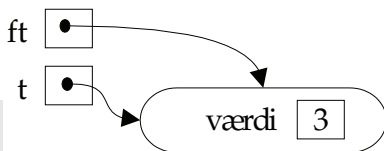
```
Terning t;  
FalskTerning2 ft;  
  
ft = new FalskTerning2();  
t = ft;
```

Hvad så med det omvendte? Kan man tildele en FalskTerning2-variabel en reference til et objekt af typen Terning?

Svaret er: Nej!

Det er jo typen af `ft` (FalskTerning2), der bestemmer, hvilke metoder og variabler vi kan bruge med `ft`. Dvs. vi ville kunne skrive:

```
Terning t;  
FalskTerning2 ft;  
  
t = new Terning();  
ft = t; // sprogfejl  
ft.snydeværdi = 2;
```



Efter punkt A  
(programmet vil ikke oversætte)

Hvis den sidste sætning kunne udføres, ville det være uheldigt: Terning-objektet som `ft` refererer til, har jo ingen `snydeværdi`!

Det er altså et brud på typesikkerhedsreglen og Java tillader det derfor ikke. Man må ikke kunne kalde noget, der ikke findes på objektet.

Bemærk, at her, som i andre sammenhænge, kigger Java kun på en linje ad gangen. F.eks. giver nedenstående stadig en sprogfejl, selvom det i princippet kunne lade sig gøre:

```
Terning t;  
FalskTerning2 ft;  
  
t = new FalskTerning2();  
ft = t; // sprogfejl  
ft.snydeværdi = 2;
```

Her refererer `ft` i sidste linje til et rigtigt FalskTerning2-objekt, og den sidste linje ville derfor give mening, men programmet kan ikke oversættes, fordi typesikkerhedsreglen med dispensation ikke er opfyldt.

## Et andet eksempel

Forestiller vi os den generelle klasse `Dyr`, med nedarvinger `Hest` og `Hund`, kan man skrive

```
Dyr d = new Hest();
```

da en `Hest` er-et `Dyr`. Men vi kan ikke skrive

```
Hest h = new Dyr();
```

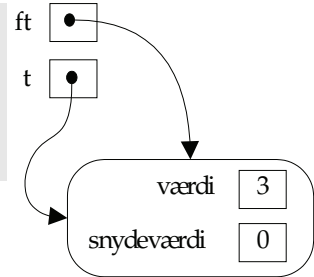
da `Dyr` er en generel klasse, der kunne være et hvilket som helst slags dyr (herunder f.eks. også en hund). Den må kan vi ikke lægge den ind i en `Hest`-variabel.

## 5.2.5 Reference-typekonvertering

Dispensationen i typesikkerhedsreglen svarer til den implicitte værditypekonvertering: Ved konvertering fra int til double behøver programmøren ikke angive eksplicit, at denne værdi skal *forsøges* konverteret. Når en typekonvertering med garanti giver det ønskede, laver Java den implicit.

I foregående eksempel så vi noget, der burde gå godt, men hvor Javas typeregul forhindrer oversættelse. Her er vi nødt til at bruge eksplicit reference-typekonvertering:

```
Terning t;  
FalskTerning2 ft;  
  
t = new FalskTerning2();  
ft = (FalskTerning2) t; // OK, men muligvis  
                        // køretidsfejl  
  
                        // punkt A  
ft.snydeværdi = 2;
```



Det ligner en almindelig eksplicit værditypekonvertering (eng.: cast) og Javas betegnelse for det er også det samme.

Hvis reference-typekonverteringen går galt (det opdages først under programudførelsen), kommer der en køretidsfejl (undtagelsen `ClassCastException` opstår) og programmet stopper.

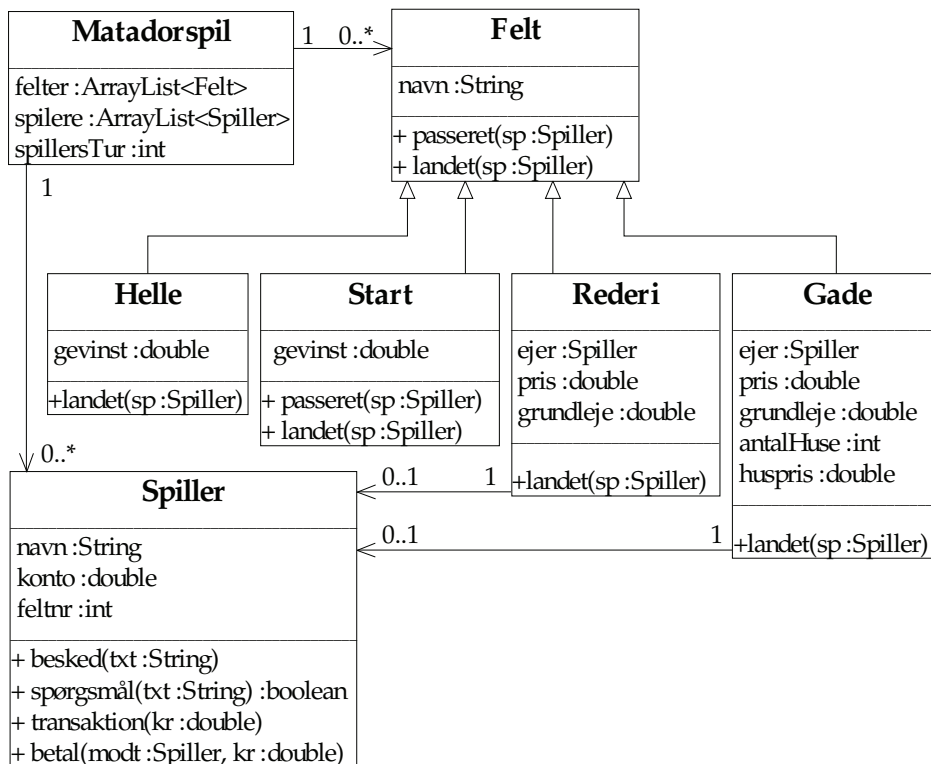
Der er dog nogle tilfælde, hvor Java, selv når man har lavet en reference-typekonvertering, kan opdage en uheldig konvertering. Hvis de to klasser, der forsøges at konverteres imellem, ikke arver fra hinanden, får man en sprogfejl på oversættertidspunktet:

```
Terning t = new Terning();  
Point p;  
p = (Point) t; // Sprogfejl: Point og Terning er urelaterede
```

## 5.3 Eksempel: Et matador-spil

Med arv kan man skabe et hierarki af klasser, der ligner hinanden (fordi de har alle fællestrækkene fra superklassen) og samtidig kan opføre sig forskelligt (polymorfi).

Her er vist klassediagrammet fra et matadorspil. Det er en skitse; et rigtigt matadorspil ville indeholde flere detaljer.



Klassen Felt, øverst i diagrammet, indeholder fællestrækkene for alle matadorspillets felter. Alle felter har et navn (f.eks. "Hvidovrevej"), og derfor definerer vi variablen navn her.

```
/** Superklassen for alle matadorspillets felter */
public class Felt
{
    String navn;          // feltets navn, f.eks. "Hvidovrevej"

    /** kaldes når en spiller passerer dette felt */
    public void passeret(Spiller sp)
    {
        sp.besked("Du passerer "+navn);
    }

    /** kaldes når en spiller lander på dette felt */
    public void landet(Spiller sp)
    {
    }
}
```

Alle felter skal også kunne håndtere, at spilleren lander på eller passerer feltet. Vi forestiller os, at metoderne landet() og passeret() bliver kaldt af en anden del af programmet, når en spillers brik henholdsvis lander på eller passerer feltet. I Felt-klassen er metoderne defineret til ikke at gøre noget.

Bemærk hvordan vi får overført en reference til spilleren når passeret() og landet() kaldes. Vi skal bruge Spiller-objektet til at kalde metoder på spilleren (f.eks. stille spørgsmål eller overføre penge). Her er klassen Spiller med oplysningerne om spilleren:

```
/** Definition af en spiller */
public class Spiller
{
    String navn;           // spillerens navn, f.eks. "Søren"
    double konto;          // antal kroner på spillerens konto
    int feltnr;            // hvad nummer felt spilleren står på. "Start" er nummer 0

    public Spiller(String navn, double konto)
    {
        this.navn = navn;
        this.konto = konto;
        feltnr = 0;
    }

    /** En besked til spilleren */
    public void besked(String besked)
    {
        System.out.println(navn+": "+besked);
    }

    /** Et ja/nej-spørgsmål. Svarer brugeren ja returneres true, ellers false */
    public boolean spørgsmål(String spørgsmål)
    {
        String spm = navn+": Vil du "+spørgsmål+"?";
        String svar = javax.swing.JOptionPane.showInputDialog(spm, "ja");
        System.out.println(spm+" "+svar);
        if (svar!=null && svar.equals("ja")) return true;
        else return false;
    }

    public void transaktion(double kr)
    {
        konto = konto + kr;
        System.out.println(navn+"s konto lyder nu på "+konto+" kr.");
    }

    public void betal(Spiller modtager, double kr)
    {
        System.out.println(navn+" betaler "+modtager.navn+": "+kr+" kr.");
        modtager.transaktion(kr);
        transaktion(-kr);
    }
}
```

Under Felt har vi klasserne Helle, Start, Rederi og Gade, der indeholder data og programkode, som er specifik for de forskellige slags felter i matadorspillet. De arver alle fra Felt og er derfor tegnet med en **er-en**-relation til Felt i diagrammet.

Klassen Helle er simpel; den skal lægge 15000 kr. til spillerens kassebeholdning, hvis spilleren lander på feltet. Det gøres ved at tilsidesætte den nedarvede landet()-metode med en, der overfører penge til spilleren.

```
/** Helle - hvis man lander her får man en gevinst */
public class Helle extends Felt
{
    double gevinst;

    public Helle (int gevinst)
    {
        navn = "Helle";           // navn er arvet fra Felt
        this.gevinst = gevinst;
    }

    public void landet(Spiller sp) // tilsidesæt metode i Felt
    {
        sp.besked("Du lander på helle og får overført "+gevinst);
        sp.transaktion(gevinst);  // opdater spillers konto
    }
}
```

I konstruktøren sætter vi feltets navn. Gevinsten ved at lande her er en parameter til konstruktøren. Metodekaldet `sp.transaktion(gevinst)` beder spiller-objektet om at føje gevinsten til kontoen.

Klassen `Start` skal overføre 5000 kr. til spilleren, der passerer eller lander på feltet. Dette gøres ved at tilsidesætte både `landet()` og `passeret()`.

```
/** Startfeltet - når en spiller passerer dette felt, får han 5000 kr */
public class Start extends Felt
{
    double gevinst;

    public Start(double gevinst)
    {
        navn = "Start";
        this.gevinst = gevinst;
    }

    public void passeret(Spiller sp)                // tilsidesæt metode i Felt
    {
        sp.besked("Du passerer start og modtager "+gevinst);
        sp.transaktion(gevinst);                    // kredit/debit af konto
    }

    public void landet(Spiller sp)                  // tilsidesæt metode i Felt
    {
        sp.besked("Du lander på start og modtager "+gevinst);
        sp.transaktion(gevinst);
    }
}
```

Nu kommer vi til felter, der kan ejes af en spiller, nemlig rederier og gader. De har en ejer-variabel, der refererer til en `Spiller` (og er derfor tegnet med en **har**-relation til klassen `Spiller` i diagrammet), en pris og en leje for at lande på grunden.

```
/** Et rederi, der kan købes af en spiller */
public class Rederi extends Felt
{
    Spiller ejer;
    double pris;
    double grundleje;

    public Rederi(String navn, double pris, double leje)
    {
        this.navn = navn;
        this.pris = pris;
        this.grundleje = leje;
    }

    public void landet(Spiller sp)
    {
        sp.besked("Du er landet på "+navn);
        if (sp==ejer)
        {
            sp.besked("Det er din egen grund");        // spiller ejer selv grunden
        }
        else if (ejer==null)
        {
            if (sp.konto > pris)                        // ingen ejer grunden, så køb den
            {
                if (sp.spørgsmål("køb "+navn+" for "+pris))
                {
                    sp.transaktion( -pris );
                    ejer=sp;
                }
            }
            else sp.besked("Du har ikke penge nok til at købe "+navn);
        }
        else
        {
            sp.besked("Leje: "+grundleje);              // feltet ejes af anden spiller
            sp.betal(ejer, grundleje);                  // spiller betaler til ejeren
        }
    }
}
```

Når en spiller lander på et rederi, skal der overføres penge fra spilleren til ejeren af grunden. Dette gøres ved at tilsidesætte den nedarvede `landet()`-metode med en, der overfører beløbet mellem parterne. Først tjekkes, om spilleren er den samme som ejeren (`sp==ejer`). Hvis dette ikke er tilfældet, tjekkes, om der ingen ejer er (`ejer==null`) og hvis der ikke er, kan spilleren købe grunden (`ejer` sættes lig `spilleren`). Ellers beordres spilleren til at betale et beløb til ejeren: `sp.betal(ejer, grundleje)`.

Klassen `Gade` repræsenterer en byggegrund og objekter af type `Gade` har derfor, ud over `ejer`, `pris` og `grundleje`, en variabel, der husker, hvor mange huse der er bygget på dem.

Når en spiller lander på grunden, skal der ske nogenlunde det samme som for et Rederi bortset fra, at hvis det er ejeren, der lander på grunden, kan han bygge et hus.

```
/** En gade, der kan købes af en spiller og bebygges */
public class Gade extends Felt
{
    Spiller ejer;
    double pris;
    double grundleje;
    int antalHuse = 0;
    double huspris;

    public Gade(String navn, double pris, double leje, double huspris)
    {
        this.navn = navn;
        this.pris = pris;
        this.grundleje = leje;
        this.huspris = huspris;
    }

    public void landet(Spiller sp)
    {
        sp.besked("Du er landet på "+navn);

        if (sp==ejer)
        {
            // eget felt
            sp.besked("Det er din egen grund");
            if (antalHuse<5 && sp.konto>huspris && // bemærk: kun hvis betingelserne
                sp.spørgsmål("købe hus for "+huspris)) // er opfyldt stilles spørgsmålet
            {
                // byg et hus
                ejer.transaktion( -huspris );
                antalHuse = antalHuse + 1;
            }
        }
        else if (ejer==null)
        {
            // ingen ejer grunden, køb den?
            if (sp.konto > pris)
            {
                if (sp.spørgsmål("købe "+navn+" for "+pris))
                {
                    sp.transaktion( -pris );
                    ejer=sp;
                }
            }
            else sp.besked("Du har ikke penge nok til at købe "+navn);
        }
        else
        {
            // felt ejes af anden spiller
            double leje = grundleje + antalHuse * huspris;
            sp.besked("Leje: "+leje);
            sp.betal(ejer, leje); // spiller betaler til ejeren
        }
    }
}
```

Lad os nu lave en klasse, der har som ansvarsområde at repræsentere et helt matadorspil, med lister over alle felter og spillere og som holder styr på, hvis tur det er:

```
import java.util.*;

public class Matadorspil
{
    ArrayList<Felt> felter = new ArrayList<Felt>(); // indeholder alle felter
    ArrayList<Spiller> spillere = new ArrayList<Spiller>(); // alle spillere
    int spillersTur = 0;

    public Matadorspil()
    {
        felter.add(new Start(5000));
        felter.add(new Gade("Rødovrevej", 10000, 400,1000));
        felter.add(new Gade("Hvidovrevej",10000, 400,1000));
        felter.add(new Rederi("Maersk", 17000,4200));
        felter.add(new Gade("Gade 3", 12000, 500,1200));
        felter.add(new Gade("Gade 4", 12000, 500,1200));
        felter.add(new Gade("Gade 5", 15000, 700,1500));
        felter.add(new Helle(15000));
        felter.add(new Gade("Frederiksberg Allé", 20000,1100,2000));
        felter.add(new Gade("Rådhuspladsen", 20000,1100,2000));
    }
}
```

I konstruktøren bygges en liste med alle felterne på brættet op. Alle felterne behandles som objekter af typen Felt (selvom de egentlig er nedarvinger).

Et spil kan spilles ved at oprette et Matadorspil-objekt for at få et bræt og derpå lægge to spillere ind i spillerlisten. Herunder spilles også 20 runder, hvor hver spiller får en tur:

```
public class BenytMatadorspil
{
    public static void main(String[] arg)
    {
        Matadorspil spil = new Matadorspil();
        spil.spillere.add(new Spiller("Søren",50000)); // opret spiller Søren
        spil.spillere.add(new Spiller("Gitte",50000)); // opret spiller Gitte

        // løb gennem 20 runder (40 ture)
        for (spil.spillersTur=0; spil.spillersTur<40; spil.spillersTur++)
        {
            // tag skiftevis Søren og Gitte (% er forklaret i afsnit 2.11.4)
            Spiller sp = spil.spillere.get(spil.spillersTur % spil.spillere.size());
            int slag = (int)(Math.random()*6)+1; // og slå et terningkast (1-6)
            System.out.println("***** "+sp.navn+" på felt "+sp.feltnr+" slår "+slag);

            for (int i=1; i<=slag; i=i+1) // nu rykkes der
            {
                // gå til næste felt. Hvis vi når over antal felter så tæl fra 0
                sp.feltnr = (sp.feltnr + 1) % spil.felter.size();
                Felt felt = spil.felter.get(sp.feltnr);

                if (i<slag) felt.passeret(sp); // kald passeret() på passerede felter
                else felt.landet(sp); // kald landet() på sidste felt
                try { Thread.sleep(300); } catch (Exception e) {} // vent 0.3 sek
            }
            try { Thread.sleep(3000); } catch (Exception e) {} // tur slut, vent 3 sek
        }
    }
}
```

Når vi skal spille en tur, finder vi et tilfældigt tal mellem 1 og 6 og rykker spilleren frem. Vi kalder passeret() på hver af de passerede felter og landet() på den sidste.

Bemærk hvordan vi sørger for, at variabelen feltnr vedbliver at have en værdi mellem 0 og antallet af felter med operatoren %, der giver resten af en division (se afsnit 2.11.4).

Linjen nederst får programmet til at holde en pause i tre sekunder, inden det går videre (try og catch vil blive forklaret i kapitel 14, Undtagelser).



Her ses uddata fra en kørsel af programmet:

```
***** Søren på felt 0 slår 1
Søren: Du er landet på Rødvorevej
Søren: Vil du købe Rødvorevej for 10000.0? ja
Sørens konto lyder nu på 40000.0 kr.
***** Gitte på felt 0 slår 6
Gitte: Du passerer Rødvorevej
Gitte: Du passerer Hvidovrevej
Gitte: Du passerer Maersk
Gitte: Du passerer Gade 3
Gitte: Du passerer Gade 4
Gitte: Du er landet på Gade 5
Gitte: Vil du købe Gade 5 for 15000.0? ja
Gittes konto lyder nu på 35000.0 kr.
***** Søren på felt 1 slår 1
Søren: Du er landet på Hvidovrevej
Søren: Vil du købe Hvidovrevej for 10000.0? ja
Sørens konto lyder nu på 30000.0 kr.
***** Gitte på felt 6 slår 3
Gitte: Du passerer Helle
Gitte: Du passerer Frederiksberg Allé
Gitte: Du er landet på Rådhuspladsen
Gitte: Vil du købe Rådhuspladsen for 20000.0? ja
Gittes konto lyder nu på 15000.0 kr.
***** Søren på felt 2 slår 5
Søren: Du passerer Maersk
Søren: Du passerer Gade 3
Søren: Du passerer Gade 4
Søren: Du passerer Gade 5
Søren: Du lander på helle og får overført 15000.0
Sørens konto lyder nu på 45000.0 kr.
***** Gitte på felt 9 slår 1
Gitte: Du lander på start og modtager 5000.0
Gittes konto lyder nu på 20000.0 kr.
***** Søren på felt 7 slår 1
Søren: Du er landet på Frederiksberg Allé
Søren: Vil du købe Frederiksberg Allé for 20000.0? ja
Sørens konto lyder nu på 25000.0 kr.
***** Gitte på felt 0 slår 3
Gitte: Du passerer Rødvorevej
Gitte: Du passerer Hvidovrevej
Gitte: Du er landet på Maersk
Gitte: Vil du købe Maersk for 17000.0? ja
Gittes konto lyder nu på 3000.0 kr.
***** Søren på felt 8 slår 5
Søren: Du passerer Rådhuspladsen
Søren: Du passerer start og modtager 5000.0
Sørens konto lyder nu på 30000.0 kr.
Søren: Du passerer Rødvorevej
Søren: Du passerer Hvidovrevej
Søren: Du er landet på Maersk
Søren: Leje: 4200.0
Søren betaler Gitte: 4200.0 kr.
Gittes konto lyder nu på 7200.0 kr.
Sørens konto lyder nu på 25800.0 kr.
***** Gitte på felt 3 slår 3
Gitte: Du passerer Gade 3
Gitte: Du passerer Gade 4
Gitte: Du er landet på Gade 5
Gitte: Det er din egen grund
Gitte: Vil du købe hus for 1500.0? ja
Gittes konto lyder nu på 5700.0 kr.
***** Søren på felt 3 slår 2
Søren: Du passerer Gade 3
Søren: Du er landet på Gade 4
Søren: Vil du købe Gade 4 for 12000.0? ja
Sørens konto lyder nu på 13800.0 kr.
***** Gitte på felt 6 slår 1
Gitte: Du lander på helle og får overført 15000.0
Gittes konto lyder nu på 20700.0 kr.
```

... (og så videre)

## 5.3.1 Polymorfi

Polymorfi vil sige, at objekter af forskellig type bruges på en ensartet måde uden hensyn til deres præcise type.

Matadorspillet udnytter polymorfi til at behandle alle feltobjekter ens (ved at kalde `landet()` og `passeret()` fra `Spiller`'s `tur()`-metode), selvom de er af forskellig type.

Fidusen er, at programkoden i `tur()`-metoden kan skrives uafhængigt af, hvilke felt-typer der findes: De rigtige `landet()`- og `passeret()`-metoder i nedarvingerne vil automatisk blive kaldt, selvom `tur()` kun kender til `Felt`-klassen.

Polymorfi er et kraftfuldt redskab til at lave meget fleksible programmer, der senere kan udvides, uden at der skal ændres ret meget i den eksisterende kode.

For eksempel kan vi til enhver tid udbygge matadorspillet med flere felttyper uden at skrive programmet om. Den programkode, der arbejder på felterne, `Spiller`-klassens `tur()`-metode, kender faktisk slet ikke til andre klasser end `Felt`!

En forudsætning for at udnytte polymorfi-mekanismen er, at objekterne "sørger for sig selv", dvs. at data og programkode er i de objekter, som de handler om.

## 5.3.2 Opgaver

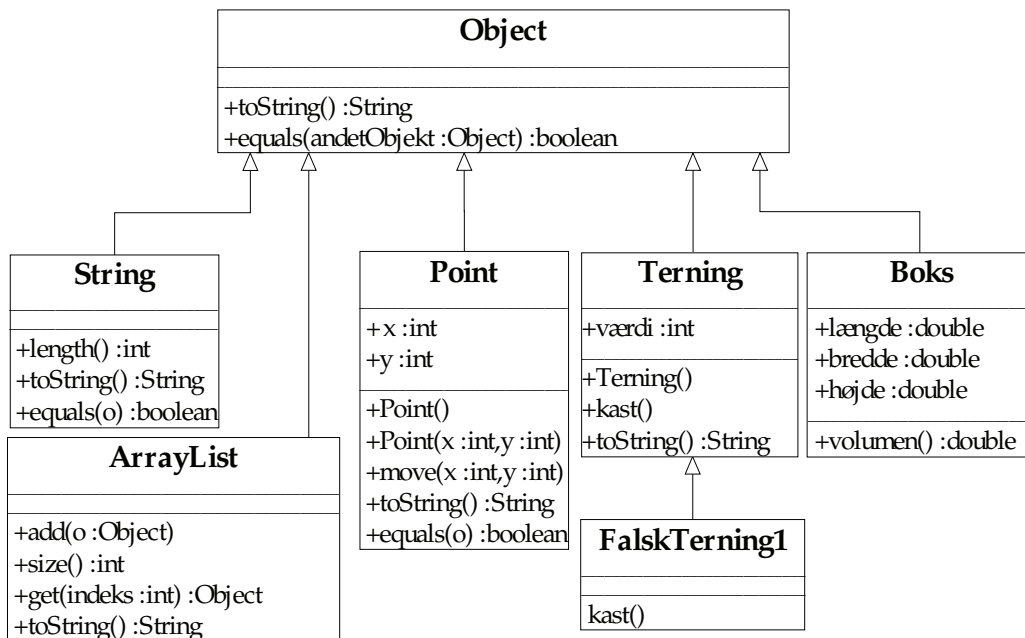
- 1) Tilføj en `Bryggeri`-klasse til matadorspillet og prøv, om det virker. På bryggerier afhænger lejen af, hvor stort et slag spilleren slog, da han landede på feltet, men lad foreløbig lejen være tilfældig.  
Du kan evt. kopiere `Gade.java` i stedet for at skrive koden forfra. Husk at indsætte et `Bryggeri` i `felter`-listen i `Matadorspil` (eller i `BenytMatadorspil`'s `main()`-metode).
- 2) Ret derefter, så lejen af et bryggeri afhænger af, hvor stort et slag spilleren slog.  
Tilføj nu objektvariablen `slag` til `Spiller` og brug den fra `BenytMatadorspil` i stedet for den lokale variabel (slet den for at undgå misforståelser). Nu kan man altid se, hvad spilleren slog sidst ved at kigge på værdien af `slag`. Udnyt dette i `landet()`-metoden i `Bryggeri` til at lade lejen afhænge af, hvad spilleren slog.

## 5.4 Stamklassen Object

Klassen Object (i pakken java.lang) er 'alle klassers moder', dvs. superklasse for alle andre klasser. Arver en klasse ikke fra noget andet, vil den automatisk arve fra Object.

### Alle klasser arver fra Object

Således arver f.eks. Terning fra Object. FalskTerning1 arver indirekte fra Object gennem Terning. Alle standardklasserne arver også fra Object, muligvis gennem andre klasser.



Det er derfor, at bl.a. `toString()`-metoden findes på alle objekter; den er defineret i **Object** og arves til alle klasser i Java. Her ses, hvad der sker, hvis man udskriver et (f.eks. **Boks**-) objekt, der ikke har sin egen `toString()`:

```
...
Boks b = new Boks();
System.out.println("b = "+b);    // b.toString() kaldes implicit
...
b = Boks@4852d1b0
```

Den bruger `toString()` fra **Object** og man kan altså se, at implementationen af `toString()` i **Object** returnerer klassens navn, et '@' og et tal<sup>1</sup>, f.eks. `Boks@4852d1b0`.

En anden metode i **Object** er `equals()`. Den har vi brugt til at undersøge, om strenge er ens, men den findes altså på ethvert objekt og kan f.eks. også bruges til at undersøge, om to lister indeholder de samme elementer. Bemærk, at man selv skal definere `equals()`-metoden på sine egne klasser, før den fungerer som forventet<sup>2</sup>.

- 1 En såkaldt hashkode, der med stor sandsynlighed er unik. Dette er ikke specielt informativt, så man definerer ofte sin egen `toString()`
- 2 Man kan altid kalde `equals()`, men det kan godt være, at den ikke giver det forventede, hvis den ikke er defineret i klassen. **ArrayList** og **String** har defineret den, mens f.eks. **StringBuilder** ikke har. Da får man opførslen fra **Object**, der kun undersøger, om objekterne er de samme (samme sted i hukommelsen), den tager ikke højde for, om to objekter indeholder ens data.

## 5.4.1 Referencer til objekter

Når alle objekter arver fra Object, kan man i en variabel af denne type gemme en reference til ethvert slags objekt, jf. reglerne om typekonvertering:

```
Object obj;  
obj = new Point();  
obj = "hej";  
obj = new FalskTerning2();
```

Omvendt kan man ikke rigtig bruge variablen til noget, før man har lavet en eksplicit typekonvertering af referencen obj:

```
Terning t;  
t = (Terning) obj;  
t.kast();
```

Her var vi heldige, at obj faktisk refererede til en (underklasse af) Terning. Vi får først at vide, om typekonverteringen er gået godt på kørselstidspunktet.

Tilsvarende kan man bruge Object som parameter/returtype og få et fleksibelt, men ikke særlig sikkert program. Klassen ArrayList benytter sig af dette: Den er en liste af typen Object, det er derfor, man kan gemme alle slags objekter i den.

```
ArrayList l = new ArrayList(); // kunne også skrive new ArrayList<Object>()  
Point p = new Point();  
l.add(p);
```

Metoden add() får noget af typen Object (dvs. et hvilket som helst objekt) som parameter.

Nedenfor er vist det samme, men her bruges en mellemvariabel, der illustrerer, at der sker en implicit reference-typekonvertering (p skal jo konverteres fra en Point-reference til en Object-reference):

```
ArrayList l = new ArrayList(); // liste med alle slags objekter (type Object)  
Point p = new Point();  
Object obj; // overflødig mellemvariabel  
obj = p; // implicit reference-typekonvertering  
l.add(obj);
```

Når man kalder get() for at få fat i objektet igen, er det nødvendigt med en eksplicit reference-typekonvertering, fordi konverteringen sker den anden vej, fra superklasse til underklasse:

```
p = (Point) l.get(0);
```

Igen vises det samme blot med mellemvariabel, så man kan se, hvilken typekonvertering der finder sted:

```
o = l.get(0); // ingen konvertering  
p = (Point) o; // eksplicit reference-typekonvertering
```

## 5.5 Konstruktører i underklasser

Vi minder om, at:

- Konstruktører definerer, hvordan objekter oprettes og med hvilke parametre de må oprettes. Der kan kun oprettes objekter på en måde, der passer med en konstruktør.
- Hvis en klasse ikke har defineret nogen konstruktører, så defineres automatisk en standardkonstruktør (uden parametre og med tom metodekrop). F.eks. har Boks automatisk fået en tom konstruktør, så den kunne oprettes med `new Boks()`.

Underklassen skal selv definere, hvordan dets objekter skal kunne oprettes, så den skal selv definere sine konstruktører. Den kan have færre eller flere konstruktører end superklassen.

Når man definerer en konstruktør på en underklasse, skal man kun initialisere den nye del af objektet. Har man f.eks. tilføjet nye variabler, skal konstruktøren initialisere dem.

Den arvede del af objektet initialiseres ved, at man fra underklassens konstruktør kalder en konstruktør fra superklassen. Dette gøres med sætningen: `"super(...);"` med eventuelle parametre. Man bruger altså her **super** som en metode. Det skal gøres som den **første** sætning i konstruktøren. Hvis man ikke selv kalder `super()` som det første, sker der det, at `super` bliver kaldt automatisk **uden** parametre.

---

**Konstruktører skal defineres separat i en underklasse**

**En konstruktør kalder først en af superklassens konstruktører**

**Superklassens konstruktør kan kaldes med: `super(parametre)`**

**Hvis man ikke selv kalder en af superklassens konstruktører, indsætter Java automatisk et kald af superklassens konstruktør uden parametre**

---

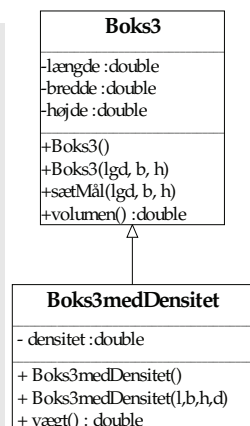
Herunder definerer vi `Boks3medDensitet`, der arver fra `Boks3` (se afsnit 4.3). Det nye er variabelen `densitet` og metoden `vægt()`. Objektet kan oprettes med: `new Boks3medDensitet()`, som opretter boksen med standardværdier eller med: `new Boks3medDensitet(lgd,b,h,d)`, hvor man angiver længde, bredde, højde og massefylde (densitet).

```
public class Boks3medDensitet extends Boks3
{
    private double densitet;

    public Boks3medDensitet()
    {
        // super(); kaldes automatisk hvis intet andet angives
        densitet = 10.0;
    }

    public Boks3medDensitet(double lgd, double b,
        double h, double densitet)
    {
        // vælg en anden konstruktør i superklassen
        // end den uden parametre
        super(lgd,b,h);
        this.densitet = densitet;
    }

    public double vægt()
    {
        // superklassen udregner volumen for os
        return volumen() * densitet;
    }
}
```



*Boks3medDensitet tillader oprettelse på andre måder end superklassen Boks3*

## 5.5.1 Konsekvenser

Ovenstående regler kombineret med reglerne for standardkonstruktøren har nogle pudsige konsekvenser. Lad os se på et eksempel med al overflødig kode skåret væk:

```
public class A
{
    public A(int i) { }
}
```

```
public class B extends A
{
}
```

Klassen B vil ikke oversætte, fordi den af Java vil blive lavet om til:

```
public class B extends A
{
    public B() // standardkonstruktør indsættes automatisk af Java
    {
        super();
    }
}
```

Standardkonstruktøren i B vil altså prøve at kalde konstruktøren i A uden parametre, men den findes jo ikke, fordi A har en anden konstruktør. Oversætteren kommer med fejlmeddelelsen "*constructor A() not found*". Så er vi nødt til at angive konstruktøren i superklassen:

```
public class B extends A
{
    public B() // vi er nødt til at definere underklassens konstruktør...
    {
        super(42); // ... så vi kan angive parametre til superklassens konstruktør
    }
}
```

Hvis vi ønsker en konstruktør med de samme parametre i B som i A, vil det se således ud:

```
public class B extends A
{
    public B(int i) // vi er nødt til at definere underklassens konstruktør...
    {
        super(i); // ... så vi kan angive parametre til superklassens konstruktør
    }
}
```

Der er til gengæld ingen problemer med:

```
public class A2
{
}
```

og

```
public class B2 extends A2
{
}
```

Java laver det om til:

```
public class A2
{
    public A2() { } // indsættes automatisk af Java
}
```

og

```
public class B2 extends A2
{
    public B2() // indsættes automatisk af Java
    {
        super();
    }
}
```

## 5.6 Ekstra eksempler

### 5.6.1 Matadorspillet version 2

Dette eksempel viser, hvordan man kan spare programkode (og dermed programmeringstid) med nedarvning. Samtidig viser det brugen af konstruktører i underklasser.

Se igen på programkoden til Rederi og Gade. Der er meget programkode, som er ens for de to klasser. Faktisk implementerer de kode, der er fælles for alle grunde, der kan ejes af en spiller og derfor vil det være hensigtsmæssigt, at følgende kode var i en Grund-klasse:

- Definition og initialisering af variablerne `pris`, `grundleje`, `ejer`.
- Håndtering af, at en spiller lander på grunden (bl.a. betaling af leje).
- Håndtering af, at en spiller lander på en grund, der ikke ejes af nogen (køb af grunden).

Det har vi gjort herunder.

Vi har været forudseende og flyttet beregningen af lejen ud fra `landet()` og ind i en separat metode `beregnLeje()`, fordi netop denne er meget forskellig for Rederi og Gade.

```
/** Mellemklasse mellem 'Felt' og underliggende klasser som Gade og Rederi */
public class Grund2 extends Felt
{
    Spiller ejer;
    double pris;
    double grundleje;

    public Grund2(String navn, double pris, double leje)
    {
        this.navn=navn;
        this.pris=pris;
        this.grundleje=leje;
    }

    public double beregnLeje()
    {
        return grundleje;
    }

    public void landet(Spiller sp)
    {
        sp.besked("Du er landet på "+navn);
        if (sp==ejer)
        {
            // spiller ejer feltet
            sp.besked("Det er din egen grund");
        }
        else if (ejer==null)
        {
            // ingen ejer grunden, så køb den
            if (sp.konto > pris)
            {
                if (sp.spørgsmål("købe "+navn+" for "+pris))
                {
                    sp.transaktion( -pris );
                    ejer=sp;
                }
            }
            else sp.besked("Du har ikke penge nok til at købe "+navn);
        }
        else
        {
            // felt ejes af anden spiller
            // udregn lejen
            double leje = beregnLeje();
            sp.besked("Leje: "+leje);
            // spiller betaler til ejeren
            sp.betal(ejer, leje);
        }
    }
}
```

Nu er Rederi ret nem. Den skal nemlig (i denne simple udgave) opføre sig præcis som Grund2. Vi skal blot definere konstruktøren, som kalder den samme konstruktør i Grund2:

```
/** Et rederi, der kan købes af en spiller */  
public class Rederi2 extends Grund2  
{  
    public Rederi2(String navn, double pris, double leje)  
    {  
        super(navn, pris, leje); // overfør værdierne til superklassens konstruktør  
    }  
}
```

Nu kommer vi til Gade. Her er beregnLeje() tilsidesat til også at tage højde for antallet af huse. Med **super** kan vi faktisk spare en hel del arbejde. Gaderne kan genbruge meget af landet()-metoden, men der er dog en ekstra mulighed for at bygge hus. Derfor kalder vi superklassens landet()-metode. Derefter, hvis spilleren, der er landet på gaden, er ejeren, prøver vi at bygge et hus.

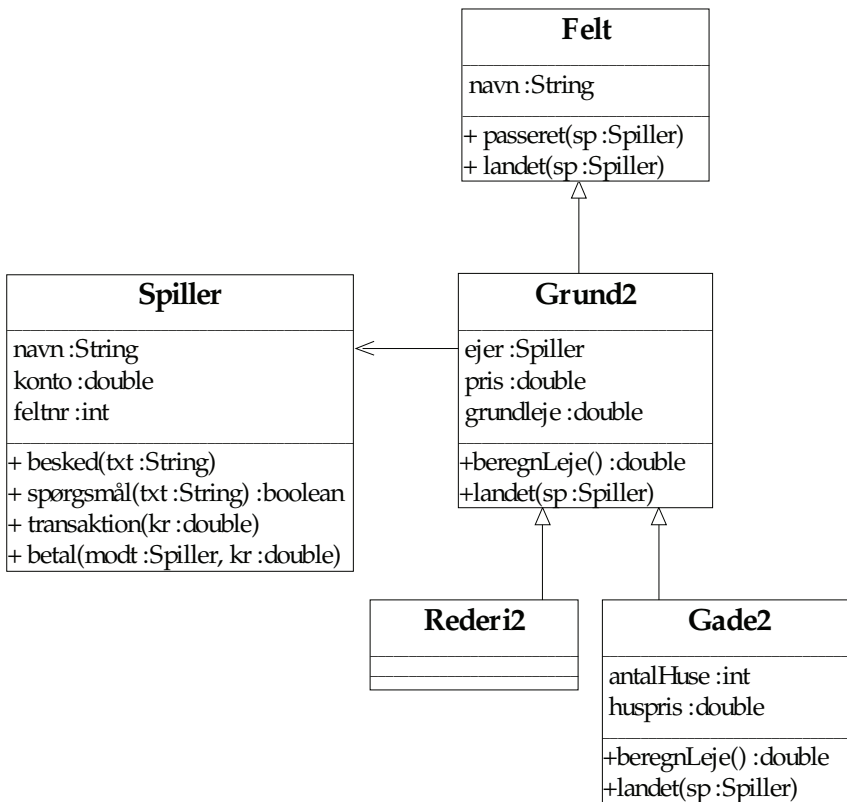
```
/** En gade, der købes af en spiller og bebygges */  
public class Gade2 extends Grund2  
{  
    int antalHuse; // antal huse og pris  
    double huspris;  
  
    public Gade2(String navn, double pris, double leje, double huspris)  
    {  
        super(navn, pris, leje); // kald Grund2's konstruktør  
        this.huspris=huspris;  
        antalHuse = 0;  
    }  
  
    public double beregnLeje() // tilsidesæt Grund2's beregnLeje()  
    {  
        return grundleje + antalHuse * huspris;  
    }  
  
    public void landet(Spiller sp)  
    {  
        super.landet(sp); // brug gamle landet()  
        if (sp==ejer)  
        {  
            // eget felt; byg hus?  
            if (antalHuse<5 && sp.konto>huspris && sp.spørgsmål("købe hus for "+pris))  
            {  
                // byg et hus  
                ejer.transaktion( -huspris );  
                antalHuse = antalHuse + 1;  
                sp.besked("Du bygger hus på "+navn+" for "+huspris);  
            }  
        }  
    }  
}
```

Denne måde at programmere på kan spare meget kode: Læg mærke til, at vi har sparet næsten halvdelen af koden væk i de to nye klasser!



Herunder ses klassediagrammet for de nye klasser.

Da Grund2 **har en** spiller (ejer), er der en pil fra Grund2 til Spiller, en **har**-relation. Resten af pilene symboliserer **er-en**-relationer, f.eks. Gade2 **er en** Grund2, Grund2 **er et** Felt.



I kapitel 22, Objektorienteret analyse og design kan du læse mere om, hvordan man designer sine klasser og hvordan man vælger hvilke er-en- og har-en-relationer, der skal være mellem klasserne.

## 5.6.2 Opgaver

- 1) Ret i `Matadorspil.java` til at bruge `Rederi2` og `Gade2` i stedet for `Rederi` og `Gade`. Kør programmet og følg med i, hvad der sker i `Gade2`'s konstruktør og `landet()`-metode.
- 2) Føj en `Bryggeri`-klasse til `matadorspillet` version 2. Husk at kalde `super()` i konstruktøren (hvorfor er det nødvendigt?). Du kan evt. kopiere `Gade2.java` i stedet for at skrive koden forfra. Hvor meget kode kan du spare?

## 5.7 Test dig selv

- 1) Hvad er nedarvning? Lav et eksempel på nedarvning.
- 2) Hvilke objekter er generelt mindst, superklassens eller underklassens?
- 3) Hvordan tilsidesætter man metoder?
- 4) Hvad er super? Nævn to forskellige slags brug af super.
- 5) Bliver konstruktører også nedarvet?
- 6) Hvordan bestemmer man, hvilken af superklassens konstruktører der skal bruges?
- 7) Hvornår er det, at superklassens konstruktør *skal* kaldes eksplicit ?
- 8) Hvis `t` har typen `Terning` og `ft` har typen `FalskTerning2`, kan man så skrive `t=ft`; ?  
Kan man skrive `ft=t`; ? Hvad er reglen?
- 9) Hvad er polymorfi?
- 10) Hvad er klassen `Object`? Hvilke metoder har den?

## 5.8 Resumé

- En klasse (underklassen) kan arve metoder og variabler fra en anden klasse (superklassen) vha. "extends". Ekstra metoder og variabler kan føjes til underklassen. Underklassen er derfor generelt større (fylder mere i hukommelsen) end superklassen.
- Man kan tilsidesætte en metode fra superklassen ved at definere den igen i underklassen. Denne metode vil så blive kaldt i stedet for superklassens version.
- Variablens type bestemmer, hvilke metoder man kan kalde og hvilke objektvariabler man kan anvende udefra.
- Objektets type (hvilken klasse den er skabt ud fra) bestemmer, hvilken metodekrop der udføres.
- Med `super` kan man kalde superklassens metoder, selvom de er tilsidesat.
- Man kan tildele en variabel af superklassen en reference til et objekt af en underklasse. Det kaldes implicit reference-typekonvertering og Java garanterer, at det går godt.
- Man kan tildele en variabel af underklassen en reference til et objekt af superklassen, hvis man bruger en eksplicit konvertering, f.eks.: `t = (Terning) l.get(3)`;
- Eksplicit typekonvertering (eng.: `cast`) kan gå galt under programkørslen og undtagelsen `ClassCastException` kan opstå.
- Konstruktører i underklassen defineres uafhængigt (evt. med andre parametre) af superklassens.
- Konstruktøren i en underklasse kalder som det første en konstruktør i superklassen. Hvis man ikke selv gør det vha. `super(...)`, indsætter Java selv et kald.
- Polymorfi vil sige, at objekter af forskellig type bruges på en ensartet måde.
- Alle klasser arver fra `Object`, der bl.a. har metoderne `toString()` og `equals()`. Arver man ikke fra noget, skriver Java automatisk "extends `java.lang.Object`".

## 5.9 Avanceret

### 5.9.1 Initialisering uden for konstruktørerne

Al initialisering behøver som bekendt ikke at foregå i konstruktører.

Når man erklærer en objektvariabel, kan man give den en initialværdi i samme linje. Det kan være en fordel, hvis man har flere konstruktører, der altid skal sætte en variabel til noget bestemt.

Initialiseringen uden for konstruktørerne sker altid før konstruktøren kaldes, så linjen mærket p1 udføres før p2.

```
import java.awt.*;
public class InitA
{
    public InitA()
    {
        p.x = 10; // p2 (udføres after p1)
    }

    Point p = new Point(); // p1 (udføres først)
}
```

Man kan også have hele små kodestumper uden for konstruktørerne ved at indkapsle dem i blok-parenteser:

```
import java.awt.*;
import java.util.*;
public class InitB extends InitA
{
    ArrayList<Point> l = new ArrayList<Point>(); // p3

    { // initialiseringsblok
        l.add(new Point(4,4)); // p4 udføres før konstruktøren p5
    }

    public InitB()
    { // p5
        l.add(new Point(5,5));
    }
}
```

Disse to former for initialisering sker i rækkefølgen oppefra og ned og efter at superklassens konstruktører kaldes, men før klassens konstruktører selv udføres.

Havde man i et program placeret sætningen...

```
InitB b = new InitB();
```

...ville punkterne i programmet blive udført i rækkefølgen: p1, p2, p3, p4 og til sidst p5.

## 5.9.2 Kald af en konstruktør fra en anden konstruktør

Man kan faktisk også kalde en konstruktør fra en anden konstruktør i samme klasse. Man skriver så `this(...)` med de ønskede parametre i stedet for `super(...)`.

Kaldet til `this(...)` skal stå som det første i konstruktøren (og det er så den anden konstruktør, der bestemmer, hvilken af superklassens konstruktører der skal kaldes). Det er en fejl både at bruge `super(...)` og `this(...)` i samme konstruktør.

```
import java.awt.*;
import java.util.*;
public class C
{
    ArrayList<Point> l;

    public C(int x, int y)
    {
        l = new ArrayList<Point>(); // opret en liste af punkter
        l.add(new Point(x,y));      // tilføj punkt med koordinater x,y
    }

    public C()
    {
        this(2,2); // kalder den anden konstruktør så listen oprettes med punkt 2,2
    }
}
```

Man kan bruge dette til at undgå at kopiere kode i konstruktørerne.

## 5.9.3 Metoder erklæret final

Hvis man erklærer en metode final, kan metoden ikke tilsidesættes i en nedarvning. Ydermere sker kald til metoden en anelse hurtigere, netop fordi den virtuelle maskine ikke behøves tjekke, om metoden er tilsidesat. Det bruges typisk til små metoder, som skal være hurtige og hvor det ikke giver mening at tilsidesætte dem. Eksempel:

```
public void final kast()
{
    værdi=6;
}
```

## 5.9.4 Metoder erklæret protected

Nogle gange vil man gerne beskytte en metode ved at erklære den private. Imidlertid ønsker man stadig, at underklasser skal kunne kalde metoden, hvilket jo ikke kan lade sig gøre, hvis metoden er private. Hvis man erklærer en metode protected i stedet for private, får man det ønskede.

Man skal dog være opmærksom på, at alle klasser i samme pakke stadig kan kalde metoden (se afsnit 6.9).

## 5.9.5 Variabel-overskygning

Reglerne for tilsidesættelse gælder kun for metoder. Hvis man har en variabel i klassen, der hedder det samme som en variabel i superklassen, afsættes der plads til begge variabler i objektet. De metoder i underklassen, der læser eller ændrer i variabelen, bruger underklassens version af variabelen, mens superklassens metoder bruger superklassens version.

---

**Man kan ikke tilsidesætte en variabel fra en superklasse i en nedarvning.**

**Variabel-overskygning sker i 90 % af alle tilfælde ved en fejl.**

---

I de sidste 10% kunne man have brugt et andet variabelnavn med samme virkning.

# 6 Pakker

Indhold:

- Forstå pakkebegrebet og nøgleordet import
- Importere og bruge standardpakkerne
- Definere egne pakker

Kapitlet forudsættes ikke i resten af bogen, men er ofte en fordel, når man skal programmere i praksis.

Forudsætter kapitel 4, Definition af klasser.

Når man laver større programmer (over 30-40 klasser), kan det være nyttigt at opdele dem i grupper. En pakke er en samling af klasser, der på en eller anden måde er beslægtede.

---

### En pakke er en navngiven samling af klasser

---

Javas standardbibliotek på mere end 1000 klasser er delt op i ca. 30 mindre pakker.

Pakker svarer til (klasse)biblioteker i C eller C++ eller "unit"-begrebet i PASCAL.

## 6.1 At importere klassedefinitioner

Vi har set, at når vi skal benytte klasser, der ligger ud over de helt grundlæggende, bliver vi nødt til at meddele oversætteren, hvor den kan forvente at finde definitionen af klassen. Dette kaldes at importere klassen.

Egentlig kunne vi godt helt udelade import-sætninger og skrive det fulde pakke- og klassenavn hver gang. Hvis vi f.eks. vil benytte ArrayList-klassen, kunne vi skrive:

```
java.util.ArrayList l;  
l = new java.util.ArrayList();
```

Det er jo lidt besværligt og derfor kan vi vælge øverst i kildetekstfilen at skrive:

```
import java.util.ArrayList;
```

Dette får oversætteren til at lede i java.util-pakken, hvis den møder en klasse, den ikke umiddelbart genkender. Nu kan vi skrive, som vi plejer:

```
ArrayList l;  
l = new ArrayList();
```

Der kan forekomme et hvilket som helst antal import-sætninger i en javafil. Import-sætninger skal stå først i filen, før klassedefinitionen. Hvis man ønsker at importere flere klassedefinitioner fra samme pakke, kan man skrive en \* i stedet for klassenavnet:

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

Dermed importerer man samtlige klasser fra denne pakke. Det vil sige, at oversætteren leder denne pakke igennem, når den møder en klasse, den ikke umiddelbart genkender. De klassedefinitioner, der ikke bruges, bliver altså bare ignoreret.

---

**Import af en klasse gør blot definitionen af klassen kendt for oversætteren – det gør ikke det færdige program større eller langsommere**

---

## 6.2 Standardpakkerne

I Javas indbyggede hjælpesystem kan man se de forskellige indbyggede pakker, der indeholder en række nyttige klasser. De vigtigste standardpakker er:

- java.lang grundfunktioner i sproget
- java.util nyttige værktøjer, såsom Date, ArrayList og meget andet
- java.awt Abstract Window Toolkit. Grafikprogrammering (se kapitel 9 og 11)
- java.io IO-funktioner, filhåndtering og datastrømme (se kapitel 15)
- java.net netværksfaciliteter (se kapitel 16)
- java.rmi Remote Method Invocation – til distribuerede systemer (se kapitel 19)
- java.sql databaseadgang (også kaldet JDBC, se kapitel 20)
- java.text håndtering af tekst uafhængigt af sprog
- javax.swing avanceret vinduesbaseret programmering (se avanceret-afsnit i kapitel 11)

Hvorfor hedder den sidste javax? javax betød oprindeligt, at det var en udvidelse til sproget, som ikke var en del af det egentlige standardbibliotek og som måske aldrig bliver det. Efterhånden er en del javax-pakker (som javax.swing) dog alligevel kommet med.

Et andet eksempel er javax.comm, som er en kommunikationspakke, der håndterer seriel og parallel transmission af data. Denne pakke er ikke kommet med i standardbiblioteket.

### 6.2.1 Pakken java.lang

De mest basale javaklasser, eksempelvis String, ligger i pakken java.lang. Denne særlige pakke indeholder en masse grundfunktioner og importeres altid af oversætteren. Det er altså ikke nødvendigt at importere den eksplicit med import java.lang.\*;

Af andre klasser i java.lang kan nævnes System (til f.eks. System.out.println()) og Math (til f.eks. Math.random() og Math.sqrt()).

## 6.3 Pakkers placering på filsystemet

Hvis vi husker, at en pakke er en navngiven samling af klasser, er det nærliggende at tænke på, hvordan filer er organiseret i undermapper på et filsystem.

---

**En klasse svarer til en fil på filsystemet**

---

**En pakke svarer til en undermappe på filsystemet**

---

For eksempel findes klassen java.util.ArrayList som filen ArrayList.class i en mappe, der hedder util, som er en undermappe til en mappe, der hedder java: java/util/ArrayList.class (i Windows: java\util\ArrayList.class).

Ofte er .class-filerne og mapperne pakket i et såkaldt Java-arkiv (.jar-fil). jar-filer minder meget om zip-filer.

Oversætteren skal kende pakkens fysiske placering i filsystemet:

1. Som en undermappe med samme navn som pakken.
2. I en undermappe med samme navn som pakken et andet sted i filsystemet, som der henvises til med CLASSPATH-variablen.
3. I en jar-fil, som der henvises til med CLASSPATH-variablen.

CLASSPATH-variablen er en miljøvariabel, der minder om PATH-variablen (defineret i AUTOEXEC.BAT i DOS). Den angiver de steder, hvor oversætteren skal lede efter klasse-definitioner.

## 6.4 At definere egne pakker

Man kan definere sine egne pakker. Dette er specielt brugbart i større systemer, hvor man har mange klasser med beslægtede funktioner, for eksempel kommunikation (internetkøb med VISA eller Dankort) eller sine egne matematik- eller databehandlingspakker.

Det er normalt at man benytter sin internetadresse eller firmanavn i navngivningen af pakkerne. F.eks: oracle.JDeveloper.layout.XYLayout (klassen er XYLayout og pakken er oracle.JDeveloper.layout), com.sybase.jdbc.SybDriver eller netscape.javascript.JSObject.

### 6.4.1 Eksempel

I følgende eksempel findes to klasser, nemlig Klasse1 og Klasse2 i en pakke (der hedder minPakke). De bruges af den eksekverbare main()-klasse BenytPakker:

```
import minPakke.*;

public class BenytPakker
{
    public static void main(String[] arg)
    {
        Klasse1 a = new Klasse1();
        Klasse2 b = new Klasse2();
        a.snak();
        b.snak();
    }
}
```

Klasse1 og Klasse2 skal ligge i en undermappe, der hedder minPakke:

```
// Filnavn: minPakke/Klasse1.java
package minPakke;

public class Klasse1
{
    public void snak()
    {
        System.out.println("Dette er Klasse1, der taler!");
    }
}
```

```
// Filnavn: minPakke/Klasse2.java
package minPakke;

public class Klasse2
{
    public void snak()
    {
        System.out.println("Dette er Klasse2, der taler!");
    }
}
```

## 6.5 Pakke klasser i jar-filer (Java-arkiver)

Laver man sine egne pakker, ønsker man ofte at kunne distribuere de oversatte .class-filer med nyttekoden til andre. Det gøres nemmest ved at pakke filerne i en jar-fil.

En jar-fil skabes med et zip-værktøj som WinZip eller GnoZip til Linux eller fra kommandolinjen med kommandoen **jar**, der følger med, når man installerer Java. Den minder meget om UNIX' tar-kommando. Man opretter et arkiv ved at skrive f.eks.:

```
jar cf program.jar *.class minPakke
```

Dette vil oprette jar-filen program.jar med alle class-filer i mappen (herunder BenytPakker.class) og alle filerne i undermappen minPakke (d.v.s. minPakke/Klasse1.class og minPakke/Klasse2.class).



Derefter kan main()-metoden i BenytPakker udføres ved at blot skrive

```
java -cp program.jar BenytPakker
```

## 6.5.1 Eksekverbare jar-filer

Hvis man vil distribuere sit program til mange mennesker, kan det være bekvemt, at brugerne kan dobbeltklikke på jar-filen, lige som f.eks. Windows-brugere dobbeltklikker på .exe-filer. Det kræver, at man lægger en såkaldt manifest-fil med ned i jar-filen:

```
jar cfm program.jar manifestfil.txt *.class minPakke
```

Filen manifestfil.txt angiver klassen med main()-metoden:

```
Manifest-Version: 1.0  
Main-Class: BenytPakker
```

Der er et par irriterende småting der kan give problemer, herunder fil- og mappeplacering og at manifestfilen skal afslutte med et linjeskift. Mange bruger derfor deres udviklingsværktøj til at lave jar-filen (JBuilder: File | New | Archive | Application) eller ændrer en eksisterende jar-fil med WinZip eller et andet værktøj der kan rette i ZIP-filer.

Til sidst kan man køre BenytPakker ved at at dobbeltklikke på program.jar (hvis filassociationerne er sat korrekt op). Ellers kan man fra kommandolinjen starte programmet med:

```
java -jar program.jar
```

## 6.6 Test dig selv

- 1) Hvad er en pakke?
- 2) Hvordan bruger man klasser fra en pakke?
- 3) Hvad er sammenhængen mellem pakkenavn og placering på filsystemet?

## 6.7 Resumé

- 1) En pakke er en samling af klasser. Klasserne har givetvis et eller andet til fælles.
- 2) Skal man bruge klassen Xx i pakken yy.zz, kan man enten skrive det fulde pakkenavn foran klassenavnet hver gang den bruges (som yy.zz.Xx), eller man kan importere den øverst i filen (med import yy.zz.Xx;) og derefter behøver man kun skrive klassenavnet. Med import yy.zz.\* importerer man alle klasser i pakken.
- 3) Klasserne i pakken yy.zz skal ligge i undermappen yy/zz/ på filsystemet.

## 6.8 Opgaver

- 1) Søg i din computer efter filer, der ender på .jar og åbn dem med et program, der kan læse ZIP-komprimerede filer (f.eks. unzip eller WinZip).  
Hvordan er filerne organiseret?
- 2) Se, om du kan finde filen, der indeholder ArrayList-klassen.  
I hvilken undermappe ligger den?
- 3) Opret nogle klasser i en pakke og en main()-klasse, der benytter dem (f.eks. BenytPakker.java, minPakke/Klasse1.java og minPakke/Klasse2.java).
- 4) Pak dem i en jar-fil.
- 5) Prøv at oprette en eksekverbar jar-fil. Prøv derefter at køre den.

Til de sidste tre opgaver kan det være en fordel at gå ind på den elektroniske udgave af bogen, <http://javabog.dk> og kopiere teksten i stedet for at taste den ind.

## 6.9 Avanceret: public, protected og private

Det er vigtigt at styre adgangen til at kalde metoder og ændre på variable, i særdeleshed når programmerne bliver store. Det kan lette overskueligheden meget hvis interne variable og metoder er skjult uden for klassen.

### 6.9.1 Variabler og metoder

Variabler og metoder erklæret **public** er altid tilgængelige inden for og uden for klassen.

Variabler og metoder erklæret **protected** er tilgængelige for alle klasser inden for samme pakke. Klasser i andre pakker kan kun få adgang, hvis de er nedarvinger.

Skriver man **ingenting**, er det kun klasser i samme pakke, der har adgang til variablen eller metoden.

Hvis en variabel eller metode er erklæret **private**, kan den kun benyttes inden for samme klasse (og kan derfor ikke tilsidesættes med nedarvning). Det er det mest restriktive.

Adgangen kan sættes på skemaform:

Adgang	public	protected	(ingenting)	private
i samme klasse	ja	ja	ja	ja
klasse i samme pakke	ja	ja	ja	nej
nedarving i en anden pakke	ja	ja	nej	nej
ej nedarving og i en anden pakke	ja	nej	nej	nej

### 6.9.2 Indkapsling med pakker

Ud af de ovenstående regler kan man se, at adgangskontrol ud over public/private først bliver interessant, når programmet spænder over flere pakker. Så kan klasserne inden for samme pakke virke nært sammen (f.eks. ændre i hinandens interne variable), mens adgangen fra klasserne i de andre pakker er begrænset.

For at indkapsle en gruppe klasser, sådan at de kan tilgå hinandens metoder, mens disse metoder ikke er synlige udefra, er man således nødt til at lægge dem i en pakke for sig

---

**Indkapsle af en gruppe klasser sker ved at lægge dem i en pakke for sig**

---

### 6.9.3 Klasser

Klasser kan erklæres public eller ingenting (men ikke protected eller private).

Klasser erklæres normalt public og er tilgængelige fra alle pakker.

```
public class X
{
    // ...
}
```

Undlader man public, er klassen kun tilgængelig inden for samme pakke.

```
class X
{
    // ...
}
```

Man kan godt have flere klasser i en fil, men højst en, der er public. Denne klasse skal hedde det samme som filnavnet.

# 7 Lokale, objekt- og klassevariabler

Indhold:

- Klassevariable
- Repetition af objektvariabler og lokale variabler
- Rekursion

Forudsættes ikke i resten af bogen.

Forudsætter kapitel 4, Definition af klasser.

De variabler, vi er stødt på indtil nu, har enten været lokale variabler eller objektvariabler.

Objektvariabler hedder sådan, fordi de bliver oprettet for hvert objekt.

Der findes også variabler, der eksisterer "i klassen", uafhængigt af om der bliver oprettet objekter. Disse kaldes *klassevariabler* og erklæres med nøgleordet **static** (derfor kaldes de ofte også for "statiske variabler").

## 7.1 Klassevariabler og -metoder

Herunder ses et eksempel på en klassevariabel og en klassemetode (antalBokse).

Klassevariabler og -metoder vises med understregning i UML-notationen (diagrammet til højre).

```
public class Boks4
{
    private double længde;           // objektvariabel
    private double bredde;           // objektvariabel
    private double højde;            // objektvariabel
    private static int antalBokse;   // klassevariabel

    public Boks4(double lgd, double b, double h)
    {
        // lgd, b og h er lokale variabler
        længde = lgd;
        bredde = b;
        højde = h;
        antalBokse = antalBokse + 1;
    }

    public static int læsAntalBokse() // klassemetode
    {
        return antalBokse;
    }

    public double volumen()
    {
        // vol er en lokal variabel
        double vol;
        vol = længde*bredde*højde;
        return vol;
    }
}
```

Boks4
<u>-antalBokse :int</u>
<u>-længde :double</u>
<u>-bredde :double</u>
<u>-højde :double</u>
<u>+Boks4(lgd,b,h :double)</u>
<u>+læsAntalBokse() :int</u>
<u>+volumen() :double</u>

Variablen `antalBokse` er en klassevariabel, fordi den er erklæret med nøgleordet **static**. Dette betyder, at variabelen er tilknyttet klassen og at alle Boks-objekter deler den samme variabel. Der vil eksistere én og kun én `antalBokse`-variabel, uanset om der oprettes 0, 1, 2 eller 100 Boks-objekter.

Ligeledes er metoden `læsAntalBokse()` en klassemetode, da den er erklæret med nøgleordet **static**. Den arbejder på klasseniveau (uafhængigt af om der er skabt nogen objekter) og kan derfor ikke anvende metoder og variabler i klassen, der eksisterer på objektniveau. Det er f.eks. forbudt at bruge variablerne `længde`, `bredde` eller `højde` inde fra `læsAntalBokse()`.

Inde fra objektet kan klassevariabler (og -metoder) bruges ligesom almindelige variabler og -metoder. Det ses f.eks. i konstruktøren:

```
    antalBokse = antalBokse + 1;
```

Og for fuldstændighedens skyld: Variablerne `længde`, `bredde` og `højde` er "normale" objektvariabler, så hvert Boks-objekt har tilknyttet én af hver. Variablen `vol` er en lokal variabel, fordi den er erklæret lokalt i `volumen`-metoden og altså kun eksisterer, når `volumen`-metoden udføres. Ligeledes med `lgd`, `b` og `h`: De eksisterer kun i Boks' konstruktør.

Vi kan afprøve Boks4 med:

```
public class BenytBoks4
{
    public static void main(String[] arg)
    {
        System.out.println("Antal bokse: " + Boks4.læsAntalBokse());

        Boks4 boksen;
        boksen = new Boks4(2,5,10);

        System.out.println("Antal bokse: " + Boks4.læsAntalBokse());

        Boks4 enAndenBoks, enTredjeBoks;
        enAndenBoks = new Boks4(5,5,10);
        enTredjeBoks = new Boks4(7,5,10);

        System.out.println("Antal bokse: " + Boks4.læsAntalBokse());
    }
}
```

---

```
Antal bokse: 0
Antal bokse: 1
Antal bokse: 3
```

Det ses, at vi udefra kan kalde klassemetoder ved bare at angive klassenavnet og metodenavnet som i `Boks4.læsAntalBokse()`. Havde `antalBokse` været tilgængelig udefra (f.eks. erklæret `public` i stedet for `private`), kunne vi få fat i den udefra med: `Boks4.antalBokse`.

Vær opmærksom på, at det normalt frarådes at definere mange klassevariabler og -metoder i ens program, da det kan gøre programmet svært at gennemskue og øger de enkelte klassers afhængighed af hinanden (høj kobling). Endelig kan det friste folk til at springe "alt det besværlige med objekter" over og dermed ikke få lært eller udnyttet mulighederne i objektorienteret programmering.

## 7.1.1 Eksempler på klassevariabler

Du har, måske uden at vide det, allerede brugt en del klassevariabler og -metoder.

Der er mange klassevariabler i Javas standardbibliotek. Af de oftest brugte kan nævnes

- Matematiske konstanter som `Math.PI` (værdien af  $\pi$ ) er klassevariabler i `Math`-klassen.
- Foruddefinerede farver som `Color.BLACK` – et `Color`-objekt, der repræsenterer farven sort. Objektet ligger som en klassevariabel i (selvsamme) `Color`-klasse (pakken `java.awt`).
- `System.out` – `System.out` er et `PrintStream`-objekt, der bl.a. har metoderne `print()` og `println()`. Objektet er en klassevariabel i `System`-klassen.

Klassevariabler er nyttige til variabler, der skal være tilgængelige overalt i programmet. Det er det nærmeste, man kommer globale variabler, som det kendes fra andre programmeringsprog.

## 7.1.2 Eksempler på klassemetoder

Af nyttige klassemetoder i standardbiblioteket kan nævnes

- Matematiske funktioner som `Math.random()`, `Math.sin(double x)`, `Math.cos(double x)`, `Math.sqrt(double x)`, `Math.abs(double x)`, `Math.exp(double x)`, `Math.log(double x)`, ...
- `Double.parseDouble(String s)` returnerer værdien af `s` som et kommatal. Nyttig til at fortolke brugerindtastede tal. F.eks. giver `Double.parseDouble("3.553")` tallet 3.553.
- Tilsvarende giver `Integer.parseInt(String s)` værdien af `s` som et heltal.

- `String.valueOf(double d)` gør det modsatte af `Double.parseDouble`, den returnerer en streng, som repræsenterer et flydende kommatall. `String.valueOf(3.21)` giver altså strengen "3.21". Findes også med `int`, `byte`, `char` etc. som parameter.
- `Character.isDigit(character t)` returnerer `true` eller `false`, afhængigt af om tegnet *t* er et ciffer. Ligeledes findes `Character.isLetter(character t)`, `Character.isLetterOrDigit(character t)`, `Character.isLowerCase(character t)`, `Character.isUpperCase(character t)` og `Character.isWhitespace(character t)`. Den sidste undersøger, om *t* er et usynligt skilletegn, f.eks. mellemrum, linjeskift, tabulator.
- `System.exit()` – stopper programudførelsen og afslutter Java.

Det vigtigste eksempel på en klassemetode er `main()`-metoden, som du selv erklærer, når du skriver et program, f.eks. `BenytBoks.main()`. Når et program startes, er det altid `main()`, der kaldes. På dette tidspunkt eksisterer der endnu ingen objekter og derfor skal `main()` være en klassemetode. Der skal jo aldrig oprettes nogen `BenytBoks`-objekter!

Klassemetoder er nyttige til "subrutiner", der skal regne et eller andet ud (ud fra parametrene) og returnere resultatet. Et eksempel er vist i afsnit 2.12.4, Klassemetoder.

## 7.2 Lokale variabler og parametre

Når en metode kaldes, opretter systemet en "omgivelse" for det metodekald. I denne omgivelse oprettes parametervariablerne og de lokale variabler.

---

**En lokal variabel er kendt fra dens erklæring og ned til slutningen af den blok, der omslutter den**

**Dette kaldes variabelens virkefelt**

---

Den lidt indviklede formulering skyldes, at man kan lave variabler, der er lokale for en hvilken som helst blok – ikke kun en metode-krop. Man kan altså skrive noget som:

```
...
int a = 10;
while (a > 0)
{
    double b; // b erklæres lokalt i while-blokken
    b = Math.random();
    ...
    System.out.println(b);
    a--;
}
System.out.println(a);
System.out.println(b); // fejl: b eksisterer ikke, da vi er uden for blokken.
...
```

Vi har desuden allerede set, at man i for-løkker kan erklære en variabel, der er lokal for løkkens krop:

```
for (int i=0; i<10; i++)
    System.out.print(i);

System.out.print(i); // fejl: i eksisterer ikke uden for løkken.
```

### 7.2.1 Parametervariabler

Parametervariabler får tildelt en *kopi* af den værdi, de blev kaldt med og opfører sig i øvrigt fuldstændigt som lokale variabler. Man kan f.eks. godt tildele dem nye værdier:

```
// metode, der udskriver et bestemt antal stjerner på skærmen.
public void udskrivStjerner(int antal)
{
    while (antal>0)
    {
        System.out.print("*");
        antal = antal-1; // Det kan man godt (men det er dårlig stil)
    }
    System.out.println();

    ...
    int stj = 10;
    udskrivStjerner(stj); // kald af udskrivStjerner
    // stj er stadig 10.
}
```

Kalderen mærker intet til, at parametervariablen `antal` har fået en ny værdi, fordi kalderens værdi netop blev *kopieret* over i parametervariablen.

Her skal man være opmærksom på forskellen mellem variabler af simple typer og variabler af objekt-typer. Da det sidste er *referencer* (adresser på objekter), peger parametervariablen på samme objekt som kalderen, når den bliver kopieret. Ændrer man i objektet, slår ændringen også igennem hos kalderen.

Derfor kan `flyt1()` herunder godt ændre `x` og `y` i kalderens punkt-objekt:

```
import java.awt.*;
public class Parametervariabler
{
    public static void flyt1(Point p, int dx, int dy)
    {
        p.x = p.x+dx; // OK, vi ændrer på kalderens objekt
        p.y = p.y+dy;
    }

    public static void flyt2(Point p, int dx, int dy)
    {
        // hmm... vi smider kalderens objekt væk... men det opdager han ikke!
        p = new Point(p.x+dx, p.y+dy);
    }

    public static void main(String[] arg)
    {
        Point p1 = new Point(10,10);
        flyt1(p1,13,14);
        System.out.println("Nu er p1="+p1);

        Point p2 = new Point(10,10);
        flyt2(p2,13,14);
        System.out.println("Nu er p2="+p2);
    }
}

Nu er p1=java.awt.Point[x=23,y=24]
Nu er p2=java.awt.Point[x=10,y=10]
```

Vi kan ikke ændre på kalderens reference, som vi forsøger på i `flyt2()`: En lokal variabel oprettes, når man går ind i blokken, hvor den er defineret og nedlægges igen, når blokken forlades. Der bliver oprettet en ny variabel, hver gang programudførelsen går ind i blokken.

---

**Når en metode kaldes, bliver der oprettet et nyt sæt lokale variabler (incl. parametervariabler) uafhængigt af, hvilke metoder, der i øvrigt kaldes eller er i gang med at blive kaldt**

---

Hvis en metode bliver kaldt to gange, eksisterer der altså to versioner af den lokale variabel – én i hver deres omgivelse. Det behøver man som regel ikke at tænke på, men det er rart at have vished for, at en anden metode ikke bare kan ændre ens lokale variabler.

*Rekursion* er en teknik, der udnytter, at der bliver oprettet en ny omgivelse med nye lokale variabler, hver gang en metode kaldes. Nogle problemer kan løses meget elegant med rekursion. I Avanceret-afsnittet i slutningen af kapitlet vises nogle eksempler på rekursion.

## 7.3 Test dig selv

- 1) Hvad er en lokal variabel?
- 2) Hvornår oprettes den og hvornår nedlægges den? Hvor mange oprettes der?
- 3) Hvad er en objektvariabel?
- 4) Hvornår oprettes den og hvornår nedlægges den? Hvor mange oprettes der?
- 5) Hvad er en klassevariabel?
- 6) Hvornår oprettes den og hvornår nedlægges den? Hvor mange oprettes der?
- 7) Hvad er en klassemetode? Hvilke andre variabler og metoder kan den anvende?

## 7.4 Resumé

Variabler erklæret **static** kaldes klassevariabler (eller statiske variabler).

Metoder erklæret **static** kaldes klassemetoder (eller statiske metoder).

Vi har brug for klassemetoder og klassevariabler i 2 tilfælde:

- Hvis vi ønsker at have fælles variabler for samtlige objekter af en klasse.
- Hvis vi ønsker at kunne anvende variabler eller metoder uden at lave nye objekter.

### 7.4.1 Tilknytning

Objektvariabler og -metoder er altid tilknyttet et konkret objekt, oprettet med f.eks.

```
boksen = new Boks();
```

En klassevariabel eksisterer kun ét sted i hukommelsen, uanset hvor mange objekter der oprettes. Alle objekterne af klassen deler den samme værdi.

Klassemetoder og -variabler er ikke tilknyttet noget konkret objekt og kan altid anvendes.

### 7.4.2 Adgang til variabler og metoder

Objektvariabler anvendes uden for objektet ved at skrive objektnavn.variabel.

Objektmetoder anvendes uden for klassen ved at skrive objektnavn.metode(), f.eks.

```
boksen.volumen();
```

Klassevariabler anvendes uden for klassen ved at skrive Klassenavn.variabel.

Klassemetoder anvendes uden for klassen ved at skrive Klassenavn.metode(), f.eks.

```
Boks.læsAntalBokse();
```

### 7.4.3 Adgang fra metoder

Almindelige metoder kan uden videre anvende andre metoder, variabler, klassemetoder og klassevariable.

Klassemetoder kan kun anvende klassemetoder og klassevariable. De kan ikke anvende objektvariabler eller -metoder (til disse skal der være et objekt).



## 7.5 Avanceret

Rekursion er velegnet, hvis en opgave kan deles op i mindre tilsvarende delopgaver.

### 7.5.1 Introduktion til rekursive algoritmer

Hvis en metode kalder sig selv, er der tale om rekursion. F.eks.:

```
public class Rekursion
{
    public static void main(String[] arg)
    {
        tælNed(3);
    }

    public static void tælNed(int tæller)
    {
        System.out.print(tæller+" ");
        if (tæller>0) tælNed(tæller-1); // tælNed() kalder sig selv !!
    }
}
```

3 2 1 0

Fidusen er, at parameteren `tæller` eksisterer én gang for hver gang, `tælNed()` bliver kaldt. Så når `tælNed()` vender tilbage til kalderen, som også er `tælNed()`, er `tællers` værdi bevareret som før kaldet.

Er man uvant med rekursion, kan det være svært at gennemskue, hvad der sker. Husk da, at et kald til en metode er uafhængigt af, om metoden eventuelt allerede "er i gang med" at blive kaldt. Ovenstående rekursion kunne "foldes ud" til følgende program:

```
public class RekursionUdfoldet
{
    public static void main(String[] arg)
    {
        tælNed(3);
    }

    public static void tælNed(int tæller)
    {
        System.out.print(tæller+" ");
        if (tæller>0) tælNeda(tæller-1); // kald tælNeda(2)
    }

    public static void tælNeda(int tæller)
    {
        System.out.print(tæller+" ");
        if (tæller>0) tælNedB(tæller-1); // kald tælNedB(1)
    }

    public static void tælNedB(int tæller)
    {
        System.out.print(tæller+" ");
        if (tæller>0) tælNedC(tæller-1); // kald tælNedC(0)
    }

    public static void tælNedC(int tæller)
    {
        System.out.print(tæller+" ");
        if (tæller>0) tælNedC(tæller-1); // kalder ikke videre, da tæller=0
    }
}
```

3 2 1 0

Det er klart, at rekursion nemt kan føre til uendelige løkker, hvis man ikke passer på.

Man kan groft sagt være sikker på, at der ikke opstår en uendelig løkke, hvis man kan påvise, at alle videre kald sker med en *mindre* opgave, end den man selv blev kaldt med og at en tilstrækkelig lille opgave bliver udført uden et videre kald.

Herover bliver opgaven mindre og mindre, fordi `tæller` har en lavere værdi i hvert kald og fordi `tælNed()` kun kalder sig selv, hvis `tæller` er større end 0.

## 7.5.2 Rekursion: Beregning af formel

Det følgende program kan analysere en streng og udregne et matematisk udtryk med de fire regnearter, sinus-funktionen og et vilkårligt antal parenteser.

```
public class Formelberegning
{
    /**
     * Finder første position af en operator, f.eks +, -, * eller /.
     * Går uden om de operatoren, der er inde i en parentes.
     * Simplet løsning, der ikke tager højde for parenteser: udtryk.indexOf(tegn)
     */
    public static int findUdenforParenteser(char tegn, String udtryk)
    {
        int par = 0;
        for (int i = 0; i < udtryk.length(); i++)
        {
            char t = udtryk.charAt(i);
            if (t == tegn && par == 0) return i; // tegn fundet udenfor parenteser!
            else if (t == '(') par++;         // vi går ind i en parentes
            else if (t == ')') par--;         // vi går ud af en parentes
        }
        return -1; // tegn ikke fundet udenfor parenteser
    }

    public static double beregn(String udtryk)
    {
        udtryk = udtryk.trim();                // fjern overflødige blanktegn
        for (int opNr = 0; opNr < 4; opNr++)    // løb gennem de fire regnearter
        {
            char op = "+-*/".charAt(opNr);     // op er nu '+', '-', '*' eller '/'
            int pos = findUdenforParenteser(op, udtryk);
            if (pos > 0)                         // findes op i udtrykket?
            {
                String vs = udtryk.substring(0, pos); // ja, find venstresiden
                String hs = udtryk.substring(pos+1); // find højresiden

                double vsr = beregn(vs);          // beregn højresidens værdi
                System.out.println("beregn("+vs+") = "+vsr);

                double hsr = beregn(hs);          // beregn venstresidens værdi
                System.out.println("beregn("+hs+") = "+hsr);

                if (op == '+') return vsr + hsr;   // beregn resultat og returnér
                if (op == '-') return vsr - hsr;
                if (op == '*') return vsr * hsr;
                return vsr / hsr;
            }
        }
        // Hvis vi kommer herved kunne der ikke dele op i flere operatorer
        if (udtryk.startsWith("(") && udtryk.endsWith(")")) // parenteser omkring?
        {
            udtryk = udtryk.substring(1, udtryk.length()-1); // fjern dem
            return beregn(udtryk); // beregn indmad
        }
        if (udtryk.startsWith("sin(") && udtryk.endsWith(")")) // sinus-funktion
        {
            udtryk = udtryk.substring(4, udtryk.length()-1); // fjern 'sin(' og ')'
            double resultat = beregn(udtryk); // beregn parameteren
            System.out.println("beregn("+udtryk+") = "+resultat);
            return Math.sin(resultat);
        }
        // intet andet fundet - så må det være et tal!
        return Double.parseDouble(udtryk);
    }

    public static void main(String[] arg)
    {
        String formel = "(1+2)*3 - sin(4/5*(6-7))";
        double værdi = beregn(formel);
        System.out.println("Formlen "+formel+" er beregnet til "+værdi);
    }
}
```

```

bereg(1) = 1.0
bereg(2) = 2.0
bereg((1+2)) = 3.0
bereg(3) = 3.0
bereg((1+2)*3) = 9.0
bereg(4) = 4.0
bereg(5) = 5.0
bereg(4/5) = 0.8
bereg(6) = 6.0
bereg(7) = 7.0
bereg((6-7)) = -1.0
bereg(4/5*(6-7)) = -0.8
bereg(sin(4/5*(6-7))) = -0.7173560908995228
Formlen (1+2)*3 - sin(4/5*(6-7)) er beregnet til 9.717356090899523

```

Metoden `bereg()` deler strengen op i mindre bidder, som den udregner værdien af ved at kalde sig selv. For eksempel deles `"(1+2)*3 - sin(4*5/(6-7))"` op i delene `"(1+2)*3"` og `"sin(4*5/(6-7))"`, der hver i sær udregnes ved at kalde `bereg()`. Bliver `bereg()` kaldt med en streng, der ikke kan opdeles yderligere, antages det, at strengen indeholder et tal, som bliver fundet med et kald til `Double.parseDouble()`.

Rækkefølgen af kaldene i programmet er:

- `bereg("(1+2)*3 - sin(4*5/(6-7))")`, der kalder
  - `bereg("(1+2)*3")`, der kalder
    - `bereg("(1+2)")`, der kalder
      - `bereg("1+2")`, der kalder
        - `bereg("1")`, der giver 1
        - `bereg("2")`, der giver 2
        - returværdierne 1 og 2 lægges sammen og giver 3
      - `bereg("3")`, der giver 3
    - returværdierne 3 og 3 multipliceres og giver 9
  - `bereg("sin(4/5*(6-7))")`, der kalder
    - `bereg("4/5*(6-7)")`, der kalder
      - `bereg("4/5")`, der kalder
        - `bereg("4")`, der giver 4
        - `bereg("5")`, der giver 5
        - returværdierne 4 og 5 ganges sammen og giver 0.8
      - `bereg("(6-7)")`, der kalder
        - `bereg("6-7")`, der kalder
          - `bereg("6")`, der giver 6
          - `bereg("7")`, der giver 7
          - returværdierne 4 og 5 trækkes fra hinanden og giver -1
        - returværdierne 0.8 og -1 divideres med hinanden og giver -0.8
      - sinus til returværdien -0.8 beregnes og giver -0.717
    - returværdierne 9 og -0.717 trækkes fra hinanden og giver 9.717

Indrykningerne illustrerer dybden af rekursionen (hvor mange gange `bereg()` er i gang med at kalde sig selv).

### 7.5.3 Rekursion: Listning af filer

I afsnit 15.9.2 ses et andet eksempel på rekursion, hvor filer i den aktuelle mappe og alle undermapper listes rekursivt.

## 7.5.4 Rekursion: Tegning af fraktaler

Dette eksempel tegner et fraktalt træ. Det forudsætter kapitel 9 Grafiske programmer.

En fraktal er en struktur, hvor man, hvis man går tæt på en del af strukturen, opdager, at delen har lige så mange detaljer som helheden.

Her er rekursion velegnet, da man så blot kan lave en metode tegnGren(), der tegner en gren i et bestemt størrelsesforhold ved at tegne "stammen" i grenen og derefter tegne mindre grene (med kald til tegnGren()) med mindre størrelsesforhold).

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

public class Fraktaltrae extends JPanel
{
    /**
     * Tegner et fraktalt træ. Hver gren er i sig selv et træ.
     * @param x    x-koordinaten hvor træets rod skal tegnes
     * @param y    y-koordinaten hvor træets rod skal tegnes
     * @param dx    x-forskydning fra rod til træets første forgrening
     * @param dy    y-forskydning fra rod til træets første forgrening
     * @param str    træets størrelse
     * @param g    Graphics-objektet
     */
    public void tegnGren(Graphics g, int x, int y, int dx, int dy, int str)
    {
        if (str < 1) return; // vi vil ikke tegne forsvindende små grene

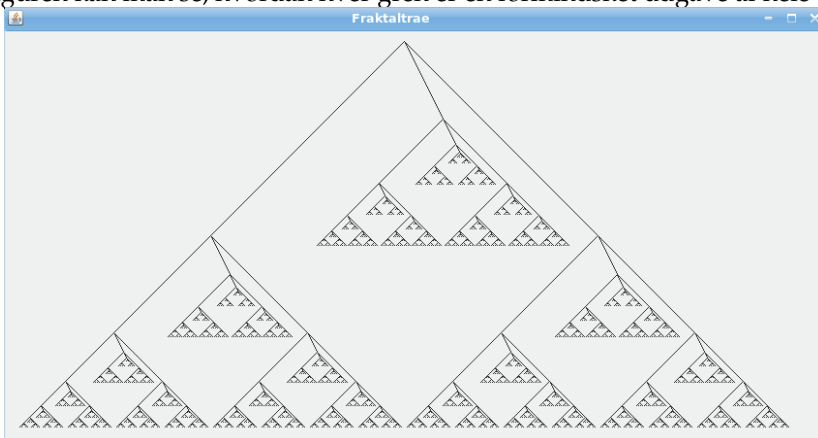
        g.drawLine(x, y, x+dx, y+dy);           // tegn stammen

        tegnGren(g, x+dx, y+dy, -str/2, str/2, str/2); // tegn gren til venstre
        tegnGren(g, x+dx, y+dy, str/10, str/5, str/3); // lille gren lidt til højre
        tegnGren(g, x+dx, y+dy, str/2, str/2, str/2); // tegn gren til højre
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        tegnGren(g, 410, 30, 0, 0, 400);
    }

    public static void main(String[] arg)
    {
        JFrame vindue = new JFrame("Fraktaltrae");
        vindue.add( new Fraktaltrae() );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        vindue.setSize(850, 450);
        vindue.setVisible(true);
    }
}
```

På figuren kan man se, hvordan hver gren er en formindsket udgave af hele træet.



# 8 Arrays

Indhold:

- Erklæring og brug af arrays.
- Arrays med startværdier.
- Arrays af objekter.
- Arrays sammenlignet med lister.

Kapitlet forudsættes ikke i resten af bogen, men er nyttigt i praktisk programmering.

Forudsætter kapitel 3, Objekter (og et enkelt sted kapitel 5, Nedarvning).

Ofte har man behov for at håndtere et større antal objekter eller simple typer på en ensartet måde. Hidtil har vi gjort det med `ArrayList`, men Java understøtter også *arrays*.

---

### **Et array er en række data af samme type**

---

Man kan f.eks. have et array af `int` eller et array af `Point`. Når man har et array af `int`, betyder det, at man har en række `int`-variabler, som ligger i arrayet og kan ændres eller læses vha. arrayet og et indeks. Indekset er nummeret på variablen i arrayet – ligesom i `ArrayList`.

Ligesom med `ArrayList` skal man skelne mellem array-variablen og array-objektet. Array-variablen refererer til array-objektet, som indeholder variablerne.

## **8.1 Erklæring og brug**

Man erklærer en array-variabel med den type data, man ønsker at lave et array af, umiddelbart efterfulgt af `[ og ]`, f.eks.:

```
int[] arr;
```

Dette erklærer, at `arr` er en variabel med typen "array af `int`". Ligesom med variabler af objekt-type er dette blot en reference hen til det egentlige array-objekt. Hvis man ønsker at oprette et array, skriver man f.eks.:

```
arr = new int[6];
```

Dette sætter `arr` til at referere til et array, der har 6 elementer.

Elementer i et array bliver altid initialiseret med 0 som standardværdi<sup>1</sup>. Arrayets værdier kan sættes og aflæses ved at angive indeks i firkantede `[]`-parenteser efter variabelnavnet:

```
public class ArrayEksempel
{
    public static void main(String[] arg)
    {
        int[] arr = new int[6];
        arr[0] = 28;
        arr[2] = 13;

        arr[3] = arr[0] + arr[1] + arr[2];

        int længde = arr.length;    // = 6, da vi oprettede det med new int[6]

        for (int i=0; i<længde; i=i+1) System.out.print( arr[i] + " " );
        System.out.println();
    }
}
```

---

```
28 0 13 41 0 0
```

Indekseringen starter altid fra 0 af og sidste lovlige indeks er lig med arrayets længde-1. Indekserer man uden for arrayets grænser, kastes en `ArrayIndexOutOfBoundsException`.

Alle arrays er objekter (derfor bruges `new`-operatoren, når vi opretter et nyt array). Alle array-objekter har variablen `length`, som fortæller, hvor mange pladser arrayet indeholder.

---

### **Længden på et array kan ikke ændres**

---

Selvom array-objekter ikke kan ændre længde, kan man lade variablen referere til et andet array-objekt med en anden længde:

```
arr = new int[8];
```

Nu refererer `arr` til et andet array med længde 8.

---

<sup>1</sup> For et array af `boolean` vil de have værdien `false` (j.v.f. afsnit 2.11.2, De simple typer). Havde arrayet indeholdt (referencer til) objekter, var de blevet sat til `null`.

## 8.1.1 Eksempel: Statistik

Arrays er gode til at lave statistik med. Her laver vi statistik på slag med to terninger:

```
public class TerningStatistik
{
    public static void main(String[] arg)
    {
        int[] statistik = new int[13];    // array med element nr. 0 til og med 12

        for (int i=0; i<100; i=i+1)
        {
            int sum = (int) (6*Math.random()+1) + (int) (6*Math.random()+1);

            statistik[sum]=statistik[sum]+1; // optæl statistikken for summen af øjne
        }

        for (int n=2; n<=12; n=n+1) System.out.println( n + ": " + statistik[n]);
    }
}
```

---

```
2: 2
3: 9
4: 7
5: 17
6: 14
7: 15
8: 8
9: 9
10: 8
11: 7
12: 4
```

## 8.1.2 Initialisere et array med startværdier

Arrays kan initialiseres med startværdier i {} og er adskilt med komma. Eksempel:

```
int[] arr = {28, 0, 13, 41, 0, 0};
```

Det er ofte meget mere bekvemt end at sætte de enkelte værdier.

Herunder et program, der udskriver antallet af dage i hver måned:

```
public class Maaneder
{
    public static void main(String[] arg)
    {
        int[] månedesLgd = {31,28,31,30,31,30,31,31,30,31,30,31};

        System.out.println("Januar har " + månedesLgd[0] + " dage.");
        System.out.println("April har " + månedesLgd[3] + " dage.");

        // med foreach-løkke, se afsnit 3.5.1.
        for (int lgd : månedesLgd) System.out.print(lgd + " ");

        System.out.println();
    }
}
```

---

```
Januar har 31 dage.
April har 30 dage.
31 28 31 30 31 30 31 31 30 31 30 31
```

Bemærk at den specielle for-each-løkke nævnt i afsnit 3.5.1 også kan bruges på arrays.

## 8.1.3 Arrayet i main()-metoden

Metoden main(), som vi har defineret utallige gange, har en parameter, som er et array af strenge. Dette array indeholder kommandolinje-argumenter ved kørsel af programmet.

```
public class Kommandolinje
{
    public static void main(String[] arg)
    {
        System.out.println("Antallet af argumenter er: " + arg.length);

        for (int i=0; i< arg.length; i=i+1)
            System.out.println("Argument "+i+" er: " + arg[i]);
    }
}
Antallet af argumenter er: 3
Argument 0 er: x
Argument 1 er: y
Argument 2 er: z
```

Programmet herover er kørt fra kommandolinjen med "java Kommandolinje x y z".

## 8.2 Gennemløb og manipulering af array

Et array er faktisk et objekt, men det har ingen metoder og kun én variabel, nemlig length. Arrays kan ikke ændre størrelse og length er således konstant.

Den eneste måde at få et array af en anden størrelse er at oprette et andet array af den ønskede størrelse og så kopiere det gamle indhold over i det nye array, så arrays er ikke særlig rare hvis antallet af elementer varierer.

Herunder ses, hvordan man kan fjerne et element fra et array:

```
public class FjernEtElement
{
    public static void main(String[] arg)
    {
        // Oprettelse og initialisering af array
        int[] a = new int[10];
        for (int n=0; n<a.length; n=n+1) a[n]=n*10;

        // Gennemløb og udskrivning af array
        System.out.print("a før: ");
        for (int n=0; n<a.length; n=n+1) System.out.print(a[n]+" ");
        System.out.println();

        // Kopiering af array / udtagning af element
        int fjernes=5;           // Element nr 5 skal fjernes.

        int[] tmp=new int[9]; // Nyt array med 9 pladser

        // bemærk at elementet der skal fjernes ikke kopieres
        for (int n=0; n<fjernes; n=n+1) tmp[n]=a[n];

        for (int n=fjernes+1; n<a.length; n=n+1) tmp[n-1]=a[n];

        a=tmp;                  // Nu refererer a til det nye array med 9 elementer

        System.out.print("a efter: ");
        for (int n=0; n<a.length; n=n+1) System.out.print(a[n]+" ");
        System.out.println();
    }
}
a før: 0 10 20 30 40 50 60 70 80 90
a efter: 0 10 20 30 40 60 70 80 90
```

Eksemplet ovenfor illustrerer de basale måder at manipulere arrays på, men bemærk, at ligesom der findes en lang række standardmetoder til at arbejde med lister (beskrevet i afsnit 3.10.2), findes der metoder til at arbejde med arrays, der gør livet meget nemmere.

Se javadokumentationen for klassen Arrays (i pakken java.util) for mere information.



## 8.3 Array af objekter

Et array af objekter oprettes på samme måde som et array af simple typer:

```
Point[] pkt = new Point[10];
```

Bemærk: Arrayet indeholder en række af *referencer* til objekter. Herover oprettes altså ingen konkrete Point-objekter! Dvs. pkt[0], pkt[1],...,pkt[9] er alle null.

Arrays kan bruges til at gå fra tal til værdier i et domæne. For eksempel konvertering af måneders numre (1-12) til deres navne:

```
public class MaanedersNavne
{
    public static void main(String[] arg)
    {
        String[] måneder = {"januar", "februar", "marts", "april", "maj", "juni",
                             "juli", "august", "september", "oktober", "november", "december" };

        System.out.println("Den 1. måned er " + måneder[0] );
        System.out.println("Den 6. måned er " + måneder[5] );
        System.out.println("Den 10. måned er " + måneder[9] );
    }
}
```

---

```
Den 1. måned er januar
Den 6. måned er juni
Den 10. måned er oktober
```

På samme måde som strenge kan andre slags objekter lægges i et array, f.eks. punkter:

```
Point[] pkt = { new Point(100,100), new Point(110,90), new Point(10,10) };
```

### 8.3.1 Polymorfi

Ligesom med almindelige variabler kan elementer i et array godt referere til nedarvinger.

```
// Bruger Terning.java og FalskTerning2.java fra kapitel 4 og 5
public class Terninger
{
    public static void main(String[] arg)
    {
        Terning[] t = { new Terning(), new FalskTerning2(), new FalskTerning2() };

        for (int i=0; i<t.length; i++) t[i].kast();    // normal for-løkke

        for (Terning ti : t) System.out.println( ti ); // foreach-løkke
    }
}
```

## 8.4 Arrays versus lister (ArrayList)

Da det er umuligt at ændre et arrays størrelse, er det besværligt at indsætte og slette elementer. Til gengæld kan arrays nemmere indeholde simple typer og man kan initialisere et array på én linje. Faktisk er ArrayList-klassen et array i en indpakning, der gør det nemmere at bruge. Hvad du vælger er op til dig selv.

En liste er god at bruge, når:

- antallet af elementer kan ændre sig
- der er brug for at indsætte og slette elementer løbende.

Et array er godt at bruge, når:

- antallet af elementer er fast og man evt. kender værdierne på forhånd
- man arbejder med simple typer som int og double
- programmet skal være meget hurtigt.

Valget er dog ikke så afgørende, da man nemt kan konvertere mellem et array af objekter og en liste (se afsnit 8.7.2).

## 8.5 Resumé

Herunder er et eksempel, der ridser brugen af arrays op.

```
public class ArrayEksempler
{
    public static void main(String[] arg)
    {
        int i0;
        int i1;
        int i2;

        // variabler skal initialiseres
        i0=3;
        i1=0;
        i2=10;

        //type    variabelnavn    nyt array med 3 pladser (alle med værdi 0)
        int[]    t                =    new int[3];

        // array-elementer er variabler der skal initialiseres.
        t[0]=3;    // Den første plads er plads 0
        t[1]=0;    // ...
        t[2]=10;   // Den sidste plads er plads 2

        // t.length er 3, fordi der er 3 pladser i arrayet.
        for (int n=0; n<t.length; n=n+1) System.out.println(t[n]);

        // initialisering af array vha. for-løkke.
        for (int n=0; n<t.length; n=n+1) t[n]=n*10;

        // Andre måder at initialisere arrays på

        //type    variabelnavn    nyt array med 3 pladser

        // konstante værdier:
        int[]    tc                =    { 3 , 0 , 10 };
        // udtryk:
        int[]    tc2               =    { i0 , i1+2 , i2*10 };
        // flere udtryk:
        int[]    tc3               =    { tc[0] , tc[2]+2 , tc2[1]*10 };

        // Tildeling af array-variabler:

        //type    variabelnavn
        int[]    a;                // erklæring af array-variabel.

        a=t;                    // tildeling. Nu refererer a og
                                // t til samme array.

        a[1]=100;                // Dette ændrer både a[1] og t[1]

        a=tc2;                    // tildeling. Nu refererer a og
                                // tc2 til samme array.
    }
}
3
0
10
```

## 8.6 Opgaver

- 1) Lav et program, der simulerer kast med 6 terninger. Der udføres f.eks. 100 kast. Optæl i et array hyppigheden af summen af øjenantallene.
- 2) Udvid programmet til at kunne lave statistik på kast med et vilkårligt antal terninger.
- 3) Ændr programmet, så man kan angive antallet af terninger på kommandolinjen.

## 8.7 Avanceret

Når et array erklæres, kan man selv bestemme, om []'erne skal være før eller efter typen.

```
int[] arr = new int[6];
```

og

```
int arr[] = new int[6];
```

er altså det samme.

Vi anbefaler den første form, fordi typen af variabelen faktisk er `int[]`. Den sidste form bliver forvirrende, hvis man erklærer flere variabler på samme linje, for da skal man huske [] efter hver variabel:

```
int arr[], arr2[];
```

### 8.7.1 Flerdimensionale arrays

Man kan også lave (for eksempel) et todimensionalt array:

```
int[][] toDim = new int[256][16];
```

Her sker der følgende:

- En variabel af typen `int[][]` (reference til array af array af `int`) oprettes.
- Der oprettes et array med 256 elementer – hvert element er af typen `int[]`.
- Der oprettes 256 arrays, hvert indeholder 16 `int`.
- De 256 arrays af `int` bliver initialiseret med 0'er.

Eksempel: Den lille og store tabel

```
public class Tabel
{
    public static void main(String[] arg)
    {
        int[][] tabel = new int[20][10];

        for (int i = 0; i < 20; i=i+1)
            for (int j = 0; j < 10; j=j+1)
                tabel[i][j] = i*j;

        System.out.println("4*5 = " + tabel[4][5]);
        System.out.println("9*8 = " + tabel[9][8]);
        System.out.println("2*6 = " + tabel[2][6]);
    }
}
```

Bemærk, at `tabel.length` er 20 (der er 20 arrays af `int`), mens `tabel[0].length` er 10 og ligeledes `tabel[1].length`, `tabel[2].length`, ... `tabel[19].length` (der er 10 `int`-værdier i hver tabel).

Det er muligt at have forskelligt antal elementer i hver tabel af `int`, f.eks. kunne vi sætte

```
tabel[3] = new int[15];
```

hvorefter længden af `tabel[3]` ville være 15, mens de andres længde stadig ville være 10.

Flerdimensionale arrays kan også initialiseres med startværdier. Her er en labyrint:

```
public class Labyrint
{
    public static void main(String[] arg)
    {
        int[][] labyrint = {
            { 0, 0, 0, 1, 1, 1, 1, 1 },
            { 0, 1, 1, 1, 0, 0, 0, 1 },
            { 0, 0, 0, 0, 0, 1, 0, 1 },
            { 1, 1, 1, 1, 0, 1, 0, 1 },
            { 0, 1, 0, 0, 0, 1, 1, 1 },
            { 0, 0, 0, 1, 1, 1, 0, 1 },
            { 1, 1, 0, 0, 0, 0, 0, 0 },
            { 1, 1, 1, 1, 1, 1, 1, 0 } };

        for (int i=0; i<8; i=i+1) {
            for (int j=0; j<8; j=j+1)
                if (labyrint[i][j] == 1) System.out.print("*");
                else System.out.print(" ");
            System.out.println();
        }
    }
}

*****
***  *
*   *
**** *
*   ***
***  *
**
*****
```

## 8.7.2 Konvertere mellem arrays og lister

Man kan nemt kan konvertere mellem et array af objekter og ArrayList :

```
import java.util.*;
public class KonvertereMellemArrayOgArrayList
{
    public static void main(String[] arg)
    {
        String[] mdrA = {"januar", "februar", "marts", "april", "maj" };

        List<String> mdrL = Arrays.asList( mdrA );           // konvertér til liste..
        ArrayList<String> mdrAL = new ArrayList<String>( mdrL ); // ...og ArrayList

        // supernem måde at udskrive ethvert array af objekter
        System.out.println("Nogle måneder: " + Arrays.asList( mdrA ) );

        // initialisering af liste v.hj.a. Arrays.asList():
        List<String> mdrL2 = Arrays.asList(
            new String[] {"januar", "februar", "marts", "april", "maj" }
        );

        // initialisering af ArrayList v.hj.a. Arrays.asList():
        ArrayList<String> mdrAL2 = new ArrayList<String>( Arrays.asList(
            new String[] {"januar", "februar", "marts", "april", "maj" }
        ));

        // konvertering tilbage igen til array
        String[] mdrA2 = mdrAL.toArray(new String[0]);
    }
}

Nogle måneder: [januar, februar, marts, april, maj]
```

Man kan få en generel liste (af typen List; et interface, der har alle de samme metoder som ArrayList) med Arrays.asList() (vil man ikke nøjes med det, må man pakke kaldet ind i en new ArrayList<String>()).

Man kan gå den anden vej og få et array ved at kalde toArray() på en liste.

# 9 Grafiske programmer

Indhold:

- Oprettelse og brug af grafiske vinduer
- At tegne grafik med Graphics-objektet
- Større opgave: Matador-spillet som et grafisk program

Kapitlet forudsættes i kapitel 10, Appletter, kapitel 11, Grafiske standardkomponenter og kapitel 12, Interfaces.

Forudsætter kapitel 3, Objekter (4, Definition af klasser og 5, Nedarvning er en fordel). Den større opgave forudsætter kapitel 5, Nedarvning.

Vi kan tegne grafik på skærmen ved at skrive en klasse, der arver fra klassen JPanel og definere metoden `paintComponent()`. Dette metode vil systemet kalde for at få grafikken tegnet på skærmen. Systemet overfører et `Graphics`-objekt (beskrevet i afsnit 9.1), som vi kan bruge til at tegne med.

I eksemplet nedenfor tegner vi en linje, en fyldt oval og noget tekst med grøn skrift.

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

public class Grafikpanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        // Herunder referer g til et Graphics-objekt man kan tegne med

        super.paintComponent(g);           // tegn først baggrunden på panelet

        g.drawLine(0,0,50,50);

        g.fillOval(5,10,300,30);

        g.setColor(Color.GREEN);

        g.drawString("Hej grafiske verden!",100,30);
    }
}
```

For at vise grafikken på skærmen skal vi definere en `main()`-metode, der opretter et `Grafikpanel`-objekt og et vindue, som viser panelet:

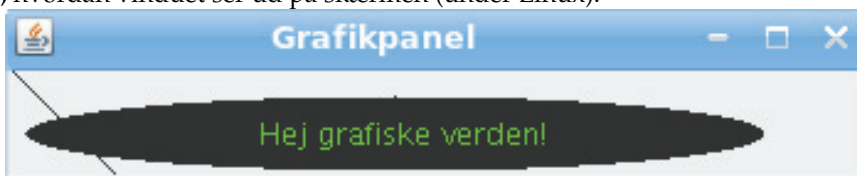
```
import javax.swing.*;

public class BenytGrafikpanel
{
    public static void main(String[] arg)
    {
        Grafikpanel panel = new Grafikpanel();           // opret panelet

        JFrame vindue = new JFrame("Grafikpanel");       // opret et vindue på skærmen
        vindue.add( panel );                             // vis panelet i vinduet

        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // reagér på luk
        vindue.setSize(350,70);                          // sæt vinduets størrelse
        vindue.setVisible(true);                          // åbn vinduet
    }
}
```

Her ses, hvordan vinduet ser ud på skærmen (under Linux):



Vinduets øverste venstre hjørne er i (0,0) og koordinaterne regnes mod højre og nedad.

## 9.1 Klassen Graphics

Graphics er beregnet til at tegne grafik (på skærm eller printer). Man skal ikke selv oprette Graphics-objekter med new, i stedet får man givet et "i hånden" af styresystemet. Herunder gengives kun nogle af metoderne – se Javadokumentationen for en komplet liste.

*java.awt.Graphics – til tegning af grafik*

### Metoder

void **drawString**(String tekst, int x, int y)  
tegner en tekst med øverste venstre hjørne i (x,y).

void **drawImage**(Image billede, int x, int y, ImageObserver observatør)  
tegner et billede med øverste venstre hjørne i (x,y); observatør bør være objektet selv (this).

void **drawLine**(int x1, int y1, int x2, int y2)  
tegner en linje mellem punkterne (x1, y1) og (x2, y2).

void **drawRect**(int x, int y, int bredde, int højde)  
tegner omridset af et rektangel.

void **drawRoundRect**(int x, int y, int bredde, int højde, int buebredde, int buehøjde)  
tegner omridset af et rektangel, der er afrundet i hjørnerne.

void **drawOval**(int x, int y, int bredde, int højde)  
tegner en oval med øverste venstre hjørne i (x,y). Er bredde==højde, tegnes en cirkel.

void **drawArc**(int x, int y, int bredde, int højde, int startvinkel, int vinkel)  
tegner en del af en oval, men kun buen fra *startvinkel* og *vinkel* grader rundt (mellem 0 og 360).

void **drawPolygon**(Polygon p)  
tegner en polygon (mangekant) ud fra et Polygon-objekt.

Tilsvarende findes **fillRect**, **fillRoundRect**, **fillOval**, **fillArc** og **fillPolygon**.

void **clearRect**(int x, int y, int bredde, int højde)  
udfylder et rektangel med baggrundsfarven.

Rectangle **getClipBounds**()

giver klippings-omridset. Kun punkter inden for dette omrids bliver faktisk tegnet, ting uden for omridset bliver beskåret til den del, der er inden for omridset.

void **translate**(int x, int y)  
forskyder koordinatsystemet, sådan at (x,y) bliver (0,0)

void **setColor**(Color nyFarve)  
sætter tegningsfarven til nyFarve. Alt bliver herefter tegnet med denne farve.

Color **getColor**()  
aflæser tegningsfarven.

void **setFont**(Font nySkrifttype)  
sætter skrifttypen til nySkrifttype. Dette påvirker tekster skrevet med drawString() herefter.

Font **getFont**()  
aflæser skrifttypen.

Har man brug for flere faciliteter til tegning af grafik end ovenstående giver, kan man gå over til at bruge Java2D (se avanceret-afsnittet i slutningen af kapitlet).

## 9.1.1 Eksempel: Grafikdemo

Her følger et eksempel, der viser mange af mulighederne, der er med Graphics-objektet.

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

public class Grafikdemo extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);          // tegn først baggrunden på panelet

        g.drawRoundRect(10,10,80,80,25,25); // tegn rektangel med runde hjørner

        g.drawArc(110,10,80,80,20,320);    // tegn buestykke

        g.fillArc(210,10,80,80,20,320);    // tegn lagkagestykke (udfyldt)

        Polygon p = new Polygon();          // lav polygon, der viser en pil:
        p.addPoint(0,13); p.addPoint(45,13); // frem
        p.addPoint(45,0); p.addPoint(60,15); p.addPoint(45,30); // spidsen
        p.addPoint(45,17); p.addPoint(0,17); // tilbage

        p.translate(300,10);                // flyt polygonen
        g.drawPolygon(p);                   // tegn polygonen

        p.translate(0,50);                  // flyt polygonen mere
        g.fillPolygon(p);                   // tegn polygonen udfyldt

        for (int i=0; i<4; i++)             // tegn forskellige skriftstørrelser
        {
            int størrelse = 10+i*4;
            Font skrifttype = new Font("Serif", Font.ITALIC, størrelse);
            g.setFont(skrifttype);
            g.drawString("Skrift "+størrelse, 400, 15+25*i);
        }

        // Indlæs billede. Forudsætter at "bog.gif" er der, hvor programmet køres.
        // Bemærk: I en applet, skriv i stedet getImage(getCodeBase(), "bog.gif")
        // Bemærk: Billedformatet skal være platformsneutralt, f.eks GIF, JPG, PNG.
        Image billede = Toolkit.getDefaultToolkit().getImage("bog.gif");

        g.drawImage(billede, 110, 100, this); // tegn billedet

        g.drawImage(billede, 0, 100, 100, 160, this); // tegn billedet skaleret
    }
}
```



Og main()-metode:

```
import javax.swing.*;

public class BenytGrafikdemo
{
    public static void main(String[] arg)
    {
        JFrame vindue = new JFrame("Grafikdemo"); // opret et vindue på skærmen
        vindue.add( new Grafikdemo() );          // vis panelet i vinduet
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // reager på luk
        vindue.setSize(500,200);                 // sæt vinduets størrelse
        vindue.setVisible(true);                 // åbn vinduet
    }
}
```



## 9.1.2 Eksempel: Kurvetegning

Her er en klasse, der tegner en farvet kurve over sinus-funktionen.

I konstruktøren bestemmer vi farverne (opretter Color-objekter) for punkterne, der skal tegnes (vi bruger heltalsdivision med %, se afsnit 2.11.4, for at få nogle gode farveværdier).

Her opretter vi også et vindue og viser panelet på skærmen (bemærk, at fordi vi gør det inden i panel-objektet, skal vi skrive `vindue.add(this)` i stedet for `vindue.add(panel)`).

Vi tegner punkterne i `paintComponent()`, der er gjort så lille og hurtig som muligt (bl.a. ved ikke at oprette objekter i denne metode) – den kaldes jo hver gang skærmen skal opdateres.

Farverne huskes i listen `farver`, der er defineret som objektvariabel (sådan at den er kendt, så længe Kurvetegning-objektet findes). På den måde får vi data overført fra konstruktør til `paintComponent()`.

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

public class Kurvetegning extends JPanel
{
    ArrayList<Color> farver; //objektvariabel kendt i konstruktør og paintComponent
    int forskydning = 50; // en forskydning i farvevalget (bruges i afsnit 9.4.1)

    public Kurvetegning() // forbered punkterne i konstruktøren
    {
        farver = new ArrayList<Color>();
        for (int i=0; i<400; i++)
        {
            Color farve = new Color(i%256, (i*2)%256, (i*4)%256);
            farver.add(farve);
        }

        JFrame vindue = new JFrame("Kurvetegning"); // opret et vindue på skærmen
        vindue.add( this ); // vis dette panel i vinduet
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // reager på luk
        vindue.setSize(400,300); // sæt vinduets størrelse
        vindue.setVisible(true); // åbn vinduet
    }

    public void paintComponent(Graphics g) // tegn punkterne
    {
        super.paintComponent(g); // tegn først baggrunden på panelet

        g.drawString("Kurvetegning", forskydning%400, forskydning%300);
        for (int x=0; x<farver.size(); x++)
        {
            int y = 140 - (int) (130*Math.sin(0.05*x));
            int i = (x+forskydning)%400;
            Color farve = farver.get(i);
            g.setColor(farve);
            g.fillRect(x, y, 5, 5);
        }
    }
}
```



Her er klassen, der opretter Kurvetegning-objektet:

```
public class BenytKurvetegning
{
    public static void main(String[] arg)
    {
        Kurvetegning kt = new Kurvetegning();
    }
}
```

## 9.2 Metoder du kan kalde

Ud over at tegne grafik har man også ofte brug for at påvirke selve vinduet eller panelet, f.eks sætte vinduets størrelse eller titel eller bede systemet om at gentegne skærmen.

*JFrame-klassens metoder – (generel) betyder, at metoden også findes i andre grafiske klasser.*

<b>repaint()</b> forårsager, at systemet kalder paintComponent() lidt senere, hvorved vinduet/panelet bliver gentegnet.	(generel)
void <b>setSize</b> (int bredde, int højde) sætter bredden og højden.	(generel)
void <b>setLocation</b> (int x, int y) sætter vinduets position på skærmen.	(kun JFrame)
void <b>setTitle</b> (String titel) sætter vinduets titel.	(kun JFrame)
void <b>setVisible</b> (boolean synlig) bestemmer, om vinduet/panelet/komponenten er synlig. Kald setVisible(true) for at åbne et vindue og setVisible(false) for at lukke det.	(generel)
void <b>setCursor</b> (Cursor museudseende) bestemmer musens udseende (muligheder er bl.a.: Cursor.DEFAULT_CURSOR, Cursor.HAND_CURSOR, Cursor.CROSSHAIR_CURSOR, Cursor.MOVE_CURSOR)	(generel)
void <b>setForeground</b> (Color forgrundsfarve) sætter forgrundsfarven, som er den farve, Graphics-objektet normalt tegner med.	(generel)
void <b>setBackground</b> (Color baggrundsfarve) sætter baggrundfarven, som er den farve, baggrunden bliver udfyldt med.	(generel)
void <b>setFont</b> (Font nySkrifttype) sætter skrifttypen (som er den skrifttype, Graphics-objektet normalt tegner med).	(generel)
Dimension <b>getSize</b> () returnerer vinduets/panelets størrelse som et Dimension-objekt med bredde og højde.	(generel)
Tilsvarende findes <b>getLocation</b> , <b>getTitle</b> , <b>getCursor</b> , <b>getForeground</b> , <b>getBackground</b> og <b>getFont</b> .	
De af ovenstående metoder, der også findes i andre grafiske objekter, er markeret med (generel)	

Bemærk at det er *systemet* og ikke dig, der kalder paintComponent().

---

**Metoden paintComponent() kaldes af systemet når vinduet skal gentegnes**

**Det er næsten altid en fejl at kalde paintComponent() fra sin egen kode**

**Ønsker man at vinduet skal gentegnes, kalder man repaint()**

---

I afsnit 9.4.4, Passiv versus aktiv visning, vil dette blive uddybet.

Bemærk også, at der er forskel på om en metode på et objekt kaldes inde fra objektet, eller udefra. Udefra kalder man metoden med en variabel, der peger på objektet, det gør man ikke indefra (man kan dog bruge `this`). Dette er forklaret i kapitel 4, Definition af klasser.

Sådan kaldes en metode *inde* fra panelet:

```
public class Grafikpanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        Dimension d = this.getSize(); // eller bare: getSize()
        System.out.println("Jeg har størrelsen: "+d);
    }
}
```

Sådan kaldes en metode *udefra*:

```
public class BenytGrafikpanel
{
    public static void main(String[] arg)
    {
        Grafikpanel panel = new Grafikpanel();

        ...

        Dimension d = panel.getSize();
        System.out.println("Panelet har størrelsen: "+d);
    }
}
```

## 9.3 Opgaver

- 1) Ændr Grafikpanel til at tegne nogle andre figurer.
- 2) Skriv noget ud når `paintComponent()` bliver kaldt (med `System.out.println()`) og se hvornår `paintComponent()` bliver kaldt (prøv f.eks. at minimere og gendanne vinduet eller dække det halvt over).
- 3) Lav et program, der viser et digitalur som tekst (vink: Brug et `Date`-objekt). Sørg for at uret opdateres hvert sekund (vink: se 9.4.1, Simple animationer).
- 4) Lav et program, der viser et analogt ur.  
Vink: Du kan benytte følgende formler til at beregne viserens længde i de to retninger:  
`x = r*Math.sin(2*Math.PI*s/60); y = r*Math.cos(2*Math.PI*s/60)`  
Lad urets størrelse afhænge af panelets størrelse (vink: Brug `getSize()`).
- 5) Ændr Grafikpanel, så den tegner et skakbræt med sorte og hvide felter.  
Et tårn og en bonde tegnes derefter på brættet.
- 6) Ændr programmet, sådan at det er nemt at ændre brikkernes koordinater.

Koordinaterne gemmes i to `Point`-objekter:

```
Point tårn = new Point(100,200);
Point bonde = new Point(200,200);
```

### 9.3.1 Opgave: Grafisk Matador-spil

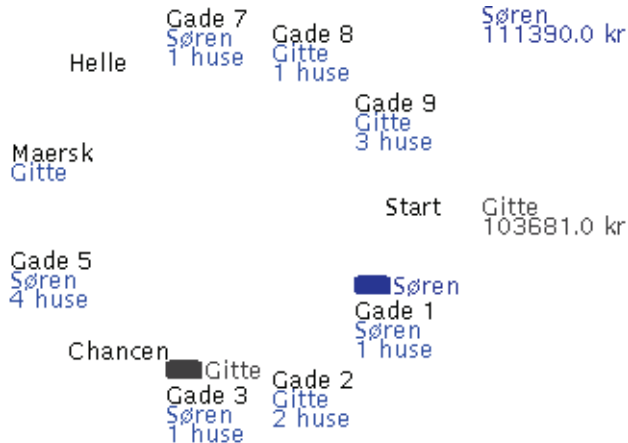
Udvid matadorspillet fra kapitel 5, Nedarvning, til at kunne vises grafisk i et vindue.

#### Vink

Når du skal programmere, så vær systematisk og del opgaven op i små delopgaver. Løs en delopgave ad gangen og afprøv om det fungerer, før du går videre.

- 1) Hent kildeteksten til matadorspillet (version 2: `Felt.java`, `Gade2.java`, `Grund2.java`, `Hel-le.java`, `Rederi2.java`, `Start.java`, `Spiller.java`, `BenytMatadorspil.java` og `Matadorspil.java` ændret til at bruge `Gade2` og `Rederi2`) og prøv det.

- 2) Genbrug Grafikpanel ovenfor. Lad initialisering ske i konstruktøren. Variabler, der skal deles mellem flere metoder, skal være objektvariabler (de lokale eksisterer jo kun i den metode, de er defineret i). Tjek om spillet stadig kan køre.
- 3) Udbyg derefter Grafikpanel til at tegne felternes navne og evt også spillerne og bilerne.
- 4) Felterne bør tegnes specielt afhængigt af deres type, sådan at f.eks. en gade også kan vise hvor mange huse, der er på den. Definér tegn(Graphics g, int x, int y) på Felt.



## Flere vink og løsningsforslag

Det er bedst, at du bruger hovedet og kun ser på vinkene, hvis du er gået i stå.

- 1) Prøve programmet.  
Har du ikke allerede prøvet matadorspillet, så prøv at køre programmet og forstå hvordan det virker. Herefter er det naturligvis meget lettere, at lave en grafisk udgave! Brug trinvis gennemgang (trace into/step over), indtil du føler, du har forstået programkoden. Først da er du klar til at prøve grafisk.

- 2) Struktur i et grafisk program.

Lav en reference i Grafikpanel til Matadorspil (evt. overført i konstruktøren):

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

public class MatadorGrafikpanel extends JPanel
{
    Matadorspil spil;
```

Husk, at vinduet først tegnes, når initialiseringen er færdig, så hvis du f.eks. kører 20 runder i konstruktøren, tager det lang tid, før systemet kalder paintComponent()!

- 3) Definér panelets paintComponent()-metode til at tegne felternes navne:

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);    // tegn først baggrunden på panelet

    for (int i = 0; i < felter.size(); i++) {
        Felt felt = felter.get(i);

        double v = Math.PI * 2 * i / felter.size(); // vinkel i radianer
        int x = 100 + (int) (100 * Math.cos(v));
        int y = 110 + (int) (100 * Math.sin(v));

        g.setColor(Color.BLACK);
        g.drawString(felt.navn, x, y);    // tegn feltets navn
```

4) Nu ændrer du sådan, at felterne har deres egen tegnemetode (der tegner navnet):

```
public class Felt
{
    String navn;

    public void tegn(Graphics g, int x, int y)
    {
        g.drawString(f.navn, x, y);
    }
}
```

... og i Grafikpanel definerer lader vi paintComponent()-metoden kalde tegn():

```
g.setColor(Color.BLACK);
felt.tegn(g, x, y);
```

Grund2 skal også tegne en ejer, så den får sin egen tegn()-metode

```
public void tegn(Graphics g, int x, int y)
{
    super.tegn(g, x, y); // kald Felts tegn() for at tegne navnet
    if (ejer != null) g.drawString(ejer.navn, x, y+15);
}
```

Ligeledes med Gade2, der også skal vise et antal huse:

```
public void tegn(Graphics g, int x, int y)
{
    super.tegn(g, x, y); // kald Grund2s tegn() for at tegne navnet og ejeren
    if (antalHuse > 0) g.drawString(antalHuse + " huse", x, y+30);
}
```

## 9.4 Avanceret

Her er nogle fif til animationer og mere avanceret grafiktegning.

### 9.4.1 Simple animationer

Som sagt forårsager repaint(), at vinduet gentegnes lidt senere (ved at systemet kalder paintComponent()). Hvis vi kalder repaint() regelmæssigt, vil vi få gentegnet vinduet regelmæssigt. Dette kan udnyttes til at lave animationer.

Følgende eksempel åbner Kurvetegning og gentegner det 25 gange i sekundet. Samtidig ændres objektvariablen forskydning, sådan at de forskellige farver "løber" hen ad kurven og ordet "Kurvetegning" bevæger sig skråt ned over skærmen.

```
public class BenytKurvetegningAnimeret
{
    public static void main(String[] arg)
    {
        Kurvetegning kt = new Kurvetegning();

        for (int t=0; t<100000; t++)
        {
            kt.forskydning = t; // ændr forskydningen så kurven ser anderledes ud
            kt.repaint(); // forårsager at paintComponent() kaldes af systemet
            try { Thread.sleep(40); } catch (Exception e) {} // Vent 0.040 sekund
        }
        System.out.println("Animation er færdig med at køre");
    }
}
```

I ovenstående eksempel udnytter vi, at main-metoden, efter at have startet grafikken, faktisk er fri til at lave andre ting; grafik-systemet kører i en separat proces (kaldet tråd) uafhængig main-metoden.

## 9.4.2 Animationer med en separat tråd

I nogle tilfælde kan det være rart, at animationen kører i en separat tråd, uafhængig af main-metoden (tråde bliver behandlet i kapitel 17). Her er et eksempel på dette:

```
public class KurveanimationMedTraad extends Kurvetegning implements Runnable
{
    public KurveanimationMedTraad()
    {
    }

    public void run()
    {
        for (int t=0; t<100000; t++)
        {
            forskydning = t;          // ændr forskydningen så kurven ser anderledes ud
            repaint();                // systemet vil kalde paintComponent() (fra grafiktråden)
            try { Thread.sleep(40); } catch (Exception e) {}
        }
    }

    public static void main(String[] arg)
    {
        KurveanimationMedTraad ka = new KurveanimationMedTraad();
        Thread tråd = new Thread(ka); // opret tråd, der kører run() på ka
        tråd.start();                 // start den nye tråd
        System.out.println("Animationen kører nu...");
    }
}
```

For kortheds skyld arver klassen fra Kurvetegning, så vi kan genbruge paintComponent() og koden, der initialiserer og åbner vinduet. Ligeledes for kortheds skyld har vi lagt main()-metoden direkte i klassen (i stedet for at have en BenytKurveanimationMedTraad-klasse med en main()-metode).

## 9.4.3 Java2D - avanceret grafiktegning

Java2D giver mulighed for mere avanceret grafikmanipulering. Blandt faciliteterne er:

- En række geometriske grundfigurer, såsom linjer, kurver, rektangler, ellipser og mekanismer til at tegne næsten enhver ønskelig geometrisk form.
- Transformationer (skalering, rotering, vridning) af figurer, tekster og billeder.
- Halvgennemsigtig tegning (hvor det, der var bagved det tegnede, stadig kan skimtes).
- Trappeudjævning (udjævnede farveovergange, eng.: antialiasing).
- Mekanismer til at afgøre, om et punkt er inden eller uden for en vilkårlig geometrisk figur, tekst eller billede (f.eks. klik med musen).

Der er en glimrende demo (med kildetekst) af Java2D i JDK'et (i jdk1.6/demo/jfc/Java2D). Dobbeltklik på jar-filen eller skriv fra kommandolinjen: java -jar Java2Demo.jar.

På <http://java.sun.com/docs/books/tutorial/2d> findes en god introduktion til Java2D.

Java2D virker ved, at paintComponent()-metoden får overført et Graphics-objekt, som i virkeligheden er et Graphics2D-objekt (Graphics2D arver fra Graphics). Det typekonverterer man til Graphics2D og har så adgang til de ekstra funktioner i Java2D:

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    ...
}
```

Man kan derefter tegne med de normale funktioner fra Graphics-klassen, men også med nogle af de mange nye metoder i Graphics2D-klassen.

De gamle tegnetoder (som drawLine(), drawRect(), ...) er erstattet med en enkelt, draw(), der kan tegne alle mulige geometriske former repræsenteret af Shape-objekter:

```
Shape s = new Line2D.Float(0, 0, 100, 100);
g2.draw(s);
```

Shape-objekter kan slås sammen med et GeneralPath-objekt (der også selv er en Shape):

```
GeneralPath figur = new GeneralPath();
figur.append( new Line2D.Float(0, 0, 100, 100), false );
figur.append( new CubicCurve2D.Float(0, 0, 80, 15, 10, 90, 100, 100), false );
figur.append( new Arc2D.Float(-30, 0, 100, 100, 60, -120, Arc2D.PIE), false );
g2.draw( figur );
```

Det samme GeneralPath-objekt kan med fordel bruges igen i hver gentegning, for hurtigere kørselstid (så opret det et andet sted end i paintComponent()).

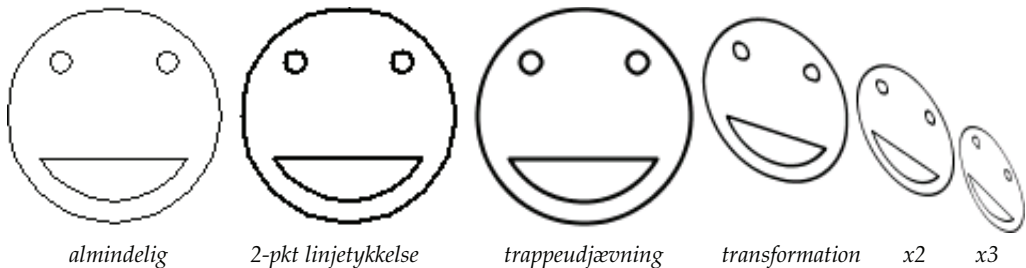
Klassen AffineTransform repræsenterer en lineær transformation af koordinatsystemet. Med den kan man lave skaleringer (større/mindre), drejninger (med/mod udret) og vridninger ("vælte" figuren).

Ved at kalde transform() på Graphics2D-objektet ændres dets koordinatsystem sådan, at alt det der efterfølgende tegnes, bliver skaleret, drejet og vredet i henhold til transformationen. Hvis man ændrer transformationen, bør man huske den gamle (fås med getTransform()) og sætte den tilbage igen (med setTransform()) før paintComponent() returnerer.

Med setStroke() sætter man bredden af linjen, der skal tegnes, om linjerne skal have kantede eller afrundede endestykker og knæk, om de skal være punkterede (og hvordan).

Med setRenderingHint() sætter man forskellige vink til tegnealgoritmen: om der skal laves trappeudjævning, om hastighed eller kvalitet skal prioriteres og en masse andet.

I eksemplet herunder opretter vi en figur og tegner den, først almindeligt, derefter med en linjetykkelse på 2 punkter, derefter med trappeudjævning og til sidst med mere og mere rotation, skalering og vridning.



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

public class Java2DDemo extends JPanel
{
    GeneralPath fig;

    AffineTransform trans;

    BasicStroke strectype = // 2-punktsstreg med kantede ender og runde knæk
        new BasicStroke(2, BasicStroke.CAP_SQUARE, BasicStroke.JOIN_ROUND);

    public Java2DDemo()
    {
        setBackground(Color.WHITE);

        // Lav 'smiley' - cirkel, to øjne og glad åben mund
        fig = new GeneralPath( new Ellipse2D.Float(0, 0, 100, 100) );
        fig.append( new Ellipse2D.Float(20, 20, 10, 10), false );
        fig.append( new Ellipse2D.Float(70, 20, 10, 10), false );
        fig.append( new Arc2D.Float(10,10, 80,80, 330,-120,Arc2D.CHORD), false );
    }
}
```

```

    trans = AffineTransform.getScaleInstance(0.7, 0.7); // formindsk
    trans.rotate(0.3); // roter
    trans.shear(0.3,0); // vrid
    trans.translate(160,-50); // flyt til siden
}

public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g; // brug Java2D

    // koordinattransformation
    AffineTransform orgTrans = g2.getTransform(); // husk original transformation
    g2.translate( 10, 10 );

    g2.draw( fig ); // tegn almindelig
    g2.translate( 110, 0 );

    g2.setStroke( stregtype );
    g2.draw( fig ); // tegn med 2-pkt linjetykkelse
    g2.translate( 110, 0 );

    g2.setRenderingHint( // sæt tegnevink til trappeudjævning (antialias)
        RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON );
    g2.draw( fig ); // tegn med trappeudjævning

    g2.transform( trans );
    g2.draw( fig ); // tegn med transformation

    g2.transform( trans );
    g2.draw( fig ); // tegn med transformation x2

    g2.transform( trans );
    g2.draw( fig ); // tegn med transformation x3

    g2.setTransform( orgTrans ); // genskab orig. transformation
}

public static void main(String[] arg)
{
    JFrame vindue = new JFrame( "Java2DDemo" );
    vindue.add( new Java2DDemo() );
    vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    vindue.setSize(500,150);
    vindue.setVisible(true);
}
}

```

## 9.4.4 Passiv versus aktiv visning

Indtil nu har vi tegnet grafik ved at definere `paintComponent()`-metoden, og så "håbet på" at systemet vil kalde vores tegnemetode. Det gør systemet:

- når nye områder af vinduet bliver synligt, f.eks. fordi andre vinduer, der dækker vores vindue, bliver flyttet
- når vinduets størrelse ændrer sig
- når vi, med et kald til `repaint()`, beder systemet om at gentegne vinduet

Dette kaldes for *passiv visning* (eng.: passive rendering), idet det er grafiksysteemets og ikke os, der bestemmer hvornår og hvilket udsnit af vinduet, der skal tegnes. Selvom vi kalder `repaint()` 3 gange lige efter hinanden, vil systemet kun kalde `paintComponent()` én gang (og måske slet ikke, hvis vinduet er minimeret eller helt overdækket af andre vinduer!).

Vil man have fuld kontrol over tegneprocessen, er man nødt til at bruge *aktiv visning*. Ved aktiv visning tager man selv initiativ til tegning af vinduet. Man definerer *ikke* `paintComponent()`.



Det følgende viser en animeret stjernehimmel, hvor vi bruger 'smiley'-figuren fra før:

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

public class AktivVisning extends JFrame
{
    Graphics2D g2;
    AffineTransform orgTrans;
    GeneralPath fig = new Java2DDemo().fig;           // stjæl 'smiley' fra andet eks.
    double[][] koord = new double[50][6];           // koordinater på figurer

    void init() {
        this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        this.setIgnoreRepaint(true);                // system skal IKKE kalde paintComponent()
        this.setVisible(true);

        g2 = (Graphics2D) this.getGraphics();        // aktiv visning: gem Graphics...
        orgTrans = g2.getTransform();                // ... og brug det som du lyster!

        for (int i=0; i<koord.length; i++)           // sæt alle stjerne-koordinater
            for (int j=0; j<koord[i].length; j++)    // til noget tilfældigt: 0=tid,
                koord[i][j] = Math.random();         // 1=x, 2=y, 3,4=skala, 5=rotation

    }

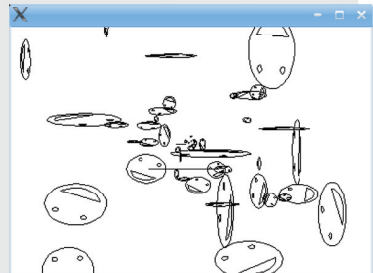
    // public void paintComponent() {}                // definer IKKE - aktiv visning!

    void tegn() {                                     // venter lidt, tegner og opdaterer koordinater
        try { Thread.sleep(10); } catch (InterruptedException ex) {} // vent lidt
        g2.setTransform( orgTrans );                 // genskab orig. transformation
        g2.setColor(Color.WHITE);                    // rens skærmen selv
        Dimension d = getSize();
        g2.fillRect(0, 0, d.width, d.height);
        g2.setColor(Color.BLACK);

        for (int n=0; n<koord.length; n++) {
            double k[] = koord[n];
            double t = k[0] = (k[0] + 0.005) % 1;    // opdater 'tid' k[0] og put i t
            g2.setTransform( orgTrans );              // genskab orig. transformation
            g2.translate( (t*(k[1]-0.5)+0.5)*d.width, (t*(k[2]-0.5)+0.5)*d.height );
            g2.scale(t*k[3], t*(k[4]+t-1));           // flyt, skaler og rotér
            g2.rotate(t*50*(k[5]-0.5));
            g2.translate(-fig.getBounds().width/2, -fig.getBounds().height/2 );
            g2.draw(fig);
        }
    }

    public static void main(String[] args)
    {
        AktivVisning vindue = new AktivVisning();
        vindue.setSize(400,300);
        vindue.init();

        for (int i=0; i<10000; i++)
            vindue.tegn();
    }
}
```



Ved opstart får vi fat i Graphics-objektet (med `this.getGraphics()`), og det bruger vi så direkte i `tegn()`-metoden. Vi definerer ikke `paintComponent()` og systemet vil derfor ikke fortælle os, når vinduet skal gentegnes<sup>1</sup> (vi gentegner vinduet efter 1/100 sekund, så det gør ikke så meget).

## 9.4.5 Fuldskærmstegning

Til f.eks. spil (hvor aktiv visning næsten altid bruges) er det rart at have fuld adgang til hele skærmen, uden at andre vinduer muligvis dækker for (noget af) vinduet.

Det giver bedre mulighed for at udnytte grafikortets hardwareacceleration, og man kan

<sup>1</sup> Det er dog muligt at lave en blanding af aktiv og passiv visning. Havde `tegn()` kun optegnet billedet, dvs ikke også ventet og opdateret koordinater, kunne vi kalde `tegn()` fra `paintComponent()`.

også bl.a. skifte opløsning på skærmen (det er ikke vist herunder, men man kan få et array af mulige opløsninger med `dev.getDisplayModes()` derpå med `dev.setDisplayMode()` vælge en af dem). Læs mere på <http://java.sun.com/docs/books/tutorial/extra/fullscreen/>.

```
import java.awt.*;

public class AktivVisningFuldskærm extends AktivVisning
{
    public static void initFuldskærm(Frame vindue) {
        GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
        GraphicsDevice dev = env.getDefaultScreenDevice();
        vindue.setUndecorated(true); // vis ikke vinduesramme, titel, luk-knap etc.
        dev.setFullScreenWindow(vindue);
    }

    public static void main(String[] args)
    {
        AktivVisning vindue = new AktivVisning();
        initFuldskærm(vindue);
        vindue.init();

        for (int i=0; i<10000; i++) {
            vindue.tegn();
        }
    }
}
```

## 9.4.6 Dobbeltbuffer

Tager det lang tid at tegne vinduet, kan man ikke undgå at blive distraheret af, at det blinker. I stedet for at tegne direkte på skærmen og lade brugeren "følge med" i den langsomme gentegning, kan man tegne billedet i en skjult grafikbuffer – en "fiktiv skærm". Først når billedet er tegnet færdigt i grafikbufferen, kopieres resultatet ud på skærmen. Dette kaldes at bruge dobbeltbuffer (eng.: double buffering).

Herunder er eksemplet udvidet til at bruge dobbeltbuffer (det vigtigste i fed):

```
import java.awt.*;
import java.awt.image.*;

public class AktivVisningFuldskærmBufferet extends AktivVisningFuldskærm
{
    public static void main(String[] args)
    {
        AktivVisning vindue = new AktivVisning();
        initFuldskærm(vindue);
        vindue.init();
        vindue.createBufferStrategy(2); // opret 2 buffere
        BufferStrategy bufferStrategy = vindue.getBufferStrategy();

        for (int i=0; i<10000; i++) {
            vindue.g2 = (Graphics2D) bufferStrategy.getDrawGraphics(); // få buffer
            vindue.tegn(); // tegn på bufferens (med dens Graphics-objekt)
            bufferStrategy.show(); // vis grafikken EFTER at der er tegnet færdigt
            vindue.g2.dispose(); // frigiv bufferen så den er klar til genbrug
        }
    }
}
```

Efter `init()` hvor vinduet er åbnet og vi kender dets størrelse, beder vi om, at vinduet får to tegnebuffer (normalt er der en, nemlig den, der vises på skærmen).

Derefter beder vi, før hver gentegning, om `Graphics`-objektet på den buffer, der *ikke* bliver vist på skærmen lige nu. Den buffer tegner vi så på og viser så først billedet *efter*, at det er tegnet færdigt<sup>1</sup>. Derefter frigiver vi bufferens `Graphics`-objekt.

---

1 Bliver grafikken vist i et vindue vil det billede blive kopieret hen til det rigtige sted i hukommelsen. I fuldskærmstilstand vil grafikkortet i stedet blive dirigeret til at vise det nye hukommelsesområde (hvis grafikkortet og styresystemet understøtter det), hvilket er meget hurtigere,

# 10 Appletter

Indhold:

- At lægge en Java-applet i en hjemmeside
- Metoder i en applet

Kapitlet forudsættes ikke i resten af bogen, men appletter anvendes i nogle eksempler i kapitel 12, Interfaces og 13, Hændelser.

Forudsætter kapitel 9, Grafiske programmer og lidt kendskab til HTML.

En applet er et javaprogram i en hjemmeside. Når siden vises, vil netlæseren (browseren, fremviseren af hjemmesiden) hente javaprogrammet og udføre det på brugerens maskine. Ordet "applet" giver mange associationer til "en lille applikation".

## 10.1 HTML-dokumentet

Hjemmesider skrives i et sprog, der hedder HTML. For mere viden om HTML henvises til de mange introduktioner til, hvordan man laver hjemmesider.

En side med en applet vil have HTML-koden <applet>, der henviser til, hvor netlæseren skal finde programkoden. Et HTML-dokument med en applet kunne se sådan her ud:

```
<html>
<head>
  <title>
    Min applet
  </title>
</head>

<body>
  Velkommen til min første applet! <br>
  <applet code="MinApplet.class" width="400" height="300"></applet> <br>
  Slut herfra!
</body>
</html>
```

Her blev i <applet>-koden angivet, at appletten hedder MinApplet og at den skal være 400 punkter bred og 300 høj. Filen MinApplet.class (den binære kode, som oversætteren laver ud fra kildetekstfilen MinApplet.java) skal ligge samme sted som HTML-dokumentet.

## 10.2 Javakoden

Selve javaprogrammet er en klasse, der arver fra JApplet og har en paint()-metode (for appletter bruges paint() i stedet for paintComponent()), hvor grafikken tegnes. Eksempel:

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

public class MinApplet extends JApplet
{
    public void paint(Graphics g)
    {
        // Herunder referer g til et Graphics-objekt man kan tegne med
        super.paint(g);    // tegn først baggrunden (og evt komponenter)

        g.drawLine(0,0,50,50);

        g.fillOval(5,10,300,30);

        g.setColor(Color.GREEN);

        g.drawString("Hej grafiske verden!",100,30);
    }
}
```

Det ligner altså præcist koden fra forrige kapitel (sammenlign med Grafikpanel kapitel 9), ud over at tegnemethoden her hedder paint() i stedet for paintComponent() og at vi her arver fra JApplet i stedet for JPanel. Dog er det ikke nødvendigt med en BenytGrafikpanel-klasse, der åbner vinduet, da netlæseren sørger for dette ud fra HTML-koden.

Herunder ses, hvordan HTML-koden med appletten ser ud i en netlæser (Netscape på Linux). Man ser, at først kommer HTML-teksten "Velkommen til ...", derunder appletten og nederst igen teksten "Slut herfra!" fra HTML-koden.



## 10.3 Metoder i appletter

En applet skal spille sammen med HTML-koden og fremvisningen. Der ligger faktisk et stort maskineri bagved, der sørger for, at den bliver vist korrekt og får relevante oplysninger om, hvad brugeren gør.

Derfor har appletter nogle metoder, som det kan være nyttigt at kende. De er delt i to grupper, nemlig dem som *du kan kalde* og dem som *systemet kalder* (og som du kan omdefinere for at få udført noget af din kode, når de kaldes).

### 10.3.1 Metoder i appletter, som du kan kalde

Disse metoder kan du kalde, når du programmerer appletter. Det er kun de vigtigste af metoderne, der er gengivet (se Javadokumentationen for en komplet liste).

*Nogle af JApplet-klassens metoder – se også dem i JFrame-klassen afsnit 9.2, der er mærket med (generel).*

URL <b>getDocumentBase()</b>	giver URL'en til der, hvor HTML-dokumentet ligger.
URL <b>getCodeBase()</b>	giver URL'en (adressen) til der, hvor .class-filen er. Ofte det samme som getDocumentBase().
Image <b>getImage</b> (URL url, String filnavn)	returnerer et billede-objekt, typisk fra en .jpg eller .gif-fil. F.eks kunne et billede, der lå samme sted som appletten, hentes med getImage(getCodeBase(),"filnavn.gif").
AudioClip <b>getAudioClip</b> (URL url, String filnavn)	returnerer et lydclip-objekt, typisk fra en .wav-fil.
String <b>getParameter</b> (String parameternavn)	returnerer den pågældende parameterværdi, hvis den er defineret i HTML-koden, ellers null. En parameter sættes med <param name="navn" value="værdi"> mellem <applet ...> og </applet>.
void <b>showStatus</b> (String tekst)	viser en meddelelse i netlæserens statusfelt (hvor status for dokumentindlæsning plejer at stå).
<b>getAppletContext().showDocument</b> (URL url)	omdirigerer netlæseren til at vise en anden URL i dette vindue, sådan at siden med appletten bliver forladt. Findes også i en variant, der åbner en URL i et nyt/et andet navngivet vindue.

Bemærk: Ovenstående metoder fungerer først *efter*, at appletten er færdigoprettet. Brug dem derfor ikke fra din egen konstruktør, men definér metoden init() i stedet (se nedenfor).

## 10.3.2 Metoder i appletter, som systemet kalder

Disse metoder kan du definere i en applet. Systemet kalder dem på bestemte tidspunkter.

```
public void init()
```

Kaldes én gang, når netlæseren indlæser HTML-dokumentet og appletten. Det er en god ide at placere programkode, der opretter objekter og initialiserer appletten, i `init()`.

Da appletten er et objekt, kan det selvfølgelig også gøres fra konstruktøren, men vær da opmærksom på, at de metoder, du kan kalde (f.eks. `repaint()` og `getSize()`), ikke fungerer korrekt, da applettens omgivelser ikke er blevet initialiseret endnu.

```
public void start()
```

Kaldes, når appletten bliver synlig. Normalt sker det lige efter `init()`, men hvis HTML-dokumentet er meget stort og appletten er i bunden af dokumentet, kan det være, den ikke er synlig med det samme. Så kaldes `start()` først, når appletten bliver synlig for brugeren. `start()` kan godt blive kaldt flere gange, hvis appletten skjules og bliver synlig igen.

```
public void paint(Graphics g)
```

Her programmerer du, hvordan appletten skal se ud, ligesom forklaret i afsnit 9.1. Metoden kaldes, hver gang der er behov for at gentegne en del eller hele appletten. Det kan være ret så ofte, så man bør have så lidt kode som muligt her, så metoden kan udføres hurtigt.

```
public void stop()
```

Kaldes, når appletten bliver skjult, fordi vinduet bliver minimeret, eller fordi brugeren går til et andet dokument (det sker det samme antal gange, som `start()` kaldes).

```
public void destroy()
```

Kaldes, når appletten smides væk af netlæseren, fordi brugeren er gået til et andet dokument eller har lukket vinduet. `destroy()` bliver kun kaldt én gang. Er der noget, der er vigtigt at få gjort, inden programmet afsluttes, kan du lægge kode til at gøre det i `destroy()`.

## 10.3.3 Eksempel

Nedenfor er Kurvetegning fra afsnit 9.1.2 igen som en applet, hvor metoderne, som systemet kalder, alle er defineret til at udskrive en tekst, når de bliver kaldt.

Vi udskriver også størrelsen af appletten og hvor den kommer fra (`getDocumentBase()`):

```
Konstruktør kaldt - bredden er her: 0
init()          kaldt - bredden er her: 400
                  URL: file:/home/jacob/bog/kode/kapitel_10/Kurvetegningsapplet.html
start()         kaldt
paint()         kaldt
paint()         kaldt
paint()         kaldt
stop()          kaldt
destroy()       kaldt
```

Man ser, at rækkefølgen af kaldene er som beskrevet. Bemærk at mange data (bl.a. applettens størrelse) endnu ikke er kendt i konstruktøren, men først i `init()`. Det er også først fra og med `init()`, at metoder som `getDocumentBase()` og `showStatus()` kan kaldes.

Da appletten er blevet kørt direkte fra harddisken (i `/home/jacob/bog/kode/kapitel_10/`), giver kaldet til `getDocumentBase()` en URL til dette sted på det lokale filsystem.

```

import java.util.*;
import java.awt.*;
import javax.swing.*;

public class Kurvetegningsapplet extends JApplet
{
    ArrayList<Color> farver; // objektvariabel kendt i både init() og paint()
    int forskydning = 50; // en forskydning i farvevalget

    public Kurvetegningsapplet()
    {
        System.out.println("Konstruktør kaldt - bredden er her: "+getSize().width);
    }

    public void init() // Forbered punkterne i init(), ikke i konstruktøren
    {
        System.out.println("init() kaldt - bredden er her: "+getSize().width);
        System.out.println("URL: "+getDocumentBase());

        farver = new ArrayList<Color>();
        for (int i=0; i<400; i++)
        {
            Color farve = new Color(i%256, (i*2)%256, (i*4)%256);
            farver.add(farve);
        }
    }

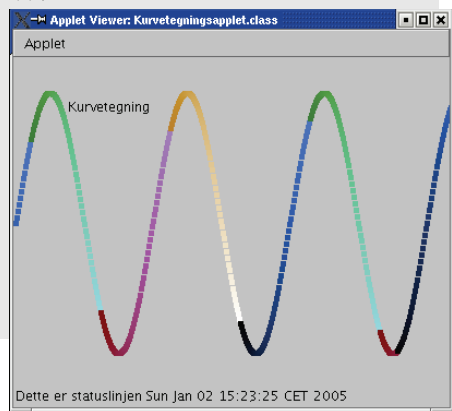
    public void start()
    {
        System.out.println("start() kaldt");
    }

    public void paint(Graphics g) // ikke paintComponent()
    {
        System.out.println("paint() kaldt");
        showStatus("Dette er statuslinjen "+new Date());
        g.drawString("Kurvetegning", forskydning%400, forskydning%300);
        for (int x=0; x<farver.size(); x++)
        {
            int y = 150 - (int) (120*Math.sin(0.05*x));
            int i = (x+forskydning)%400;
            Color farve = farver.get(i);
            g.setColor(farve);
            g.fillRect(x, y, 5, 5);
        }
    }

    public void stop()
    {
        System.out.println("stop() kaldt");
    }

    public void destroy()
    {
        System.out.println("destroy() kaldt");
    }
}

```



## 10.4 Resumé

- En applet er et grafisk miniprogram, der ligger i en hjemmeside (HTML-dokument).
- Det er netlæseren (eng.: browser), der sørger for at oprette appletten og vise den på skærmen. Derfor er det ikke nødvendigt at gøre det fra en main()-metode.
- Den har nogle metoder til fælles med andre grafiske objekter, navnlig metoden `paint(Graphics g)`, som systemet kalder, når appletten skal gentegnes.
- Der er også metoder, der kun understøttes i appletter, bl.a. `init()`, som systemet kalder én gang, når appletten indlæses. Derudover findes `start()`, `stop()` og `destroy()`. Disse metoder bliver kun kaldt i appletter, ikke i `JPanel` eller andre grafiske objekter.

Du kan læse mere om appletter på <http://java.sun.com/applets> og prøve nogen på f.eks. <http://spil.tv2.dk>.

## 10.5 Avanceret

Hvis man vil programmere seriøst med appletter, er der visse ting, man skal være opmærksom på i forhold til "rigtige" programmer med en main()-metode.

Bemærk: En anden mulighed end appletter er at bruge Java Web Start, der tillader et almindeligt grafisk program at blive distribueret og kørt via netværket. Læs mere om Java Web Start på adressen: <http://java.sun.com/products/javawebstart/developers.html>

### 10.5.1 Sikkerhedsbegrænsninger for appletter

Da appletter automatisk hentes over netværket og køres hos brugeren (ligesom de hjemmesider, de er en del af), er det nødvendigt med omfattende sikkerhedsforanstaltninger for at sikre, at "ondsindede appletter" ikke har adgang til at gøre skade.

Derfor er appletter forhindret i at gøre visse ting:

- Læse eller skrive filer på brugerens maskine (f.eks. ikke ændre i C:\AUTOEXEC.BAT).
  - Læse eller skrive i udklipsholderen (den kunne indeholde vigtige informationer).
  - Afslutte Java (med System.exit()).
  - Starte nye programmer (med Runtime.getRuntime().exec()).
  - Udskrive til en printer.
  - Forbinde sig til andre maskiner over netværket ud over den, de kom fra.
- F.eks. kan en applet ikke bruges til at udspionere det lokale netværk.
- Appletter kan kun hente eller sende data til den webserver, den selv kom fra.

Disse restriktioner kan ophæves ved at forsyne appletten med en digital signatur. Signaturen identificerer afsenderen og er knyttet til appletkoden sådan, at hvis nogen ændrer i applettens kode, passer signaturen ikke mere.

### 10.5.2 Pakker og CODE/CODEBASE i HTML-koden

Lad os igen se på HTML-koden til en applet

```
<applet codebase="." code="MinApplet.class" width=400 height=300></applet>
```

CODEBASE angiver, i hvilken mappe appletten skal findes og svarer til CLASSPATH. "." betyder "den aktuelle mappe", dvs. der hvor HTML-filen ligger. I så tilfælde kan CODEBASE også udelades. Her kunne også stå en henvisning til en anden maskine, f.eks.:

```
<applet codebase = "http://java.sun.com/applets/" ...
```

CODE angiver placeringen af filen i forhold til CODEBASE. Har man placeret sin klasse i en pakke, skal CODE være "pakkenavn/Klassenavn.class" og CODEBASE skal henvise til rodmappen (dvs. mappen, hvori "pakkenavn" ligger). Se også kapitel 6 om pakker.

### 10.5.3 Begrænsninger i ældre netlæsere

Når man programmerer appletter, støder man ofte på det problem, at appletten virker i udviklingsmiljøet, men ikke i en eller flere netlæsere. Det skyldes, at mange netlæsere kun understøtter en ældre (mindre) udgave af Java, oftest JDK 1.4, mens udviklingsmiljøerne bruger JDK version 1.6 eller senere. Fra version til version er der sket en udvidelse af antallet af klasser og visse klasser har fået tilføjet flere metoder. Hvis netlæseren mangler nogle klasser eller metoder, som appletten benytter, vil den ikke kunne køre appletten.

Det er lettest at se, hvad problemet er, ved at kigge i netlæserens Java-konsolvindue og se præcis, hvilken fejlmeddelelse netlæseren viser (i Mozilla findes det under: Værktøjer | Webudvikling | Java-konsol, i Opera under: Windows/Other/Java).



# 11 Grafiske standardkomponenter

Indhold:

- Design af en grafisk brugergrænseflade med et udviklingsværktøj
- De vigtigste grafiske komponenter og deres egenskaber
- Containere og layout-managere
- Menuer

Forudsættes af kapitel 13, Hændelser.

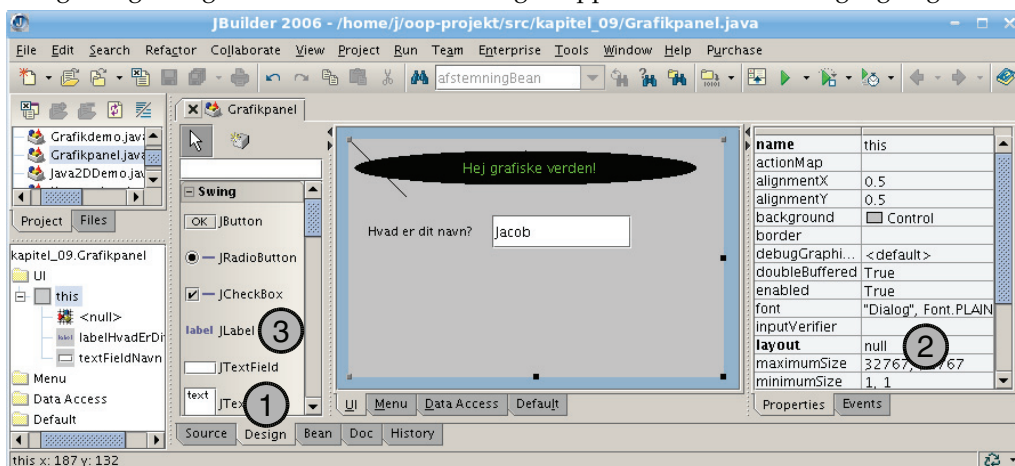
Kapitlet forudsætter kapitel 9, Grafiske programmer, og at du har adgang til et værktøj, der kan udvikle grafiske brugergrænseflader (f.eks. Borland JBuilder, Oracle JDeveloper, Netbeans eller Eclipse).

Når man skal lave en grafisk brugergrænseflade (eng.: GUI, graphical user interface), gøres det oftest ved at anvende standardkomponenter. Vi vil starte med at se på, hvordan det gøres i praksis med et værktøj, og derefter studere den kode, der kommer ud af det.

## 11.1 Generering med et værktøj

Med moderne udviklingsværktøjer kan man udarbejde en grafisk brugergrænseflade ud fra standardkomponenter på ret kort tid og uden at skulle programmere ret meget selv.

Herunder er beskrevet, hvordan man gør i JBuilder 2006 med sideløbende forklaringer til udviklingsværktøjerne Oracle JDeveloper og Netbeans. Bruger du et andet værktøj (herunder JBuilder 2007), må du prøve dig lidt frem. Ideerne er de samme og koden, der genereres, ligner også nogenlunde, men menuerne og knapperne varierer selvfølgelig noget.



I JBuilder 2006/JDeveloper er det muligt at tage en eksisterende klasse, der arver fra JPanel eller JApplet f.eks. Grafikpanel fra kapitel 9. Hvis du vil oprette en ny, så vælg 'New...' og Application eller Applet. Skriv et navn på din klasse og klik 'Finish'. Tegn evt. noget i paintComponent().

I Netbeans skal du under 'New File..' vælge 'Java GUI Forms' og 'JPanel Form'. Se evt.: <http://java.sun.com/docs/books/tutorial/uiswing/learn/index.html>. I andre værktøjer: Opret en ny JApplet eller JPanel (bruger du JPanel skal du også huske en Benyt...-klasse).

- 1) Gå over på Design-fanen (ved punkt 1 nederst). Den er delt op i en del, hvor du designer din brugergrænseflade i midten og en tabel af egenskaber til højre (punkt 2).  
JDeveloper: Højreklik på Grafikpanel.java og vælg 'UI Editor'.  
Netbeans: Gå over på fanen 'GUI Editing', hvis du ikke allerede står der.
- 2) Her skal du først ændre layout fra '<default layout>' til 'null' (punkt 2 til højre; måske skal du klikke på den grå flade i designeren først).  
Netbeans: Højreklik på den grå flade og vælg 'Set Layout' og 'Null layout'.
- 3) Nu kan du gå i gang med at lægge komponenter ind på grænsefladen.  
Vælg først en JLabel fra komponentpaletten (punkt 3) og klik på den grå flade. Der dukker en etikette med en tekst op. På egenskabstabellen til højre kan du ændre dens variabelnavn (*name* øverst) til f.eks. labelHvadErDitNavn. Længere nede er egenskaben *text*, der bestemmer, hvad der skal stå på etiketten. Ret den til f.eks. "Hvad er dit navn?".
- 4) Indsæt derefter et JTextField (et indtastningsfelt – lidt længere nede i listen).  
Ret variabelnavnet til textFieldNavn og teksten til f.eks. "Jacob".

Gå tilbage til Source-fanen. Nu ser kildeteksten nogenlunde således ud:

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

public class Grafikpanel extends JPanel
{
    JLabel labelHvadErDitNavn = new JLabel();
    JTextField textFieldNavn = new JTextField();

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);           // tegn først baggrunden på panelet
        g.drawLine(0,0,50,50);
        g.fillOval(5,10,300,30);
        g.setColor(Color.GREEN);
        g.drawString("Hej grafiske verden!",100,30);
    }

    public Grafikpanel() {
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        labelHvadErDitNavn.setText("Hvad er dit navn?");
        labelHvadErDitNavn.setBounds(new Rectangle(15, 69, 108, 15));
        textFieldNavn.setText("Jacob");
        textFieldNavn.setBounds(new Rectangle(141, 61, 112, 29));
        this.setLayout(null);
        this.add(textFieldNavn);
        this.add(labelHvadErDitNavn);
    }
}
```

De to objekter, vi satte på i designeren, er erklæret og oprettet øverst uden for metoderne:

```
JLabel labelHvadErDitNavn = new JLabel();
JTextField textFieldNavn = new JTextField();
```

Nedenunder står vores gamle paintComponent() uændret. Herunder er der oprettet en konstruktør, der kalder metoden jbInit() (i Netbeans hedder metoden initComponents()).

Den andet kode, 'try{ ... } catch (Exception e) {...}' er beregnet til at håndtere undtagelser og vil blive forklaret i kapitel 14, Undtagelser. Se bort fra den for nu.

I metoden jbInit() nedenunder lægger værktøjet koden til at initialisere de grafiske komponenter. Man ser her, hvordan både JLabel og JTextField har metoden setText() og at begge objekter får kaldt denne metode (svarende til, at vi ændrede egenskaben *text*).

```
labelHvadErDitNavn.setText("Hvad er dit navn?");
textFieldNavn.setText("Jacob");
```

De andre kommandoer i jbInit() sørger for at placere komponenterne korrekt i vinduet.

"Design"- og "Source"-fanen er to måder at se programmet på og man kan frit skifte mellem dem. Laver man en designændring, vil det blive afspejlet i koden i jbInit(). Ændrer man i koden, vil designet ændre sig (i Netbeans er kildekoden dog beskyttet mod ændringer).

---

**Retter du eller tilføjer kode til værktøjets genererede kode (i jbInit()), så sørg for, at det ligner værktøjets egen kode, ellers kan værktøjet have svært ved at opretholde sammenhængen mellem kode og design.**

---

Anden kode kan du putte i konstruktøren, f.eks. lige over eller under kaldet til jbInit().

## 11.1.1 Interaktive programmer

Lad os nu tilføje en knap og et indtastningsfelt på flere linjer (JTextArea). Jeg kalder dem for buttonOpdater og textAreaHilsen. Knappen skal selvfølgelig gøre noget. Fra Design-fanen, dobbeltklik på knappen, og vupti! Der genereres automatisk en metode til at håndtere et klik (og pakken java.awt.event bliver importeret):

```
void buttonOpdater_actionPerformed(ActionEvent e) {  
    }  
}
```

Hvis du kigger i jbInit(), kan du se, at JBuilder har indsat følgende kode:

```
buttonOpdater.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        buttonOpdater_actionPerformed(e);  
    }  
});
```

Det er disse linjer, der sørger for at "lytte efter hændelser" på knappen, sådan at når man klikker på buttonOpdater, så kaldes metoden buttonOpdater\_actionPerformed(). Det vil vi komme tilbage til i kapitel 13, Hændelser.

Nu kan du indsætte kode, der udfører en handling. Skriv f.eks. noget ud til systemoutput:

```
void buttonOpdater_actionPerformed(ActionEvent e) {  
    System.out.println("Opdater!");  
}
```

Vi kunne også lave noget sjovere, f.eks. læse den indtastede tekst fra textFieldNavn og skrive den i textAreaHilsen. JBuilder har lavet koden, der sætter teksterne for os og ved at studere den kan man få en ide til, hvordan det skal gøres:

```
String navn = textFieldNavn.getText(); // aflæs navnet  
textAreaHilsen.setText("Hej kære "+navn); // sæt navnet
```

Her har vi tastet "Jacob Nordfalk" ind og trykket på "opdater!"-knappen (paintComponent()) er ændret til også at tegne navnet 5 gange).



Her kommer det fulde eksempel med en main-metode, der viser vinduet.

```
import javax.swing.*;  
public class BenytGrafikpanelMedKomponenter  
{  
    public static void main(String[] arg)  
    {  
        JFrame vindue = new JFrame( "GrafikpanelMedKomponenter" );  
        vindue.add( new GrafikpanelMedKomponenter() );  
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);  
        vindue.setSize(350,300);  
        vindue.setVisible(true);  
    }  
}
```

```

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

public class GrafikpanelMedKomponenter extends JPanel
{
    JLabel labelHvadErDitNavn = new JLabel();
    JTextField textFieldNavn = new JTextField();
    JButton buttonOpdater = new JButton();
    JTextArea textAreaHilsen = new JTextArea();

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g); // tegn baggrunden på panelet
        g.drawLine(0,0,50,50);
        g.fillOval(5,10,300,30);
        g.setColor(Color.GREEN);
        String navn = textFieldNavn.getText();
        for (int i=0; i<50; i=i+10)
            g.drawString("Hej "+navn+" !",100+i,30+i);
    }

    public GrafikpanelMedKomponenter() {
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    /** Udviklingsværktøjets initialisering af komponenter.
     * Udviklingsværktøj definerer gerne en separat metode hvor de
     * initialiserer komponenterne. I JBuilder og JDeveloper hedder
     * den jbInit(), mens den hedder initComponents() i Betbeans.
     * Initialiseringen kunne dog lige så godt ligge direkte i konstruktøren.
     * Ændr med varsomhed, ellers kan værktøjet ikke genkende "sin" kode!
     */
    private void jbInit() throws Exception {
        labelHvadErDitNavn.setText("Hvad er dit navn?");
        labelHvadErDitNavn.setBounds(new Rectangle(15, 69, 108, 15));
        textFieldNavn.setText("Jacob");
        textFieldNavn.setBounds(new Rectangle(129, 61, 95, 29));
        buttonOpdater.setText("Opdater!");
        buttonOpdater.setMnemonic(KeyEvent.VK_O);
        buttonOpdater.setBounds(new Rectangle(231, 60, 91, 32));
        buttonOpdater.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                buttonOpdater_actionPerformed(e);
            }
        });
        textAreaHilsen.setText("Her kommer en tekst...");
        textAreaHilsen.setBounds(new Rectangle(6, 102, 316, 78));
        this.setLayout(null);
        this.add(labelHvadErDitNavn);
        this.add(textAreaHilsen);
        this.add(buttonOpdater);
        this.add(textFieldNavn);
    }

    void buttonOpdater_actionPerformed(ActionEvent e) {
        String navn = textFieldNavn.getText();
        System.out.println("Opdater! navn="+navn);
        textAreaHilsen.setText("Hej kære "+navn);
        repaint(); // gentegn vinduet
    }
}

```

## 11.1.2 Genvejstaster

Genvejstaster kan laves på de fleste komponenter med kaldet `setMnemonic()`. I ovenstående eksempel kan man trykke Alt-O for at trykke på 'Opdater'-knappen, fordi vi kaldte:

```
buttonOpdater.setMnemonic(KeyEvent.VK_O);
```

## 11.2 Overblik over komponenter

Grafiske komponenter er objekter, der bruges som en synlig del af en grafisk brugergrænseflade, f.eks. knapper, valglister, indtastningsfelter, etiketter.

Alle komponenter arver fra JComponent-klassen og har derfor dennes træk til fælles:

Metoderne `setForeground(Color c)` og `setBackground(Color c)` sætter hhv. forgrundsfarven og baggrundsfarven for komponenten, svarende til egenskaberne *foreground* og *background*. Egenskaberne kan aflæses med `getForeground()` og `getBackground()`.

En anden egenskab er *font*, der bestemmer skrifttypen. I tråd med de andre egenskaber sættes den med `setFont(Font f)` og aflæses med `getFont()`.

Dette kan sammenfattes i en tabel over egenskaber, der er fælles for alle komponenter.

Egenskab	Type	Sættes med	Læses med
foreground	Color	<code>setForeground(Color c)</code>	<code>getForeground()</code>
background	Color	<code>setBackground(Color c)</code>	<code>getBackground()</code>
font	Font	<code>setFont(Font f)</code>	<code>getFont()</code>
visible	boolean	<code>setVisible(boolean synlig)</code>	<code>isVisible()</code>

Nedenfor er de mest almindelige komponenter beskrevet sammen med deres egenskaber.

### 11.2.1 JLabel

En Label

En etiket, der viser en tekst (som brugeren ikke kan redigere i).

Udover de fælles egenskaber findes egenskaben *text*, der angiver, hvad der står i feltet.

Egenskab	Type	Sættes med	Læses med
text	String	<code>setText(String t)</code>	<code>getText()</code>

Mere info: <http://java.sun.com/docs/books/tutorial/uiswing/components/label.html>

### 11.2.2 JButton



En trykknop. Egenskaben *text* angiver, hvad der står på knappen.

Egenskab	Type	Sættes med	Læses med
text	String	<code>setText(String t)</code>	<code>getText()</code>

### 11.2.3 JCheckBox og JRadioButton



JCheckBox giver et afkrydsningsfelt, som brugeren kan sætte et flueben i (eller fjerne).

JRadioButton bruges til radioknapper, der gensidigt udelukker hinanden. En gruppe af radioknapper skal knyttes sammen af et ButtonGroup-objekt (se eksemplet senere).

*text* angiver, hvad der står ved feltet. *selected* angiver, om feltet/radioknappen er afkrydset.

Egenskab	Type	Sættes med	Læses med
text	String	<code>setText(String t)</code>	<code>getText()</code>
selected	boolean	<code>setSelected (boolean afkrydset)</code>	<code>isSelected()</code>

## 11.2.4 JTextField



Et indtastningsfelt på én linje. Egenskaben *text* angiver, hvad der står i feltet. *columns* angiver, hvor bredt feltet skal være.

*editable* angiver, om brugeren kan redigere teksten i indtastningsfeltet.

Egenskab	Type	Sættes med	Læses med
text	String	setText(String t)	getText()
editable	boolean	setEditable(boolean rediger)	isEditable()
columns	int	setColumns(int bredde)	getColumns()

## 11.2.5 JTextArea



Et indtastningsfelt på flere linjer.

Egenskaberne *text*, *rows* og *columns* angiver, hvad der står i feltet, hhv. bredde og højde.

Egenskab	Type	Sættes med	Læses med
text	String	setText(String t)	getText()
editable	boolean	setEditable(boolean rediger)	isEditable()
columns	int	setColumns(int bredde)	getColumns()
rows	int	setRows(int højde)	getRows()

JTextField og JTextArea har en del egenskaber til fælles og disse fællestræk ligger i superklassen JTextComponent, der også har andre arvinger, såsom JPasswordField og JTextPane (der tillader redigering tekst med typografier, såsom HTML og RTF). Læs mere om disse på: <http://java.sun.com/docs/books/tutorial/uiswing/components/text.html>

## 11.2.6 JComboBox



En valgliste. Det er nemmest at bruge metoden addItem(String elementnavn) til at tilføje indgange, men der er også bl.a. mulighed for at give et array af elementer i konstruktøren.

```
JComboBox comboBox1 = new JComboBox();  
...  
comboBox1.addItem("ComboBox Rød");  
comboBox1.addItem("ComboBox Grøn");  
comboBox1.addItem("ComboBox Blå");
```

Med getItem() undersøger man, hvad brugeren har valgt. Hvis man foretrækker at kende nummeret på det valgte element, bruger man selectedIndex().

Se også: <http://java.sun.com/docs/books/tutorial/uiswing/components/combobox.html>

## 11.2.7 JList

List rød  
List grøn  
List blå

En liste, hvor flere af indgangene kan ses samtidigt og hvor man kan vælge en eller flere elementer. Desværre er komponenten lidt svær at bruge for begyndere; det nemmeste er at oprette et array af elementer og overføre arrayet i konstruktøren:

```
String[] listedata = {"List rød", "List grøn", "List blå"};  
JList list1 = new JList(listedata);
```

Med `getSelectedValue()` undersøger man, hvad brugeren har valgt. Hvis man foretrækker at kende nummeret på det valgte element bruger man `getSelectedIndex()`.

Egenskaberne `rows` og `multipleMode` angiver hhv. hvor mange indgange, der kan ses ad gangen og om man kan vælge flere indgange samtidigt (ved at holde Ctrl nede og klikke).

Egenskab	Type	Sættes med	Læses med
<code>rows</code>	<code>int</code>	<code>setRows(int højde)</code>	<code>getRows()</code>
<code>multipleMode</code>	<code>boolean</code>	<code>setMultipleMode(boolean m)</code>	<code>getMultipleMode()</code>

Er `multipleMode` slået til, kan man løbe gennem antallet af indgange og for hver indgang bruge `isSelected(int indeks)` til at se, om indgangen er valgt.

## 11.3 Eksempel

Herunder et eksempel (genereret med JBuilder) med komponenterne omtalt i forrige afsnit.



```
import javax.swing.*;  
public class BenytOverblikOverKomponenter  
{  
    public static void main(String[] arg)  
    {  
        JFrame vindue = new JFrame( "OverblikOverKomponenter" );  
        vindue.add( new OverblikOverKomponenter() );  
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);  
        vindue.setSize(500,150);  
        vindue.setVisible(true);  
    }  
}
```



```

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

public class OverblikOverKomponenter extends JPanel
{
    // opret alle komponenterne og husk dem i nogle objektvariabler
    JLabel label1 = new JLabel();
    JButton button1 = new JButton();
    JCheckBox checkbox1 = new JCheckBox();
    JCheckBox checkbox2 = new JCheckBox();
    JCheckBox checkbox3 = new JCheckBox();
    JRadioButton radio1 = new JRadioButton();
    JRadioButton radio2 = new JRadioButton();
    ButtonGroup buttonGroup1 = new ButtonGroup();
    JTextField textField1 = new JTextField();
    JTextArea textArea1 = new JTextArea();
    JComboBox comboBox1 = new JComboBox();

    String[] listedata = {"List rød", "List grøn", "List blå"};
    JList list1 = new JList(listedata);

    FlowLayout flowLayout1 = new FlowLayout(); // layout-manager (se senere)

    public OverblikOverKomponenter() {
        try {
            jbInit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        label1.setText("En Label");

        button1.setText("OK");

        checkbox1.setText("En"); // sæt afkrydsningsfelternes navne
        checkbox2.setText("To");
        checkbox3.setText("Tre");

        radio1.setText("Radio1"); // sæt radioknappernes navne og
        radio2.setText("Radio2");
        buttonGroup1.add(radio1); // gruppe - så de gensidigt udelukker hinanden
        buttonGroup1.add(radio2);
        radio1.setSelected(true);

        comboBox1.addItem("ComboBox Rød");
        comboBox1.addItem("ComboBox Grøn");
        comboBox1.addItem("ComboBox Blå");

        textField1.setColumns(10);
        textField1.setText("Et TextField");

        textArea1.setColumns(15);
        textArea1.setRows(5);
        textArea1.setText("Et TextArea");

        this.setLayout(flowLayout1); // sæt layout-manager (se senere)

        this.add(label1); // til sidst skal komponenterne føjes
        this.add(button1); // til containeren (se senere)
        this.add(checkbox1);
        this.add(checkbox2);
        this.add(checkbox3);
        this.add(radio1);
        this.add(radio2);
        this.add(textField1);
        this.add(textArea1);
        this.add(comboBox1);
        this.add(list1);
    }
}

```

## 11.4 Overblik over containere

En *container* er beregnet til at indeholde komponenter og styre, hvordan de vises på skærmen. Alle containere har en såkaldt *layout-manager* tilknyttet, der hjælper containeren med at afgøre, hvor og med hvilken størrelse komponenterne skal vises.

For at en komponent bliver vist, skal den tilføjes en container. I eksemplet ovenfor er panelet den container, komponenterne bliver tilføjet og derfor står der sidst i initialiseringen:

```
this.add(button1);
```

### 11.4.1 JWindow

JWindow repræsenterer et bart vindue uden en titellinje eller lukkeknop øverst. Det bruges meget sjældent direkte. Man bruger i stedet arvingerne JFrame og JDialog.

### 11.4.2 JDialog

Dialog bruges til dialog-bokse, vinduer, der dukker op med et eller andet spørgsmål, som skal besvares, før man kan gå videre. Egenskaben *modal* angiver, om dialog-boksen er modal, dvs. om man skal lukke den, før man kan få adgang til ejer-vinduet. Den sættes med `setModal(boolean m)` og aflæses med `isModal()`. Hvis vinduet er modalt vil et kald til `setVisible(true)` vente, indtil brugeren har udfyldt og lukket dialog-boksen igen.

I mange tilfælde vil `JOptionPane`, der er beskrevet i afsnit 2.12.1, være nemmere at bruge. Således kunne man f.eks. vise panelet `OverblikOverKomponenter` i en dialog med koden:

```
import javax.swing.*;
public class DialogMedOverblikOverKomponenter
{
    public static void main(String[] arg)
    {
        JOptionPane.showMessageDialog(null, new OverblikOverKomponenter());
    }
}
```



### 11.4.3 JFrame

En JFrame er den simpleste og oftest brugte måde at definere et "normalt" vindue med titel, luk-knap, minimér-knap etc. Den er brugt i næsten alle eksempler i denne bog.

### 11.4.4 JPanel

Et panel er den simpleste og oftest brugte container. Den indeholder simpelthen komponenterne (i henhold til layout-manageren). JPanel er samtidig også en komponent, dvs. man kan tilføje den til enhver anden container (et vindue eller endda til et andet JPanel).

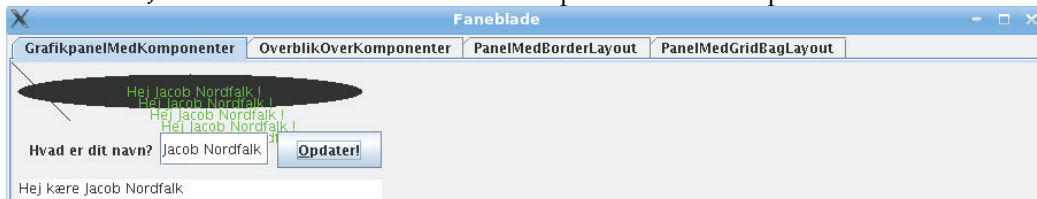
Kan ens vindue opdeles i flere logiske dele (f.eks. har JBuilder i afsnit 11.1 en komponent-palette, designvindue og egenskabstabel) er det god praksis at lægge disse i hver sit panel.

### 11.4.5 JApplet

En applet er et panel, der er beregnet til at blive vist i en netlæser. Se kapitel 10, Appletter.

## 11.4.6 JTabbedPane

Et panel, der viser et sæt af faneblade, der hver kan indeholde én komponent – som næsten altid er et JPanel. Herunder viser vi alle eksemplerne fra dette kapitel i hver sit faneblad:



```
import javax.swing.*;
public class BenytFaneblade
{
    public static void main(String[] arg)
    {
        JTabbedPane faneblade = new JTabbedPane();

        faneblade.add("GrafikpanelMedKomponenter", new GrafikpanelMedKomponenter());
        faneblade.add("OverblikOverKomponenter", new OverblikOverKomponenter());
        faneblade.add("PanelMedBorderLayout", new PanelMedBorderLayout());
        faneblade.add("PanelMedGridBagLayout", new PanelMedGridBagLayout());

        JFrame vindue = new JFrame("Faneblade");
        vindue.add( faneblade );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // reagér på luk
        vindue.pack(); // lad vinduet selv bestemme sin størrelse
        vindue.setVisible(true); // åbn vinduet
    }
}
```

## 11.5 Layout-managere

En layout-manager styrer layoutet af komponenterne på et JPanel eller en anden container. Alle containere har egenskaben *layout*, der kan sættes med metoden `setLayout(Layout l)`.

For mere info, se: <http://java.sun.com/docs/books/tutorial/uiswing/layout/>

### 11.5.1 Ingen styring (null-layout)

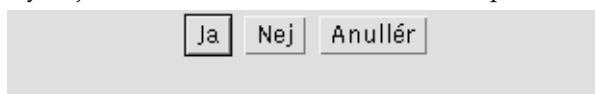
I udviklingsfasen er det mest bekvemt at sætte layout-manageren til null. Dette tillader udvikleren at sætte komponenterne, som han vil på en hvilken som helst (x,y)-position og med en hvilken som helst højde og bredde. Koden ser således ud:

```
this.setLayout(null); // sæt null-layout
this.add(button1); // tilføj knap
button1.setBounds(new Rectangle(231, 60, 91, 32)); // sæt position/størrelse
```

null-layout tager slet ikke højde for vinduets størrelse, så hvis vinduet bliver for lille, vil nogle af komponenterne ikke blive vist. Når programmet er ved at være færdigt, bør man derfor ændre programmet til at bruge en layout-manager, der kan styre komponenternes størrelse og indbyrdes placering ud fra deres behov.

### 11.5.2 FlowLayout

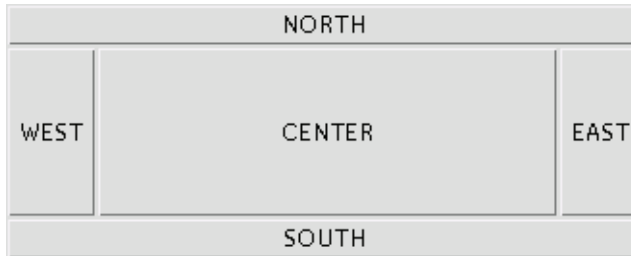
FlowLayout placerer komponenterne ligesom tekst, der er centreret: Øverst fra venstre mod højre og på en ny linje nedenunder, når der ikke er mere plads.



I eksemplet OverblikOverKomponenter ovenfor blev FlowLayout brugt (egentligt var det overflødigt at angive, for angiver man ikke nogen layout-manager i et panel/applet, vil FlowLayout blive brugt automatisk).

## 11.5.3 BorderLayout

BorderLayout tager højde for vinduets størrelse og tilpasser komponenternes størrelse efter den tilgængelige plads. Komponenterne kan placeres på 5 mulige positioner, nemlig NORTH, SOUTH, EAST, WEST og CENTER.



Den mest almindelige måde at lave det grafiske layout af et skærbillede er med BorderLayout. I de områder, hvor man ønsker at placere flere komponenter, sætter man først et JPanel og komponenterne tilføjes så panelet.

Angiver man ikke nogen layout-manager i et vindue, vil BorderLayout blive brugt (i eksemplet nedenfor kunne de to linjer omkring BorderLayout strengt taget fjernes).

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

public class PanelMedBorderLayout extends JPanel
{
    JButton button1 = new JButton();
    JButton button2 = new JButton();
    JButton button3 = new JButton();
    JButton button4 = new JButton();
    JButton button5 = new JButton();

    BorderLayout borderLayout1 = new BorderLayout();

    public PanelMedBorderLayout() {
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        button1.setText("NORTH");
        button2.setText("SOUTH");
        button3.setText("EAST");
        button4.setText("WEST");
        button5.setText("CENTER");

        this.setLayout(borderLayout1);

        this.add(button1, BorderLayout.NORTH);
        this.add(button2, BorderLayout.SOUTH);
        this.add(button3, BorderLayout.EAST);
        this.add(button4, BorderLayout.WEST);
        this.add(button5, BorderLayout.CENTER);
    }
}
```

Man sætter altså først layoutet ved at kalde setLayout() med et BorderLayout-objekt. Derefter kan add() kaldes med komponenterne og deres placering på borderlayoutet.

Bemærk, at når man bruger en layout-manager, bør man lade den afgøre vinduets størrelse ud fra komponenternes behov, ved at kalde `pack()` i stedet for `setSize()` på vinduet. Denne metode pakker komponenterne i vinduet optimalt og sætter vinduesstørrelsen passende.

```
import javax.swing.*;
public class BenytPanelMedBorderLayout
{
    public static void main(String[] arg)
    {
        JFrame vindue = new JFrame( "PanelMedBorderLayout" );
        vindue.add( new PanelMedBorderLayout() );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        vindue.pack();// sætter en rimelig vinduesstørrelse (i stedet for setSize())
        vindue.setVisible(true);
    }
}
```

## 11.5.4 GridBagLayout

En anden måde at lave layout er med `GridBagLayout`, som lægger komponenterne efter et usynligt gitter. Hver komponent kan fylde en eller flere celler i højden eller bredden.



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

public class PanelMedGridBagLayout extends JPanel
{
    GridBagLayout gridBagLayout1 = new GridBagLayout();
    JButton knap1 = new JButton();
    JButton knap2 = new JButton();
    JButton knap3 = new JButton();
    JButton knap4 = new JButton();
    JButton knap5 = new JButton();
    JCheckBox chkHø = new JCheckBox();
    JCheckBox chkVe = new JCheckBox();
    JCheckBox chkCe = new JCheckBox();
    JTextArea tekst = new JTextArea();

    public PanelMedGridBagLayout() {
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        knap1.setText("knap1 (på 3x1 celler)");
        knap2.setText("knap2 (1x2)");
        knap3.setText("knap3 (på 1x1 celle)");
        knap4.setText("knap4 (1x1)");
        knap5.setText("knap5 (1x1)");
        chkHø.setText("Højre");
        chkVe.setText("Venstre");
        chkCe.setText("Centreret");
        tekst.setColumns(15);
        tekst.setRows(2);
        tekst.setText("Tekstfelt (3x3 celler)");

        this.setLayout(gridBagLayout1);
    }
}
```

```
// til sidst skal komponenterne føjes til containeren
this.add(knap1, new GridBagConstraints(0, 0, 3, 1, 0.0, 0.0,
GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(0,0,0,0),0,0));
this.add(knap2, new GridBagConstraints(3, 0, 1, 2, 0.0, 0.0,
GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(0,0,0,0),0,0));
this.add(knap3, new GridBagConstraints(0, 1, 1, 1, 0.0, 0.0,
GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(0,0,0,0),0,0));
this.add(knap4, new GridBagConstraints(1, 1, 1, 1, 0.0, 0.0,
GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(0,0,0,0),0,0));
this.add(knap5, new GridBagConstraints(2, 1, 1, 1, 0.0, 0.0,
GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(0,0,0,0),0,0));
this.add(chkHø, new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0,
GridBagConstraints.EAST, GridBagConstraints.NONE, new Insets(0,0,0,0),0,0));
this.add(chkVe, new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0,
GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(0,0,0,0),0,0));
this.add(chkCe, new GridBagConstraints(0, 4, 1, 1, 0.0, 0.0,
GridBagConstraints.CENTER, GridBagConstraints.NONE, new Insets(0,0,0,0),0,0));
this.add(tekst, new GridBagConstraints(1, 2, 3, 3, 0.0, 0.0,
GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(0,0,0,0),0,0));
}
}
```

Når en komponent tilføjes, angives i et GridBagConstraints-objekt:

- Komponentens position (cellekolonne og -række)
- Komponentens spændvidde i højde og bredde
- Vægt i højde og bredde (komponenter med størst værdi får mest af eventuel overskydende plads)
- Justering i tilfælde af overskydende plads (CENTER, EAST, WEST, NORTHEAST, ...)
- Om komponenten skal strækkes til at fylde overskydende plads (BOTH, NONE, HORIZONTAL, VERTICAL)
- Til sidst nogle parametre til indsættelse af ekstra plads.

Her er koden, der viser vinduet.

```
import javax.swing.*;
public class BenytPanelMedGridBagLayout
{
    public static void main(String[] arg)
    {
        JFrame vindue = new JFrame( "PanelMedGridBagLayout" );
        vindue.add( new PanelMedGridBagLayout() );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        vindue.pack();
        vindue.setVisible(true);
    }
}
```

## 11.6 Menuer

En JFrame, JApplet og JDialog (men ikke JPanel og andre containere som ikke er selvstændige vinduer) kan have en menubjælke med rullegardiner tilknyttet.

Herunder har vi puttet en menu på vinduet, der viser GrafikpanelMedKomponenter.



Menuer er relativt enkle at lave: En menubjælke laves med `new JMenuBar()`, en menu (et rullegardin) med `new JMenu()` og et menupunkt med `new JMenuItem()`. Derefter skal menupunkterne tilføjes menuerne, og menuerne føjes til menubjælken (er det en undermenu føjes den til overmenuen) med `add()`. Man sætter teksterne med `setText()`. Til sidst sættes menubjælken på vinduet ved at kalde `vindue.setJMenuBar(menubjælke)`.

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

public class BenytGrafikpanelMedKomponenterOgMenu
{
    public static void main(String[] arg)
    {
        JFrame vindue = new JFrame( "GrafikpanelMedKomponenterOgMenu" );
        final GrafikpanelMedKomponenter panel = new GrafikpanelMedKomponenter();
        vindue.add( panel );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        vindue.setSize(350,300);

        JMenuBar menubjælke = new JMenuBar();

        JMenu    rullegardinFil    = new JMenu();
        JMenuItem menupunktOpdat   = new JMenuItem();
        JMenuItem menupunktAfslut  = new JMenuItem();

        JMenu    rullegardinHjælp = new JMenu();

        rullegardinFil.setText("Fil");
        rullegardinFil.setMnemonic(KeyEvent.VK_F);
        menupunktOpdat.setText("Opdater");
        menupunktAfslut.setText("Afslut");
        menupunktAfslut.setMnemonic(KeyEvent.VK_A);

        rullegardinHjælp.setText("Hjælp");

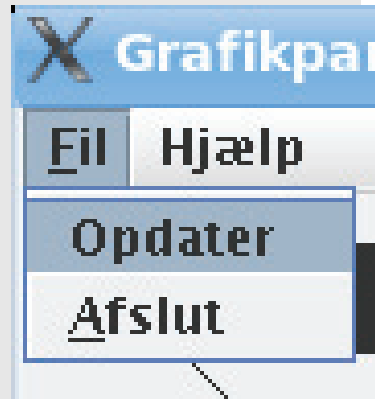
        rullegardinFil.add(menupunktOpdat);
        rullegardinFil.add(menupunktAfslut);
        menubjælke.add(rullegardinFil);
        menubjælke.add(rullegardinHjælp);

        vindue.setJMenuBar(menubjælke);

        menupunktOpdat.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                panel.buttonOpdater_actionPerformed(e);
            }
        });

        menupunktAfslut.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("Farvel!");
                System.exit(0);
            }
        });

        vindue.setVisible(true); // som det allersidste: vis vinduet
    }
}
```



Man kan også sætte genvejstaster på menuer, med `setMnemonic()`. Disse aktiveres, som andre genvejstaster, med Alt-tasten, sådan at Alt-F åbner Fil-menuen.

For at fange når brugeren vælger et menupunkt, skal man lytte efter `actionPerformed` ligesom med `JButton`-klassen (det blev diskuteret i afsnit 11.1.1).

Mere info: <http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html>

## 11.7 Test dig selv

- 1) Hvad er en grafisk komponent?
- 2) Giv tre eksempler på komponenter.
- 3) Hvad er en container?
- 4) Giv tre eksempler på containere.
- 5) Hvad er en layout-manager?
- 6) Giv tre eksempler på layout-managere.
- 7) Beskriv rollefordelingen mellem komponenter, containere og layout-managere.

## 11.8 Resumé

- Grafiske komponenter, såsom JButton, JLabel og JTextField, er objekter, der kan ses på skærmen. De har nogle egenskaber, som kan ændres fra udviklingsværktøjet eller med get- og set-metoder. De arver alle fra JComponent og har derfor de fælles egenskaber *foreground*, *background* og *font*.
- Containere, såsom JPanel, JTabbedPane, JFrame, JDialog og JApplet, kan indeholde grafiske komponenter.
- Containere har en layout-manager tilknyttet, der bestemmer, hvordan komponenterne skal placeres indbyrdes og hvor meget plads de skal have.
- Et godt sted at læse mere er på: <http://java.sun.com/docs/books/tutorial/uiswing/>



## 11.9 Avanceret

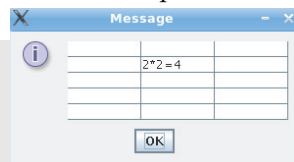
Der er flere komponenter end dem nævnt ovenfor. Vigtigst er, `JTable`, en regneark-lignende komponent til at vise og manipulere tabeldata og `JTree`, der viser en træstruktur ligesom stifinderen i Windows. Disse er dog lidt sværere at anvende, da de kræver et specielt data-model-objekt, der kan fortælle dem, hvilke data de skal vise:

```
import javax.swing.*;
import javax.swing.table.*;

public class BenytJTable {
    public static void main(String[] arg) {
        DefaultTableModel tm = new DefaultTableModel(5,3);
        tm.setValueAt("2*2=4",2,2);
        JTable tabel = new JTable(tm);
        JOptionPane.showMessageDialog(null,tabel);

        tabel.setModel(new DenLilleTabel());
        JOptionPane.showMessageDialog(null,tabel);
    }
}

class DenLilleTabel extends AbstractTableModel
{
    public int getColumnCount() { return 10; }
    public int getRowCount() { return 10; }
    public Object getValueAt(int r,int k) { return r+1+"*"+(k+1)+"="+ (r+1)*(k+1); }
}
```



1*1=1	1*2=2	1*3=3	1*4=4	1*5=5
1*2=2	2*2=4	2*3=6	2*4=8	2*5=10
1*3=3	3*2=6	3*3=9	3*4=12	3*5=15
1*4=4	4*2=8	4*3=12	4*4=16	4*5=20
1*5=5	5*2=10	5*3=15	5*4=20	5*5=25
1*6=6	6*2=12	6*3=18	6*4=24	6*5=30
1*7=7	7*2=14	7*3=21	7*4=28	7*5=35
1*8=8	8*2=16	8*3=24	8*4=32	8*5=40
1*9=9	9*2=18	9*3=27	9*4=36	9*5=45
1*10=10	10*2=20	10*3=30	10*4=40	10*5=50

Det er nemmest at bruge en `DefaultTableModel` som datamodel-objekt, men for fuld udnyttelse må man lave sit eget. Det viser klassen `DenLilleTabel`, der har 10 rækker og 10 kolonner, og hver celle har værdi som (rækkenummer+1) gange (kolonnennummer+1).

### 11.9.1 HTML-kode i komponenter

Alle Swing-komponenter understøtter visning af HTML-kode, så man kan f.eks. få knapper og etiketter, hvor dele af teksten er fed, kursiv eller anderledes størrelse eller farve.

Det gøres ved, i komponenternes `setText()`, at starte med "<html>":

```
import javax.swing.*;
public class BenytOverblikOverKomponenterMedHTML {
    public static void main(String[] arg) {
        OverblikOverKomponenter panel = new OverblikOverKomponenter();
        JFrame vindue = new JFrame( "OverblikOverKomponenterHTML" );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        panel.label1.setText("<html>En <font color=\"red\">Rød</font> Label<br>"
            +"på flere linjer");
        panel.button1.setText("<html>OK<br><font size=\"1\">(måske)</font>");

        panel.comboBox1.removeAllItems();
        panel.comboBox1.addItem("<html>ComboBox <font color=\"red\">Rød</font>");
        panel.comboBox1.addItem("<html>ComboBox <font color=\"green\">Grøn</font>");
        panel.comboBox1.addItem("<html>ComboBox <font color=\"blue\">Blå</font>");

        panel.checkbox1.setText("<html><font size=\"7\">En</font>");
        panel.checkbox2.setText("<html><i>To</i>");
        panel.checkbox3.setText("<html><b>Tre</b>");

        vindue.add( panel );
        vindue.setSize(500,150);
        vindue.setVisible(true);
    }
}
```



## 11.9.2 Flertrådet komponentprogrammering

Grafiksystemet tager *ikke* højde for flertrådede problematikker a la dem nævnt i afsnit 17.4.2. Grafiksystemets hændelsestråd, d.v.s. den tråd systemet kalder `paintComponent()` og metoder på hændelseslyttere med (eng.: event dispatch thread), er derfor den *eneste* tråd, der må opdatere den grafiske brugergrænseflade (kald til `repaint()` undtaget).

Hvis du fra en anden tråd, f.eks. fra `main()`-metoden, ønsker at opdatere noget, skal du kalde `SwingUtilities.invokeLater()` eller `SwingUtilities.invokeAndWait()` med et `Runnable`-objekt. Objektets `run()`-metode vil så blive udført af grafiksystemets tråd.

```
import java.util.*;
import javax.swing.*;

public class FlertraadetBrugAfKomponenter {
    public static void main(String[] arg) throws Exception {
        final OverblikOverKomponenter panel = new OverblikOverKomponenter();
        JFrame vindue = new JFrame( "FlertraadetBrugAfKomponenter" );
        vindue.add( panel );
        vindue.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        vindue.setSize(350,300);
        vindue.setVisible(true);
        for (int t=0; t<100000; t++) {
            SwingUtilities.invokeLater(new Runnable() { // indre anonym klasse
                public void run() { // med metoden run()
                    panel.label1.setText("hej "+new Date());

                    if (Math.random()>0.95) panel.comboBox1.removeAllItems();
                    panel.comboBox1.addItem("hejsa "+new Date());
                    int antal = panel.comboBox1.getItemCount();
                    panel.comboBox1.setSelectedIndex( (int) (Math.random()*antal));
                } // slut på metoden run(), klasse og invokeLater()-kald
            }); try { Thread.sleep(500); } catch (Exception e) {} // Vent 1/2 sekund
        }
    }
}
```

**Øvelse:** Prøv at køre eksemplet og vælg forskellige ting fra valglisten. Kommentér så linjerne i fed væk og se de forskellige mærkelige effekter, der optræder, når du vælger noget.

## 11.9.3 Brug af komponenter vs. `paintComponent()`

Som vi har set, er der to måder at arbejde med en grafisk brugergrænseflade på:

- I `paintComponent()` tegner man "i hånden" med kommandoer til `Graphics`-objektet.
- Man kan tilføje grafiske standardkomponenter til ens vindue.

Hvis du, som i `GrafikpanelMedKomponenter`, gør begge dele, så bemærk, at det er meget vigtigt at kalde superklassens `paintComponent()`-metode.

---

**Husk altid at kalde `super.paintComponent(g)` fra `paintComponent()`, ellers vil den kode, der stod i den oprindelige `paintComponent()` ikke blive udført**

---

**Øvelse:** Fjern kaldet `super.paintComponent(g)` fra `GrafikpanelMedKomponenter` og kørs eksemplet `BenytFaneblade` fra afsnit 11.4.6. Hvad sker der?

Flyt kaldet til `super.paintComponent(g)` ned tilsidst i `paintComponent()`. Hvorfor kan grafikken ikke ses mere? Overvej hvorfor rækkefølgen af tegnekommandoerne er vigtig.

Sæt et kald til `setOpaque(false)` ind i konstruktøren for at gøre panelet 'gennemsigtigt' (den tegner kun sine komponenter uden først at rense baggrunden) og prøv igen.

Lav en nedarving af `JButton`, kald den `KrydsKnap`, der tegner et fedt kryds henover sig selv, og prøv at bruge den. Eksperimentér med `paintComponent()` i den på samme måde.

# 12 Interfaces - grænseflader til objekter

Indhold:

- Forstå og definere interfaces
- Polymorfi og anvendelse af interfaces
- Standardbibliotekets brug af interfaces

Forudsættes af kapitel 13, Hændelser, kapitel 17, Flertrådet programmering, kapitel 18, Serialisering, kapitel 19, RMI og kapitel 21, Avancerede klasser.

Forudsætter kapitel 5, Nedarvning.

I et eksempel anvendes appletter, beskrevet i kapitel 10, Appletter.

I generel sprogbrug er et interface (da.: snitflade) en form for grænseflade, som man gør noget gennem. F.eks. er en grafisk brugergrænseflade de vinduer med knapper, indtastningsfelter og kontroller, som brugeren har til interaktion med programmet.

Vi minder om, at en klasse er definitionen af en type objekter. Her kunne man opdele i

- 1) *Grænsefladen* – hvordan objekterne kan bruges udefra.  
Dette udgøres af navnene<sup>1</sup> på metoderne, der kan ses udefra.
- 2) *Implementationen* – hvordan objekterne virker indeni.  
Dette udgøres af variabler og programkoden i metodekroppene.

Et 'interface' svarer til punkt 1): En definition af, hvordan objekter bruges udefra. Man kan sige, at et interface er en "halv" klasse.

---

### Et interface er en samling navne på metoder (uden krop)

---

Et interface kan implementeres af en klasse – det vil sige, at klassen definerer alle interface's metoder sammen med programkoden, der beskriver, hvad der skal ske, når metoderne kaldes.

## 12.1 Definere et interface

Lad os definere et interface kaldet Tegnbar, der beskriver noget, der kan tegnes.

```
import java.awt.*;

public interface Tegnbar
{
    public void sætPosition(int x, int y);
    public void tegn(Graphics g);
}
```

I stedet for "class" erklæres et interface med "interface".

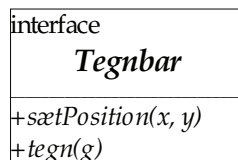
Metoder i et interface har ingen krop, alle metodeerklæringerne følges af et semikolon.

Der kan ikke oprettes objekter ud fra et interface. Det kan opfattes som en "tom skal", der skal "fyldes ud" af en rigtig klasse, der implementerer metoderne (definerer kroppene).

Man ser, at tegnbare objekter:

- har en metode til at sætte positionen på skærmen
- har en metode til at tegne objektet.

I UML-notation (tegningen til højre) er Tegnbar-interfacet tegnet med kursiv. Alle metoderne er abstrakte (= ikke implementerede) og er derfor også tegnet kursivt.



## 12.2 Implementere et interface

Lad os nu definere en klasse, der implementerer Tegnbar-interfacet.

---

**En klasse kan erklære, at den *implementerer et interface* og så skal den definere alle metoderne i interface'et og give dem en metodekrop**

---

Vi skal altså definere alle interface's metoder sammen med programkoden, der beskriver, hvad der skal ske, når metoderne kaldes.

---

<sup>1</sup> Egentlig signaturen, dvs. metodenavn og antal og type af parametre.

```
import java.awt.*;
public class Stjerne implements Tegnbar
{
    private int posX, posY;

    public void sætPosition(int x, int y)    // kræves af interfacet Tegnbar
    {
        posX = x;
        posY = y;
    }

    public void tegn(Graphics g)            // kræves af interfacet Tegnbar
    {
        g.drawString("*",posX,posY);
    }
}
```

Her har klassen Stjerne "udfyldt skallen" for Tegnbar ved at skrive "implements Tegnbar" og definere sætPosition()- og tegn()-metoderne (vi har også variabler til at huske x og y).

## 12.2.1 Variabler af type Tegnbar

Man kan erklære variabler af en interface-type. Disse kan referere til alle slags objekter, der implementerer interfacet<sup>1</sup>. Herunder erklærer vi en variabel af type Tegnbar og sætter den til at referere til et Stjerne-objekt.

```
Tegnbar t;
t = new Stjerne();    // Lovligt, Stjerne implementerer Tegnbar
```

Stjerne-objekter er også af type Tegnbar. Ligesom ved nedarvning siger man, at der er relationen Stjerne er-en Tegnbar og at t er polymorf, da den kan referere til alle slags Tegnbar-objekter.

Man kan ikke oprette objekter ud fra et interface (der er en "skal" og intet siger om, hvordan metoderne er implementerede – hvordan skulle objektet reagere, hvis de blev kaldt?).

```
t = new Tegnbar();    // FEJL! Tegnbar er ikke en klasse
```

## 12.2.2 Eksempler med interfacet Tegnbar

Lad os udvide (arve fra) Terning og få den til at implementere Tegnbar-interfacet:

```
import java.awt.*;
public class GrafiskTerning extends Terning implements Tegnbar
{
    int x, y;

    public void sætPosition(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    private void ci(Graphics g, int i, int j)
    {
        g.fillOval(x+10*i,y+10*j,8,8);    // Tegn fyldt cirkel
    }

    public void tegn(Graphics g)
    {
        int ø = værdi;
        g.drawRect(x,y,30,30);    // Tegn kant

        if (ø==1) ci(g,1,1);    // Tegn 1-6 øjne
        else if (ø==2) { ci(g,0,0); ci(g,2,2); }
        else if (ø==3) { ci(g,0,0); ci(g,1,1); ci(g,2,2); }
        else if (ø==4) { ci(g,0,0); ci(g,0,2); ci(g,2,0); ci(g,2,2); }
        else if (ø==5) { ci(g,0,0); ci(g,0,2); ci(g,2,0); ci(g,2,2); ci(g,1,1); }
        else {ci(g,0,0); ci(g,0,1); ci(g,0,2); ci(g,2,0); ci(g,2,1); ci(g,2,2); }
    }
}
```

<sup>1</sup> Det vil sige alle objekter, hvis klasse implementerer interfacet.

For at gøre koden kort har tegn() en hjælpemetode ci(), der tegner en cirkel for et øje.

Bemærk:

- Man kan godt have flere metoder end specificeret i interfacet (i dette tilfælde ci()).
- GrafiskTerning er en Tegnbar og samtidig en Terning. Der kan kun arves fra én klasse, men samtidigt kan der godt implementeres et interface (faktisk også flere).

Lad os gøre det samme med et raflebæger. For variationens skyld lader vi bægeret altid have den samme position ved at lade sætPosition()'s krop være tom.

```
import java.awt.*;
public class GrafiskRaflebaeger extends Raflebaeger implements Tegnbar
{
    public GrafiskRaflebaeger()
    {
        super(0);
    }

    public void sætPosition(int x, int y)
    {
        // tom metodekrop
    }

    public void tegn(Graphics g)
    {
        g.drawOval(80,20,90,54);
        g.drawLine(150,115,170,50);
        g.drawLine(100,115,80,50);
        g.drawArc(100,100,50,30,180,180);
    }
}
```

Kunne vi have udeladt sætPosition()-metoden, der alligevel ikke gør noget? Nej, vi har lovet at implementere begge metoder, om det så blot er med en tom krop, idet vi skrev "implements Tegnbar".

*En hvilken som helst klasse kan gøres til at være Tegnbar.* Herunder udvider vi standardklassen Rectangle til at være Tegnbar:

```
import java.awt.*;
public class Rektangel extends Rectangle implements Tegnbar
{
    public Rektangel(int x1, int y1, int width1, int height1)
    {
        super(y1,x1,width1,height1);
    }

    public void sætPosition(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    public void tegn(Graphics g)
    {
        g.drawRect(x,y,width,height);
    }
}
```

## 12.2.3 Visning af nogle Tegnbare objekter

Lad os nu lave et vindue, der viser nogle tegnbare objekter:

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

public class TegnbareObjekter extends JPanel
{
    ArrayList<Tegnbar> tegnbare = new ArrayList<Tegnbar>();
    GrafiskRaflebaeger baege = new GrafiskRaflebaeger();

    public void sætPositioner()
    {
        for (Tegnbar t : tegnbare) {
            int x = (int) (Math.random()*200);
            int y = (int) (Math.random()*200);
            t.sætPosition(x,y);
        }
    }

    public TegnbareObjekter()
    {
        Stjerne s = new Stjerne();
        tegnbare.add(s);

        tegnbare.add( new Rektangel(10,10,30,30) );
        tegnbare.add( new Rektangel(15,15,20,20) );

        GrafiskTerning t;
        t = new GrafiskTerning();
        baege.tilføj(t);
        tegnbare.add(t);

        t = new GrafiskTerning();
        baege.tilføj(t);
        tegnbare.add(t);

        tegnbare.add(baege);

        sætPositioner();
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);

        sætPositioner();

        for (Tegnbar t : tegnbare) t.tegn(g);
    }
}
```

```
import javax.swing.*;
public class BenytTegnbareObjekter
{
    public static void main(String[] arg)
    {
        JFrame vindue = new JFrame( "TegnbareObjekter" );
        vindue.add( new TegnbareObjekter() );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        vindue.setSize(300,300);
        vindue.setVisible(true);
        while (true) {
            try { Thread.sleep(3000); } catch (Exception e) {}// vent 3 sek.
            vindue.repaint(); // gentegn skærm
        }
    }
}
```

Programmet holder styr på objekterne i tegnbare-listen. Da stjerner, rektangler, terningerne og raflebægeret alle er Tegnbare, kan de behandles ens, hvad angår tegning og positionering.

## 12.2.4 Polymorfi

Det er meget kraftfuldt, at man kan erklære variabler af en interface-type. Disse kan referere til alle mulige slags objekter, der implementerer interfacet. Herefter kan vi f.eks. løbe en liste igennem og arbejde på objekterne i den, selvom de er af vidt forskellig type.

Dette så vi i `paintComponent()`-metoden i `TegnbareObjekter`-klassen:

```
for (Tegnbar t : tegnbare)
{
    t.tegn(g);
}
```

Et interface som `Tegnbar` kan bruges til at etablere en fællesnævner mellem vidt forskellige objekter, som derefter kan behandles ens. Dette kaldes polymorfi (græsk: "mange former").

Fællesnævneren – nemlig at de alle implementerer det samme interface – tillader os at arbejde med objekter *uden at kende deres præcise type*. Dette kan i mange tilfælde være en fordel, når vi arbejder med objekter, hvor vi ikke kender (eller ikke interesserer os for) den eksakte type.

## 12.3 Interfaces i standardbibliotekerne

Interfaces bliver brugt i vid udstrækning i standardbibliotekerne og mange steder benyttes polymorfi til at gøre det muligt at lade systemet arbejde på programmørens egne klasser.

I det følgende vil vi se nogle eksempler på, at implementationen af et interface fra standardbiblioteket gør, at vores klasser passer ind i systemets klasser på forskellig måde.

### 12.3.1 Sortering med en Comparator

Hvis et objekt implementerer `Comparator`-interfacet, skal det definere metoden:

```
public int compare(Object obj1, Object obj2)
```

Metoden skal sammenligne `obj1` og `obj2` og afgøre, om objektet, som `obj1` peger på, kommer før objektet, som `obj2` peger på, eller omvendt.

For eksempel kunne vi ved at implementere `Comparator<Terning>` lave en klasse, der sammenligner `Terning`-objekter ud fra antallet af øjne, de viser:

```
import java.util.*;
public class TerningComparator implements Comparator<Terning>
{
    public int compare(Terning t1, Terning t2) // kræves af Comparator
    {
        if (t1.værdi == t2.værdi) return 0; // t1 og t2 skal på samme plads i listen
        if (t1.værdi > t2.værdi) return 1; // t1 skal efter t2
        else return -1; // t1 skal før t2
    }
}
```

En `Comparator` giver standardbiblioteket mulighed for at sammenligne nogle objekter og sortere dem i forhold til hinanden. Sortering kan bl.a. ske ved at kalde metoden `Collections.sort()` med en liste af objekter og en `Comparator`:

```
import java.util.*;
public class BenytTerningComparator
{
    public static void main(String[] arg)
    {
        ArrayList<Terning> liste = new ArrayList<Terning>();
        liste.add( new Terning());
        liste.add( new Terning());
        liste.add( new Terning());
        liste.add( new Terning());
        liste.add( new Terning());

        System.out.println("før sortering: "+liste);
        TerningComparator sammenligner = new TerningComparator();
```

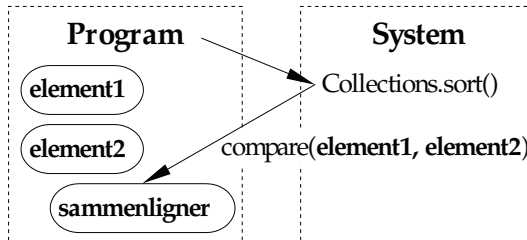


```

Collections.sort(liste, sammenligner );
System.out.println("efter sortering: "+liste);
}
}
før sortering: [6, 3, 4, 2, 4]
efter sortering: [2, 3, 4, 4, 6]

```

Metoden sort() vil løbe listen igennem og sammenligne elementerne ved kalde compare() på Comparator-objektet for at sortere dem i rækkefølge.



*Systemkaldet Collections.sort() sorterer en liste af elementer ved hjælp af en sammenligner (Comparator).*

*Systemet finder rækkefølgen af elementerne ved at kalde compare() på sammenligneren til at afgøre om de enkelte elementer kommer før eller efter hinanden.*

## 12.3.2 Flere tråde med Runnable

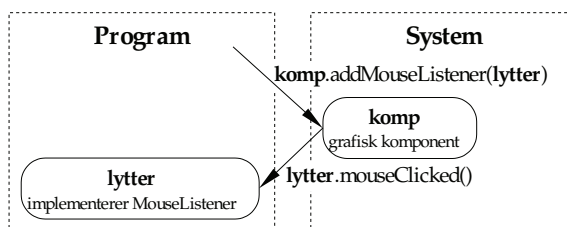
Hvis man vil bruge flere tråde (processer, der kører samtidigt i baggrunden) i sit program, kan dette opnås ved at implementere interfacet Runnable og definere metoden run(). Derefter opretter man et tråd-objekt med new Thread(objektDerImplementererRunnable). Når tråden startes (med trådobjekt.start()), vil det begynde en parallel udførelse af run()-metoden i objektDerImplementererRunnable.

Dette vil blive behandlet i kapitel 17, Flertrådet programmering.

## 12.3.3 Lytte til musen med MouseListener

Når man programmerer grafiske brugergrænseflader, kan det være nyttigt at kunne få at vide, når der er sket en hændelse, f.eks. at musen er klikket et sted.

Dette sker ved, at man definerer et objekt (lytteren), der implementerer MouseListener-interfacet. Den har forskellige metoder, f.eks. mouseClicked(), der er beregnet på et museklik.



*Programmet registrerer en lytter på en grafisk komponent (komp) og systemet kalder derpå metoder i lytteren, når bestemte hændelser sker.*

Lytteren skal registreres i en grafisk komponent, f.eks. en knap eller et vindue. Det gøres ved at kalde komponentens addMouseListener()-metode med en reference til lytteren. Derefter vil, hver gang brugeren klikker på komponenten, lytterens mouseClicked() blive kaldt.

Analogt findes lyttere til tastatur, musebevægelser, tekstfelter, kontroller osv. I kapitel 13 om grafiske brugergrænseflader og hændelser er disse ting beskrevet nærmere.

## 12.4 Test dig selv

- 1) Hvad er et interface?
- 2) Giv et eksempel.
- 3) Hvis en klasse erklærer, at den implementerer et interface, hvad skal den så indeholde?
- 4) Hvis flere klasser implementerer samme interface, hvad kan man så gøre?
- 5) Hvad er polymorfi?
- 6) For at etablere en fællesnævner mellem nogle objekter kunne man også benytte ned-arvning. Hvorfor skulle man bruge et interface i stedet for at arve fra en klasse?

## 12.5 Resumé

- 1) Et interface er en skal, der indeholder navne og parametre på nogle metoder, men uden krop.
- 2) Se f.eks. Tegnbar-interfacet.
- 3) En klasse, der implementerer et interface, skal indeholde alle metoderne fra interfacet med metodekroppe.
- 4) Flere klasser, der implementerer samme interface, f.eks. Tegnbar, har interfacet som fællesnævner. Objekter fra disse klasser kan behandles ensartet, f.eks. med en variabel af typen Tegnbar.
- 5) Det kaldes polymorfi, fordi objekterne kan have "mange former" (variabler, metoder,..).
- 6) Det kan være, at man ikke kan arve fra en fælles klasse (fordi der arves fra noget andet), eller lighederne er for små til, at man ønsker det.

## 12.6 Opgaver

- 1) Lav klassen Hus, der skal implementere Tegnbar. Føj den til TegnbareObjekter og prøv, om det virker.
- 2) Prøv at tilføje et ikke-Tegnbar't objekt (f.eks. en streng eller et Point-objekt) til tegnbare-listen. Hvad sker der så? Hvilken fejlmeddelelse kommer der?
- 3) Lav tre implementationer af Comparator, der sorterer strenge hhv. alfabetisk, omvendt alfabetisk og alfabetisk efter andet tegn i strengene. Lav en liste (ArrayList) med ti strenge og test din sortering med Collections.sort(liste, Comparator-objekt).
- 4) Kig på matador-spillet afsnit 5.3. Ændr Felt til at være et interface.

# 12.7 Avanceret

## 12.7.1 Collections - ArrayList's familie

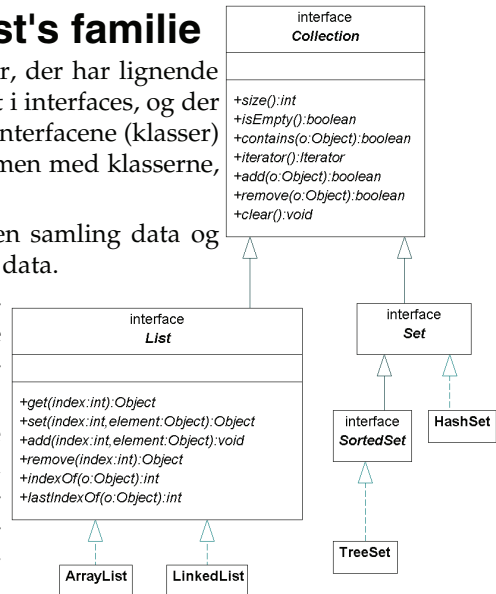
ArrayList har faktisk en stor 'familie' af klasser, der har lignende funktioner. Alle deres fælles metoder beskrevet i interfaces, og der findes så flere forskellige implementationer af interfacene (klasser) til forskelligt brug. De er vist her til højre sammen med klasserne, der implementerer dem.

Interfacet Collection (da.: samling) beskriver en samling data og indeholder metoder fælles for alle samlinger af data.

Der er ikke nogen klasser, der direkte implementerer Collection. I stedet implementerer de interfacene List, Set eller SortedSet, der udbygger (arver fra) Collection.

List beskriver en *ordnet* liste, dvs. elementerne har en rækkefølge og hvert element har en plads, ligesom med ArrayList (der da også implementer List). LinkedList er en anden implementation, der bruger såkaldte dobbelthægtede lister internt i stedet for et array.

Set er en *uordnet* mængde. I en mængde kan der højst være én af hvert element, og rækkefølgen, som elementerne blev sat ind i, huskes ikke. Klasserne HashSet og TreeSet implementerer den (TreeSet sorterer automatisk elementerne).



## 12.7.2 Implementere flere interfaces

En klasse kan godt implementere flere interfaces. Har vi f.eks.:

```
public interface I
{
    public void m();
}
```

og

```
public interface I2
{
    public void m2();
}
```

kan vi godt implementere både I og I2:

```
public class D implements I, I2
{
    public void m()
    {
        System.out.println("m() blev kaldt i D-objekt");
    }

    public void m2()
    {
        System.out.println("m2() blev kaldt i D-objekt");
    }
}
```

Der er altså ikke nogen specielle problemer med at implementere flere interfaces.

Selv hvis to interfaces skulle kræve nogle af de samme metoder defineret (f.eks. både I og I2 krævede m() defineret), vil klassen også godt kunne opfylde dette.

## 12.7.3 At udbygge interfaces

Et interface kan udbygges, f.eks.:

```
public interface I2a extends I2
{
    public void m2a();
}
```

En klasse, der implementerer I2a, skal have defineret begge metoderne m2() og m2a() og vil derfor automatisk også implementere I2.

## 12.7.4 Multipel arv

Multipel arv betyder, at en klasse kan nedarve fra flere andre klasser. Man kunne f.eks. forestille sig at lave en fælles nedarving fra en LudoTerning og FalskTerning, der kunne hedde FalskLudoTerning.

Der følger en række problemer med denne mulighed og det kan blive meget uoverskueligt at programmere. F.eks. i FalskLudoTerning – har klassen en eller to værdier for antallet af øjne? Afhænger det af, om LudoTerning og FalskTerning har en (eller flere!) fælles super-klasser? (og hvad så, hvis de har nogle fælles, men også nogle ikke-fælles superklasser?).

Java understøtter ikke multipel arv og det er en af grundene til, at Java anses for simpleere at programmere end f.eks. C++, som *har* multipel arv. I C++ bruges multipel arv mest som middel til at opnå en fælles grænseflade til forskellige slags objekter, så man kan arbejde med dem ensartet – polymorfi.

Interfaces afløser langt hen ad vejen behovet for multipel arv – uden de problemer, multipel arv kan medføre, da variabler og programkode ikke føres med.

## 12.7.5 Variabler i et interface

Ud over metoder indeholder et interface en gang imellem også variabler. Disse bliver automatisk erklæret `public static final`<sup>1</sup>, det vil sige, at de altid er fuldt tilgængelige klassevariabler, der ikke kan ændres.

Det kan bruges til at erklære konstanter, der skal kunne benyttes i flere klasser, f.eks.:

```
public interface Tegnekonstanter
{
    double LÆNGDE = 10;
    double BREDDE = 10;
}
```

De er tilgængelige fra andre klasser med `Tegnekonstanter.LÆNGDE` og `Tegnekonstanter.BREDDE` fra andre klasser.

## 12.7.6 Statisk import

Klasser, der implementerer `Tegnekonstanter`, ville automatisk have defineret konstanterne `LÆNGDE` og `BREDDE`, men det frarådes at misbruge 'implements' på den måde. I stedet er det muligt at importere statiske variabler/konstanter (og endda metoder) med:

```
import static Tegnekonstanter.* // importer alt statisk fra Tegnekonstanter
// herefter er LÆNGDE og BREDDE defineret
```

Dette kaldes 'statisk import' og bør bruges sparsomt. Det er ikke begrænset til interfaces:

```
import static java.lang.Math.*;
// herefter kan man skrive PI, sin(), sqrt() etc uden 'Math' foran.
```

---

<sup>1</sup> `public`=tilgængelig for alle, `static`=klassevariabel, `final`=konstant; umulig at ændre.

# 13 Hændelser i grafiske brugergrænseflader

Indhold:

- Forstå hændelser og lyttere
- Abonnere på hændelser

Forudsættes af kapitel 21, Avancerede klasser.

Forudsætter kapitel 11, Grafiske standardkomponenter og 12, Interfaces.

I eksemplerne anvendes appletter, beskrevet i kapitel 10, Appletter.

Hændelser (eng.: events) spiller en stor rolle i programmering af grafiske brugergrænseflader. Når brugeren foretager en handling, f.eks. bevæger musen, klikker, trykker en knap ned, ændrer i et tekstfelt osv., opstår der en *hændelse*. I Java er alle hændelser objekter (af typen Event) med metoder til at undersøge de præcise detaljer omkring hændelsen.

Hændelser udsendes af de grafiske komponenter (knapper, vinduer osv.) og hvis man vil behandle en bestemt type hændelser fra en bestemt grafisk komponent, skal man "lytte" efter den hændelse. Det gøres ved at registrere en *lytter* (eng.: listener) på komponenten.

Når en lytter til en bestemt slags hændelser er registreret hos en komponent, bliver der kaldt en metode på lytteren, når den pågældende slags hændelser indtræffer (f.eks. kaldes `mouseClicked()`, når der klikkes med musen). For at sikre, at lytteren har den pågældende metode, skal lytter-objektet implementere et interface, der garanterer, at det har metoden.

Eksempel: Paneler (JPanel) kan udsende hændelser af typen `MouseEvent`. Klassen `JPanel` har derfor metoden `addMouseListener(MouseListener lytter)`, der kan bruges til at registrere lytter-objekter i vinduet. Det er kun objekter af typen `MouseListener`, der kan registreres som lyttere. `MouseListener` er et interface, så man skal lave en klasse, der implementerer `MouseListener` og skabe lytter-objekter ud fra dette. Når brugeren f.eks. klikker med musen i vinduet, udsender det en `MouseEvent`-hændelse til alle lytter-objekter, der er blevet registreret vha. `addMouseListener()`. Det gør vinduet ved at kalde metoden `mouseClicked(MouseEvent hændelse)` på lytter-objekterne.

## 13.1 Eksempel: LytTilMusen

Herunder definerer vi klassen `Muselytter`, der implementerer `MouseListener` og skriver ud til skærmen, hver gang der sker noget med musen.

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

public class Muselytter implements MouseListener
{
    public void mousePressed(MouseEvent hændelse)    // kræves af MouseListener
    {
        Point trykpunkt = hændelse.getPoint();
        System.out.println("Mus trykket ned i "+trykpunkt);
    }

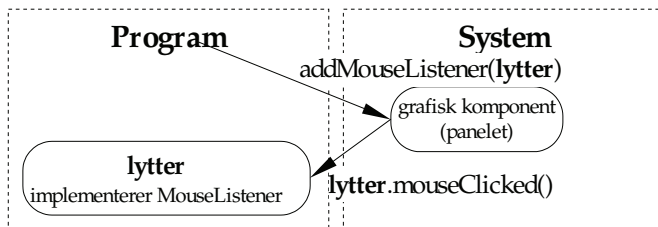
    public void mouseReleased(MouseEvent hændelse)   // kræves af MouseListener
    {
        Point slippunkt = hændelse.getPoint();
        System.out.println("Mus sluppet i "+slippunkt);
    }

    public void mouseClicked(MouseEvent hændelse)   // kræves af MouseListener
    {
        System.out.println("Mus klikket i "+hændelse.getPoint());
    }

    //-----
    // Ubrugte hændelser (skal defineres for at implementere MouseListener)
    //-----
    public void mouseEntered (MouseEvent event) {} // kræves af MouseListener
    public void mouseExited (MouseEvent event) {} // kræves af MouseListener
}
```

Lad os nu lave et grafisk objekt, der:

- 1) Opretter et `muselytter`-objekt.
- 2) Registrerer lytter-objektet, så det får kaldt sine metoder, når der sker noget med musen.



Programmet registrerer en lytter på en grafisk komponent (panelet) ved at kalde `addMouseListener()` på den. Derefter, når hændelse 'klik med musen' sker, kalder systemet `mouseClicked()` på lytteren,

```
import javax.swing.*;
public class LytTilMusen extends JPanel
{
    public LytTilMusen()
    {
        Muselytter lytter = new Muselytter();
        this.addMouseListener(lytter); // this er panelet selv
    }
}
```

Her er et eksempel på uddata fra programmet:

```
Mus trykket ned i java.awt.Point[x=132,y=209]
Mus sluppet i java.awt.Point[x=139,y=251]
Mus trykket ned i java.awt.Point[x=101,y=199]
Mus sluppet i java.awt.Point[x=101,y=199]
Mus klikket i java.awt.Point[x=101,y=199]
```

## 13.2 Eksempel: Linjetegning

Det foregående eksempel giver ikke panelet besked om, at der er sket en hændelse. Det har man brug for, hvis man f.eks. vil tegne noget på panelet.

Herunder er et eksempel, hvor lytter-objektet (Linjelytter) giver informationer om klik videre til panelet (Linjetegning), sådan at en blå linje tegnes mellem det punkt, hvor man trykkede museknappen ind og det punkt, hvor man slap museknappen.

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

public class Linjetegning extends JPanel
{
    public Point trykpunkt;
    public Point slippunkt;

    public Linjetegning()
    {
        Linjelytter lytter = new Linjelytter();
        lytter.panelet = this; // initialiserer lytterens reference til panelet
        this.addMouseListener(lytter);
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g); // tegn først baggrunden på panelet
        g.drawString("1:"+trykpunkt+" 2:"+slippunkt,10,10);
        if (trykpunkt != null && slippunkt != null)
        {
            g.setColor(Color.BLUE);
            g.drawLine(trykpunkt.x, trykpunkt.y, slippunkt.x, slippunkt.y);
        }
    }
}
```

Lytteren skal give panelet besked om klik vha. panelets to variabler, trykpunkt og slippunkt. Derfor er Linjelytter nødt til at have en reference (af type Linjetegning) til panelet:

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

public class Linjelytter implements MouseListener
{
    public Linjetegning panelet;           // Reference til panelet

    public void mousePressed(MouseEvent hændelse) // kræves af MouseListener
    {
        panelet.trykpunkt = hændelse.getPoint();
    }

    public void mouseReleased(MouseEvent hændelse) // kræves af MouseListener
    {
        panelet.slippunkt = hændelse.getPoint();
        panelet.repaint(); // Gentegn panelet lige om lidt.
    }

    //-----
    // Ubrugte hændelser (skal defineres for at implementere interfacet)
    //-----
    public void mouseClicked(MouseEvent event) {} // kræves af MouseListener
    public void mouseEntered (MouseEvent event) {} // kræves af MouseListener
    public void mouseExited (MouseEvent event) {} // kræves af MouseListener
}
```

Med linjen

```
panelet.repaint();
```

fortæller vi Linjetegning-panelet, at den skal gentegne sig selv. Det forårsager kort efter et kald til dens paintComponent()-metode.



Her er en klasse der viser dette og alle de andre paneler i dette kapitel, i hvert sit faneblad (beskrevet i afsnit 11.4.6).

```
import javax.swing.*;
public class BenytAltKapitel13
{
    public static void main(String[] arg)
    {
        JTabbedPane faneblade = new JTabbedPane();
        faneblade.add("1 LytTilMusen", new LytTilMusen());
        faneblade.add("2 Linjetegning", new Linjetegning());
        faneblade.add("3 Linjetegning2", new Linjetegning2());
        faneblade.add("4 Kruseduller", new Kruseduller());
        faneblade.add("5 LytTilKnap", new LytTilKnap());
        faneblade.add("6 Tastetryk", new Tastetryk());
        JFrame vindue = new JFrame("BenytAltKapitel13");
        vindue.add( faneblade );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // reager på luk
        vindue.pack(); // lad vinduet selv bestemme sin størrelse
        vindue.setVisible(true); // åbn vinduet
    }
}
```



## 13.2.1 Linjetegning i én klasse

Herunder er Linjetegning igen, men nu som et panel, der *selv* implementerer MouseListener. Det er linjen:

```
this.addMouseListener(this);
```

der registrerer panel-objektet selv som lytter.

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

public class Linjetegning2 extends JPanel implements MouseListener
{
    private Point trykpunkt;
    private Point slippunkt;

    public Linjetegning2()
    {
        this.addMouseListener(this);
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g); // tegn først baggrunden på panelet
        g.drawString("1:" + trykpunkt + " 2:" + slippunkt, 10, 10);
        if (trykpunkt != null && slippunkt != null)
        {
            g.setColor(Color.BLUE);
            g.drawLine(trykpunkt.x, trykpunkt.y, slippunkt.x, slippunkt.y);
        }
    }

    public void mousePressed(MouseEvent hændelse) // kræves af MouseListener
    {
        trykpunkt = hændelse.getPoint();
    }

    public void mouseReleased(MouseEvent hændelse) // kræves af MouseListener
    {
        slippunkt = hændelse.getPoint();
        repaint();
    }

    //-----
    // Ubrugte hændelser (skal defineres for at implementere interfacet)
    //-----
    public void mouseClicked(MouseEvent event) {} // kræves af MouseListener
    public void mouseEntered (MouseEvent event) {} // kræves af MouseListener
    public void mouseExited (MouseEvent event) {} // kræves af MouseListener
}
```

Bemærk, at nu kan trykpunkt og slippunkt være private i stedet for public, fordi de ikke behøver at være tilgængelige udefra.

## 13.3 Ekstra eksempler

Ovenfor har vi brugt MouseListener som illustration. Her vil vi give eksempler på brug af de andre typer lyttere (beskrevet i appendiks senere i kapitlet).

### 13.3.1 Lytte til musebevægelser

Med MouseMotionListener får man adgang til hændelserne mouseMoved og mouseDragged. Det kan bruges til at tegne grafiske figurer ved at hive musen hen over skærmen.

Herunder er et panel, man kan tegne kruseduller på. Vi husker punktet, når musen trykkes ned (mousePressed()) og tegner en linje fra forrige punkt til musen, når den trækkes med nedtrykket knap (mouseDragged()).



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

public class Kruseduller extends JPanel
    implements MouseListener, MouseMotionListener
{
    public Kruseduller()
    {
        this.addMouseListener(this);
        this.addMouseMotionListener(this);
    }

    Point punkt;

    public void mousePressed(MouseEvent hændelse) // kræves af MouseListener
    {
        punkt = hændelse.getPoint();
    }

    public void mouseDragged(MouseEvent hændelse) // kræves af MouseMotionListener
    {
        Point gammeltPunkt = punkt;
        punkt = hændelse.getPoint();
        Graphics g = getGraphics();
        g.drawLine(gammeltPunkt.x, gammeltPunkt.y, punkt.x, punkt.y);
    }

    public void mouseReleased (MouseEvent hændelse){} // kræves af MouseListener
    public void mouseClicked (MouseEvent event) {} // kræves af MouseListener
    public void mouseEntered (MouseEvent event) {} // kræves af MouseListener
    public void mouseExited (MouseEvent event) {} // kræves af MouseListener
    public void mouseMoved (MouseEvent hændelse){} //kræves af MouseMotionListener
}
```

Her sker tegningen af grafikken direkte i håndteringen af hændelsen. Da vi ikke husker de gamle punkter, kan vi ikke gentegne krusedullen, hvis systemet kalder `paintComponent()`.

### 13.3.2 Lytte til en knap

Det vigtigste interface til programmering af grafiske brugergrænseflader er `ActionListener` med metoden `actionPerformed()`. Den bruges bl.a. til at lytte til, om knapper bliver trykket på. Her er et eksempel, hvor den tekst, der er valgt med musen i et tekstområde, bliver kopieret til det andet tekstområde, når man trykker på knappen.



```

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

public class LytTilKnap extends JPanel implements ActionListener
{
    private JTextArea t1, t2;
    private JButton kopierKnap;

    public LytTilKnap()
    {
        setLayout(new FlowLayout());
        String s = "Her er en tekst.\nMarkér noget af\nden og tryk\nKopier...";
        t1 = new JTextArea(s, 5,15);
        add(t1);
        kopierKnap = new JButton("Kopier>>");
        kopierKnap.addActionListener(this);
        add(kopierKnap);
        t2 = new JTextArea( 5,15);
        t2.setEditable(false);
        add(t2);
    }

    public void actionPerformed(ActionEvent e)    // kræves af ActionListener
    {
        t2.setText(t1.getSelectedText() );
    }
}

```

Læg mærke til, at vi registrerer lytteren (som er panel-objektet selv) hos knappen.

### 13.3.3 Lytte efter tastetryk

Vores sidste eksempel er med KeyListener-interfacet, der tillader at lytte efter tastetryk.

Programmet herunder viser en tekst. Hver gang der tastes et bogstav, bliver det tilføjet teksten. Med piletasterne kan man rykke teksten op og ned. Retur-tasten sletter teksten.

```

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

public class Tastetryk extends JPanel implements KeyListener
{
    String tekst = "tast noget - pil op/ned rykker teksten ";
    Point pos = new Point(20,20);

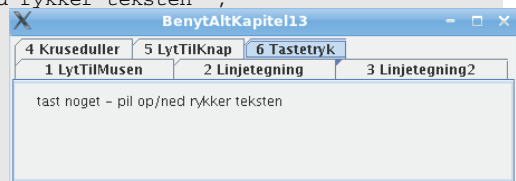
    public Tastetryk()
    {
        addKeyListener(this);
        requestFocus();
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);           // tegn først baggrunden på panelet
        g.drawString(tekst, pos.x, pos.y);
    }

    public void keyPressed(KeyEvent e)
    {
        if (e.getKeyCode() == KeyEvent.VK_ENTER) tekst = "tekst: ";
        else if (e.getKeyCode() == KeyEvent.VK_UP)  pos.y = pos.y - 10;
        else if (e.getKeyCode() == KeyEvent.VK_DOWN) pos.y = pos.y + 10;
        else tekst = tekst + e.getKeyChar();
        repaint();
    }

    public void keyReleased(KeyEvent e) {} // kræves af KeyListener
    public void keyTyped(KeyEvent e)   {} // kræves af KeyListener
}

```



# 13.4 Appendiks

## 13.4.1 Lyttere og deres metoder

Det følgende er en oversigt over lytter-interfaces og deres hændelser (de ligger alle i pakken `java.awt.event`).

### ActionListener

Hændelsen `ActionEvent` sendes af den pågældende komponent, når brugeren klikker på en knap, trykker retur i et tekstfelt, vælger noget i et afkrydsningsfelt, radioknap, menu eller lignende.

```
public interface ActionListener {  
    public void actionPerformed(ActionEvent e);  
}
```

### ComponentListener

Hændelsen `ComponentEvent` sendes af alle grafiske komponenter (`JButton`, `TextField`, `CheckBox` osv. og `JFrame`, `JApplet`, `JPanel`,...), når de hhv. ændrer størrelse, position, bliver synlige eller usynlige.

```
public interface ComponentListener {  
    public void componentResized(ComponentEvent e);  
    public void componentMoved(ComponentEvent e);  
    public void componentShown(ComponentEvent e);  
    public void componentHidden(ComponentEvent e);  
}
```

### FocusListener

Hændelsen `FocusEvent` sendes af komponenter, når de får fokus (dvs. hvis brugere trykker på en tast, vil det påvirke netop denne komponent). Kun en komponent har fokus ad gangen<sup>1</sup>.

```
public interface FocusListener {  
    public void focusGained(FocusEvent e);  
    public void focusLost(FocusEvent e);  
}
```

### ItemListener

Hændelsen `ItemEvent` sendes af afkrydsningsfelter og radioknapper, når en mulighed bliver krydset af eller fravalgt.

```
public interface ItemListener {  
    void itemStateChanged(ItemEvent e);  
}
```

### KeyListener

Hændelser af typen `KeyEvent` sendes af komponenten, der har tastaturfokus. `keyPressed()` kaldes, når en tast bliver trykket ned (bemærk, at der godt kan være flere taster trykket ned samtidig, f.eks. `Ctrl` og `C`) og `keyReleased()`, når den bliver sluppet. Er man mere overordnet interesseret i, hvad brugeren taster ind, bør man benytte `keyTyped()`, der svarer til, at brugeren har trykket en tast ned og sluppet den igen.

```
public interface KeyListener {  
    public void keyTyped(KeyEvent e);  
    public void keyPressed(KeyEvent e);  
    public void keyReleased(KeyEvent e);  
}
```

---

<sup>1</sup> Man kan anmode om fokus på en komponent ved at kalde `requestFocus()` på den.

## MouseListener

Hændelsen `MouseEvent` kan sendes af alle grafiske komponenter. `mousePressed()` kaldes, når en museknap bliver trykket ned og `mouseReleased()`, når den bliver sluppet igen. Er man mere overordnet interesseret i at vide, om brugeren har klikket et sted (trykket ned og sluppet på det samme sted), bør man benytte `mouseClicked()`. `mouseEntered()` og `mouseExited()` sendes, når musen går ind over hhv. væk fra komponenten.

```
public interface MouseListener {
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseClicked(MouseEvent e);

    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
}
```

## MouseMotionListener

Kan sendes af alle grafiske komponenter. `mouseDragged()` kaldes, når en museknap er trykket ned og hives (bevæges, mens museknappen forbliver trykket ned). `mouseMoved()` svarer til, at musen flyttes (uden nogle knapper trykket ned).

```
public interface MouseMotionListener {
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e);
}
```

## TextListener

Sendes af tekstfelter (`TextField` og `TextArea`), når brugeren ændrer teksten.

```
public interface TextListener {
    public void textValueChanged(TextEvent e);
}
```

## WindowListener

Hændelsen `WindowEvent` sendes af vinduer (`Frame` og `Dialog`), når de åbnes, forsøges lukket, lukkes, minimeres, gendannes, får fokus og mister fokus.

```
public interface WindowListener {
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}
```

Det er dette interface, der skal implementeres, hvis man vil fange, når brugeren vil lukke vinduet og sørge for, at programmet stopper. Eksempel:

```
import java.awt.event.*;
public class LukProgram implements WindowListener {
    public void windowOpened(WindowEvent e) {};
    public void windowClosing(WindowEvent e) { System.exit(0); }
    public void windowClosed(WindowEvent e) {};
    public void windowIconified(WindowEvent e) {};
    public void windowDeiconified(WindowEvent e) {};
    public void windowActivated(WindowEvent e) {};
    public void windowDeactivated(WindowEvent e) {};
}
```

Klassen kan bruges fra dine egne programmer, ved at tilføje et `LukProgram`-objekt til et vindue, som lytter:

```
vindue.addWindowListener( new LukProgram() );
```

Dette svarer således til at kalde `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`.

## 13.5 Avanceret

Det følgende er nyttigt at vide, hvis man arbejder meget med grafiske brugergrænseflader.

### 13.5.1 Automatisk genereret kode til hændelseshåndtering

Hvis du bruger et værktøj til udviklingen, kan du for hver komponent vælge præcis, hvilke hændelser du vil lytte til (i JBuilder og JDeveloper ved at klikke på events-fanen). Værktøjet genererer så kode til at fange hændelserne.

Dette sker med *anonyme klasser* (beskrevet i kapitel 21). Kigger man på koden, kan man godt få en ide om, hvad der foregår, selvom man ikke kender til anonyme klasser.

Herunder viser vi, hvordan et udviklingsværktøj ville generere koden for LytTilKnap til at fange actionPerformed-hændelsen på kopierKnap (sml. med LytTilKnap i afsnit 13.3.2):

```
kopierKnap.addActionListener(new java.awt.event.ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        kopierKnap_actionPerformed(e);  
    }  
});
```

Det er en anonym klasse (skrevet i fed), der implementerer ActionListener()-interfacet, der bliver føjet til kopierKnap som lytter.

Den anonyme klasse har kun én metode, actionPerformed(), og der sker kun én ting, nemlig at kopierKnap\_actionPerformed() bliver kaldt. Denne metode er defineret andetsteds i koden (nederst i klassen) og i den kan vi sætte vores programkode ind:

```
void kopierKnap_actionPerformed(ActionEvent e) {  
    // vores programkode her, f.eks.  
    t2.setText(t1.getSelectedText() );  
}
```

### 13.5.2 Adaptere

Nogle lytter-interfaces har ret mange metoder og hvis man kun har brug for én af dem, kan det være besværligt at skulle lave en masse tomme metode-kroppe, blot fordi de skal implementeres.

Der findes nogle klasser i standardbiblioteket, som implementerer de tilsvarende lytter-interfaces med tomme metodekroppe. Dem kan man bruge til at arve fra og så bare tilside-sætte de metoder, man har brug for. Disse klasser hedder "adaptere" og har samme navngivning som lytter-interfaces: ComponentAdapter, FocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter og WindowAdapter.

F.eks. kunne Linjelytter fra 13.2 omskrives, så det ikke længere er nødvendigt med tomme metoder til mouseClicked(), mouseEntered() og mouseExited():

```
import java.awt.event.*;  
public class Linjelytter2 extends MouseAdapter  
{  
    public Linjetegning panelet;  
  
    public void mousePressed(MouseEvent hændelse)  
    {  
        panelet.trykpunkt = hændelse.getPoint();  
    }  
  
    public void mouseReleased(MouseEvent hændelse)  
    {  
        panelet.slippunkt = hændelse.getPoint();  
        panelet.repaint();  
    }  
}
```

# 14 Undtagelser og køretidsfejl

Indhold:

- Forstå stakspor
- Fange undtagelser og udskrive stakspor
- Sende undtagelser videre og håndtere dem det rigtige sted

Kapitlet forudsættes i resten af bogen og evnen til at kunne læse et stakspor er vigtig, når man skal finde fejl i sit program.

Forudsætter kapitel 4, Definition af klasser (kapitel 5, Nedarvning er en fordel).

Som programmør skal man tage højde for fejlsituationer, som kan opstå, når programmet udføres. Det gælder f.eks. inddata fra brugeren, der kan være anderledes, end man forventede (brugeren indtaster f.eks. bogstaver et sted, hvor programmet forventer tal) og adgang til ydre enheder, som kan være utilgængelige, f.eks. filer, printere og netværket.

Hvis programmet prøver at udføre en ulovlig handling, vil der opstå en *undtagelse* (eng.: exception) og programudførelsen vil blive afbrudt på det sted, hvor undtagelsen opstod.

Lad os undersøge nærmere, hvad der sker. Herunder prøver vi at indeksere ud over en listes grænser:

```
1 import java.util.*;
2
3 public class SempelUndtagelse
4 {
5     public static void main(String[] arg)
6     {
7         System.out.println("Punkt A");           // punkt A
8         ArrayList l = new ArrayList();
9         System.out.println("Punkt B");           // punkt B
10        l.get(5);
11        System.out.println("Punkt C");           // punkt C
12    }
13 }
```

```
Punkt A
Punkt B
java.lang.ArrayIndexOutOfBoundsException: 5 >= 0
    at java.util.ArrayList.get(ArrayList.java:417)
    at SempelUndtagelse.main(SempelUndtagelse.java:10)
Exception in thread "main"
```

Når vi kører programmet, kan vi se, at det stopper mellem punkt B og C med en fejl:

```
java.lang.ArrayIndexOutOfBoundsException: 5 >= 0
```

Den efterfølgende kode udføres ikke og vi når aldrig punkt C.

---

### Programudførelsen afbrydes, når der opstår en undtagelse

---

I dette kapitel vil vi illustrere, hvordan undtagelser opstår og hvordan de håndteres. Af plads- og overskuelighedshensyn er eksemplerne ret små og undtagelseshåndtering derfor ikke specielt nødvendig. Man skal forestille sig større situationer, hvor der opstår fejl, der ikke lige er til at gennemskue (i dette eksempel kunne der være meget mere kode ved punkt B).

Man kan tænke på undtagelser som en slags protester. Indtil nu har vi regnet med, at objekterne pænt "parerede ordre", når vi gav dem kommandoer eller spørgsmål (kaldte metoder). Fra nu af kan metoderne "spænde ben" og afbryde programudførelsen, hvis situationen er uacceptabel.

Det er det, som `get(5)` på den tomme `ArrayList` gør: Som svar på "giv mig element nummer 5" kaster den `ArrayIndexOutOfBoundsException` og siger "5 >= 0", dvs. "det kan jeg ikke, for 5 er større end antallet af elementer i listen, som er 0!".



## 14.1 Almindelige undtagelser

Ud over `ArrayIndexOutOfBoundsException` som beskrevet ovenfor kan der opstå en række andre fejlsituationer. De mest almindelige er kort beskrevet nedenfor.

Der opstår en undtagelse af typen `NullPointerException`, hvis man kalder metoder på en variabel, der ingen steder refererer hen (en objektreference, der er null):

```
ArrayList l = null;
l.add("x");
Exception in thread "main" java.lang.NullPointerException
    at SimpelUndtagelse.main(SimpelUndtagelse.java:6)
```

Hvis man laver aritmetiske udregninger, kan der opstå undtagelsen `ArithmeticException`, f.eks. ved division med nul:

```
int a = 5;
int b = 0;
System.out.print(a/b);
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at SimpelUndtagelse.main(SimpelUndtagelse.java:7)
```

`ClassCastException` opstår, hvis man prøver at typekonvertere en objektreference til en type, som objektet ikke er, f.eks. en `Gade` til et `Rederi`:

```
Felt f = new Gade("Gade 2", 10000, 400, 1000);
Rederi r = (Rederi) f;
Exception in thread "main" java.lang.ClassCastException: Gade
    at SimpelUndtagelse.main(SimpelUndtagelse.java:6)
```

## 14.2 At fange og håndtere undtagelser

Undtagelser kan fanges og håndteres. Det gøres ved at indkapsle den kritiske kode i en try-blok og behandle eventuelle undtagelser i en catch-blok:

```
try
{
    ...                // programkode hvor der er en risiko
    ...                // for at en undtagelse opstår
}
catch (Undtagelsestype u) // undtagelsen der skal fanges, f.eks. Exception
{
    ...                // kode som håndterer fejl af
    ...                // typen Undtagelsestype
}
...                // dette udføres både hvis ingen undtagelse opstod
...                // og hvis der opstod fejl af typen Undtagelsestype
```

Når programmet kører normalt, springes catch-blokken over. Hvis der opstår undtagelser i try-blokken, hoppes ned i catch-blokken, der håndterer fejlen og derefter udføres koden efter catch.

Undtagelsestypen bestemmer, hvilke slags undtagelser der fanges<sup>1</sup>.

Man kan fange alle slags ved at angive en generel undtagelse, f.eks. `Exception`, eller kun fange en bestemt slags undtagelser, f.eks. `ArrayIndexOutOfBoundsException`.

---

<sup>1</sup> Andre typer undtagelser fanges ikke. Hvis de opstår, afbrydes programmet ligesom uden try-catch.

Ser vi på vores ArrayList-eksempel igen, kunne det med undtagelseshåndtering se ud som:

```
1 import java.util.*;
2 public class SempelUndtagelse2
3 {
4     public static void main(String[] arg)
5     {
6         System.out.println("Punkt A");           // punkt A
7         try
8         {
9             ArrayList l = new ArrayList();
10            System.out.println("Punkt B");         // punkt B
11            l.get(5);
12            System.out.println("Punkt C");         // punkt C
13        }
14        catch (Exception u)
15        {
16            System.out.println("Der opstod en undtagelse!");
17        }
18        System.out.println("Punkt D");           // punkt D
19    }
20 }
```

```
Punkt A
Punkt B
Der opstod en undtagelse!
Punkt D
```

Læg mærke til, at punkt C (der ligger i try-blokken, efter at undtagelsen opstod) ikke bliver udført. Punkt D (efter catch-blokken) bliver udført under alle omstændigheder.

## 14.2.1 Undtagelsesobjekter og deres stakspor

En undtagelse er, ligesom alt andet i Java, repræsenteret ved et objekt. En reference til dette undtagelses-objekt overføres som parameter til catch-blokken.

Objektet har nyttige informationer om fejlen. Metoden `printStackTrace()` udskriver et stakspor (eng.: stack trace), der beskriver de metodekald, der førte til, at undtagelsen opstod:

```
...
catch (Exception u)
{
    System.out.println("Der opstod en undtagelse!");
    u.printStackTrace();
}
...
```

```
Punkt A
Punkt B
Der opstod en undtagelse!
java.lang.ArrayIndexOutOfBoundsException: 5 >= 0
    at java.util.ArrayList.get(ArrayList.java:441)
    at SempelUndtagelse2.main(SempelUndtagelse2.java:11)
Punkt D
```

Staksporet er nyttigt, når man skal finde ud af, hvordan fejlen opstod. Det viser præcist, at undtagelsen opstod i `get()` i `ArrayList`, som blev kaldt fra `SempelUndtagelse2.java` i `main()`-metoden linje 11.

## 14.3 Undtagelser med tvungen håndtering

Indtil nu har oversætteren accepteret vores programmer, hvad enten vi håndterede eventuelle undtagelser eller ej, dvs. det var helt frivilligt, om vi ville tage højde for de mulige fejlsituationer.

Imidlertid er der nogle handlinger, der kræver håndtering, bl.a.:

- læsning og skrivning af filer (kaster bl.a.: `FileNotFoundException`, `IOException`)
- netværkskommunikation (`UnknownHostException`, `SocketException`, `IOException`)
- databaseforespørgsler (`SQLException`)
- indlæsning af klasser (`ClassNotFoundException`)

Når programmøren kalder metoder, der kaster disse undtagelser, *skal* han fange dem.

### 14.3.1 Fange undtagelser eller sende dem videre

Som eksempel vil vi indlæse en linje fra tastaturet og udskrive den på skærmen:

```
import java.io.*;
public class TastaturbrugerFejl
{
    public static void main(String[] arg)
    {
        BufferedReader ind = new BufferedReader(new InputStreamReader(System.in));
        String linje;
        linje = ind.readLine();
        System.out.println("Du skrev: "+linje);
    }
}
```

Metoden `readLine()` læser en linje fra datastrømmen (tastaturet), men når den udføres, kan der opstå undtagelsen `IOException`<sup>1</sup>.

Oversætteren tvinger os til at tage højde for den mulige undtagelse:

```
TastaturbrugerFejl.java:8: unreported exception java.io.IOException; must be
caught or declared to be thrown
    linje = ind.readLine();
```

Fejlmeddelelsen ville på dansk lyde: "I `TastaturbrugerFejl.java` linje 8 er der en uhåndteret undtagelse `IOException`; den skal fanges, eller det skal erklæres, at den bliver kastet".

Vi er altså tvunget til enten at:

1) *fange undtagelsen* ved at indkapsle koden i en try-catch-blok, f.eks.:

```
try {
    linje = ind.readLine();
    System.out.println("Du skrev: "+linje);
} catch (Exception u) {
    u.printStackTrace();
}
```

---

<sup>1</sup> Det er ikke så sandsynligt netop for tastaturindlæsning, men klasserne, vi bruger, er beregnet til at læse vilkårlige datastrømme, måske endda over netværket, og her vil `IOException` opstå, f.eks. hvis datastrømmen er blevet lukket, der ikke er mere data at læse, eller der er opstået en netværksfejl. Scanner-klassen, beskrevet i afsnit 2.3.1, er mere velegnet til netop tastaturindlæsning.

... eller:

2) *erklære, at den bliver kastet*, dvs. at den kan opstå i main()-metoden. Det gør man med ordet *throws*:

```
public static void main(String[] arg) throws IOException
```

Det sidste signalerer, at hvis undtagelsen opstår, skal metoden afbrydes helt og kalderen må håndtere fejlen (i dette tilfælde er det systemet, der har kaldt main(), men oftest vil det være os selv).

---

**Undtagelser med tvungen håndtering skal enten fanges (med try-catch i metodekroppen) eller sendes videre til kalderen (med throws i metodehovedet)**

---

## 14.3.2 Konsekvenser af at sende undtagelser videre

Det har konsekvenser at sende undtagelser videre, for da skal kalderen håndtere dem.

Eksempel: Lad os sige, at vi har uddelegeret læsningen fra tastaturet til en separat Tastatur-klasse, der kan læse en linje fra tastaturet med læsLinje() eller læse en linje og omsætte den til et tal med læsTal():

```
1 import java.io.*;
2
3 public class Tastatur
4 {
5     private BufferedReader ind;
6
7     public Tastatur()
8     {
9         ind = new BufferedReader(new InputStreamReader(System.in));
10    }
11
12    public String læsLinje()
13    {
14        try {
15            String linje = ind.readLine();
16            return linje;
17        } catch (IOException u)
18        {
19            u.printStackTrace();
20        }
21        return null;
22    }
23
24    public double læsTal()
25    {
26        String linje = læsLinje();
27        return Double.parseDouble(linje);
28    }
29 }
```

Tastatur
-ind :BufferedReader
+læsLinje() :String +læsTal() :double

Herover fanger vi undtagelsen IOException ved dens "rod" i læsLinje().

Den kunne gøres simpleere ved at fjerne håndteringen og erklære IOException kastet:

```
public String læsLinje() throws IOException
{
    String linje = ind.readLine();
    return linje;
}
```

Nu sender læsLinje() undtagelserne videre, så nu er det kalderens problem at håndtere den.

Vi kalder selv metoden fra `læsTal()`, så her er vi nu enten nødt til at fange eventuelle undtagelser:

```
public double læsTal()
{
    try {
        String linje = læsLinje();
        return Double.parseDouble(linje);
    } catch (IOException u)
    {
        u.printStackTrace();
    }
    return 0;
}
```

... eller igen sende dem videre.

Herunder er `Tastatur` igen, men `IOException` kastes nu videre fra begge metoder.

```
import java.io.*;

public class TastaturKasterUndtagelser
{
    private BufferedReader ind;

    public TastaturKasterUndtagelser()
    {
        ind = new BufferedReader(new InputStreamReader(System.in));
    }

    public String læsLinje() throws IOException
    {
        String linje = ind.readLine();
        return linje;
    }

    public double læsTal() throws IOException
    {
        String linje = læsLinje();
        return Double.parseDouble(linje);
    }
}
```

Om man skal fange undtagelser eller lade dem "ryge videre" afhænger af, om man selv kan håndtere dem fornuftigt, eller kalderen har brug for at få at vide, at noget gik galt.

Hvad sker der f.eks. i `Tastatur`, hvis der opstår en undtagelse i `læsLinje()` kaldt fra `læsTal()`? Jo, `læsLinje()` returnerer en null-reference til `læsTal()`, der sender denne reference til `parseDouble()`, der sandsynligvis "protesterer" med en `NullPointerException`, for man kan ikke konvertere null til et tal. Der opstår altså en følgeføj, fordi vi fortsætter, som om intet var hændt.

I dette tilfælde må `TastaturKasterUndtagelser` altså siges at være bedst, selvom den altså giver kalderen mere arbejde.

## 14.4 Præcis håndtering af undtagelser

Det kan have væsentlige konsekvenser, på hvilket niveau undtagelserne fanges, selv inden for samme metode.

Lad os bruge Tastatur til at lave et lille regneprogram, der lægger tal sammen. Vi spørger først brugeren, hvor mange tal det skal være (med læsTal()) og derefter kan han taste tal-lene ind. Til sidst spørger vi, om han vil prøve igen.

```
1 public class SumMedTastatur
2 {
3     public static void main(String[] arg)
4     {
5         Tastatur t = new Tastatur();
6         boolean stop = false;
7         try
8         {
9             while (!stop)
10            {
11                System.out.print("Hvor mange tal vil du lægge sammen? ");
12                double antalTal = t.læsTal();
13                double sum = 0;
14
15                for (int i=0; i<antalTal; i=i+1)
16                {
17                    System.out.print("Indtast tal: ");
18                    sum = sum + t.læsTal();
19                }
20                System.out.println("Summen er: "+sum);
21                System.out.print("Vil du prøve igen (j/n)? ");
22                if ("n".equals(t.læsLinje())) stop = true; // undersøg om det er "n"
23            }
24        } catch (Exception u) {
25            System.out.println("Der opstod en undtagelse!");
26            u.printStackTrace();
27        }
28    }
29 }
```

```
Hvor mange tal vil du lægge sammen? 2
Indtast tal: 1
Indtast tal: 2
Summen er: 3.0
Vil du prøve igen (j/n)? j
Hvor mange tal vil du lægge sammen? 3
Indtast tal: 1
Indtast tal: 3
Indtast tal: 5
Summen er: 9.0
Vil du prøve igen (j/n)? n
```

Brugeren taster og taster ... men hvad sker der, hvis han taster forkert?

```
Hvor mange tal vil du lægge sammen? 3
Indtast tal: 1
Indtast tal: 17xxøføf
Der opstod en undtagelse!
java.lang.NumberFormatException: 17xxøføf
    at
    java.lang.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1182)
    at java.lang.Double.parseDouble(Double.java:190)
    at Tastatur.læsTal(Tastatur.java:27)
    at SumMedTastatur.main(SumMedTastatur.java:18)
```

Her opstod en anden undtagelse: 17xxøføf kunne ikke konverteres til et tal. Igen er staksporet nyttigt til at finde fejlen (læst nedefra og op viser det, at main() i linje 18 kaldte læsTal(), der i linje 27 kaldte parseDouble(), der er en del af standardbiblioteket<sup>1</sup>).

Programmet afslutter, da try-catch-blokken er yderst. En smartere opførsel ville være, at den igangværende sum blev afbrudt og brugeren blev bedt om at starte forfra.

---

<sup>1</sup> Selvom det er mindre væsentligt, kan man også se, at parseDouble() faktisk har kaldt en anden metode, nemlig readJavaFormatString().

Det kan vi opnå ved at have try-catch *inde* i while-løkken:

```
public class SumMedTastatur2
{
    public static void main(String[] arg)
    {
        Tastatur t = new Tastatur();
        boolean stop = false;

        while (!stop)
        {
            System.out.print("Hvor mange tal vil du lægge sammen? ");
            try
            {
                double antalTal = t.læsTal();
                double sum = 0;

                for (int i=0; i<antalTal; i=i+1)
                {
                    System.out.print("Indtast tal: ");
                    sum = sum + t.læsTal();
                }
                System.out.println("Summen er: "+sum);
            } catch (Exception u) {
                System.out.println("Indtastningsfejl - " + u);
            }
            System.out.print("Vil du prøve igen (j/n)? ");
            if ("n".equals(t.læsLinje())) stop = true;
        }
    }
}
```

```
Hvor mange tal vil du lægge sammen? 5
Indtast tal: 1
Indtast tal: x2z
Indtastningsfejl - java.lang.NumberFormatException: x2z
Vil du prøve igen (j/n)? j
Hvor mange tal vil du lægge sammen? 3
Indtast tal: 1200
Indtast tal: 1
Indtast tal: 1.9
Summen er: 1202.9
Vil du prøve igen (j/n)? n
```

Hvis en undtagelse opstår, smides den aktuelle sum væk og programmet spørger brugeren, om han vil prøve igen med en ny sum (efter catch-blokken). Svarer han ja, starter programmet forfra i while-løkken.

Med omhyggelig placering af try-catch-blokke kan man altså kontrollere, præcis hvordan programmet skal opføre sig i fejlsituationer:

---

**Kode, hvori der kan opstå en undtagelse og efterfølgende afhængig kode, bør være i samme try-catch-blok**

---

I eksemplet ovenfor finder vi først antallet af tal med læsTal(). Hvis det går galt, giver det heller ikke mening at gå i gang med at udregne en sum, da vi ikke ved, hvor mange tal den skal bestå af.

## 14.5 Fange flere slags undtagelser

Ovenfor har vi behandlet alle undtagelser ens. Det er muligt at hægte flere catch-sætninger med hver sin type undtagelse på samme try-blok.

```
try {
    ...
}
catch (NumberFormatException u1)
{
    System.out.println("Fejl i fortolkningen af inddata");
}
catch (IOException u2)
{
    System.out.println("Inddata kunne ikke læses:"+u2);
}
catch (NullPointerException u3)
{
    u3.printStackTrace();
}
```

Alle undtagelses-klasser arver fra `Exception` og man kan fange *enhver* undtagelse ved, at fange deres fælles superklasse. Fejlhåndteringen bliver så generel, ligegyldigt hvilken type undtagelse der opstod (men husk at udskrive staksporet, så du kan se hvad der skete!)

```
try {
    ...
}
catch (Exception u)
{
    System.out.println("Fejl:");
    u.printStackTrace();
}
```

## 14.6 Resumé

- Hvis en undtagelse opstår, afbrydes programudførelsen og der hoppes til den første catch-blok, der håndterer undtagelsen (eller en superklasse).
- Vha. en try-catch-blok kan man fange en undtagelse og dermed sørge for, at resten af programmet ikke behøver at tage sig af den – man stopper undtagelsen.
- Hvis undtagelsen ikke håndteres der, hvor den opstår, sendes den videre til den kaldende metode. Håndteres den slet ikke, stopper programmet og et stakspor udskrives.
- Et stakspor er en beskrivelse af kald-stakken, dvs. hvilke metoder der har kaldt hvilke. Undtagelsesobjekter har metoden `printStackTrace()` til at udskrive staksporet.
- Nogle undtagelser har tvungen håndtering. Det betyder, at de skal håndteres der, hvor de opstår, eller at metoden skal fortælle til sine kaldere, at den slags undtagelser kan opstå. Det gøres ved at skrive "throws Exception" (eller en nedarving) i metodehovedet.

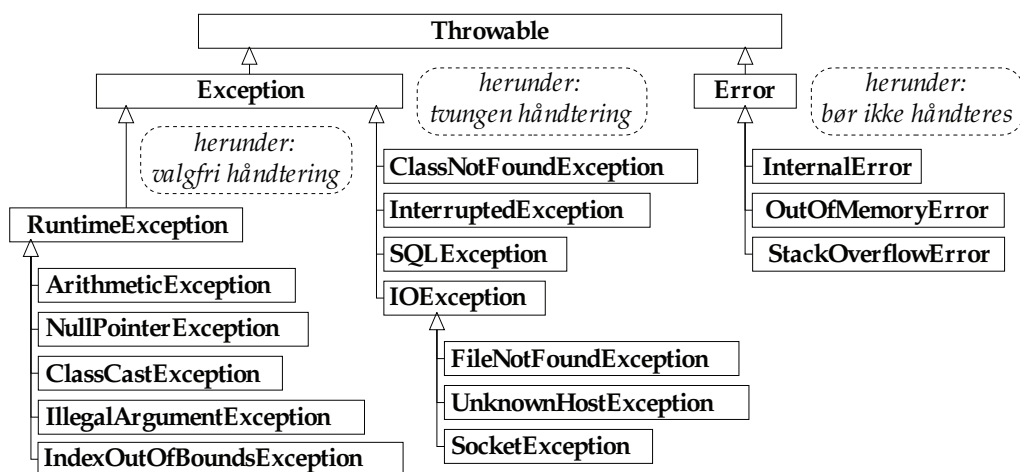
## 14.7 Opgaver

- 1) Flyt try og catch i `SumMedTastatur2` sådan, at programmet smider den aktuelle sum væk og prøver igen uden at spørge brugeren (gør det ved kun at bytte om på linjerne).
- 2) Ret programmet, så det tæller antallet af gange, en sum blev påbegyndt.  
Det er klart, at man skal tælle en variabel op, men hvor skal optællingen placeres?
- 3) Ret programmet, så det også tæller antallet af gange, en sum blev korrekt afsluttet.
- 4) Ændr sådan, at programmet smider den aktuelle indtastning væk, men lader brugeren fortsætte med at regne på den samme sum (vink: Lav for-løkken om til en while-løkke og placér optællingen sådan, at den kun udføres, hvis indtastningen går godt).



# 14.8 Avanceret

Her er en oversigt over de fleste undtagelser.



Til venstre ses undtagelser, der kan opstå hvor som helst i programmet og som er uden tvungen håndtering (eng.: unchecked exception), som ArithmeticException, NullPointerException, ClassCastException, IllegalArgumentException (herunder nedarvingen NumberFormatException), IndexOutOfBoundsException. De arver alle fra RuntimeException.

Midt for er undtagelser i forbindelse med visse metodekald og som derfor har tvungen håndtering (eng.: checked exception): ClassNotFoundException, InterruptedException, SQLException, der alle arver fra Exception og FileNotFoundException, UnknownHostException, SocketException, der alle arver fra IOException.

Til højre ses kritiske fejl som InternalError, OutOfMemoryError og StackOverflowError. Disse fejl bør afslutte programmet og bør derfor aldrig ignoreres. De arver alle fra Error.

## 14.8.1 Fange Throwable

Da alle undtagelser arver fra Exception, kan de fanges med "catch (Exception e)". Derved vil de kritiske fejl (der arver fra Error) alligevel falde igennem. Ønsker man virkelig at fange *alt*, skal man derfor fange Throwable. Det er dog ikke normalt tilrådeligt:

```
try
{
    ...
}
catch (Throwable e)
{
    System.out.println("Fejl:");
    e.printStackTrace();
}
```

## 14.8.2 Selv kaste undtagelser (throw)

I Boks-eksemplerne i kapitel 4 kunne Boksen 'nægte' at sætte dens værdier, men programmet ville i øvrigt fortsætte, som om værdierne var lovlige. Med throw kan man selv kaste undtagelser og afbryde programudførelsen. Herunder et Boks-eksempel, som afbryder programudførelsen med undtagelsen IllegalArgumentException, hvis man forsøger at sætte dens mål til noget ulovligt:

```

public class Boks5
{
    private double længde;
    private double bredde;
    private double højde;

    public Boks5(double lgd, double b, double h)
    {
        sætMål(lgd,b,h);
    }

    public void sætMål(double lgd, double b, double h)
    {
        if (lgd<=0||b<=0||h<=0) throw new IllegalArgumentException("Ugyldige mål.");
        længde = lgd;
        bredde = b;
        højde = h;
    }

    public double volumen()
    {
        return længde*bredde*højde;
    }
}

```

Da `IllegalArgumentException` arver fra `RuntimeException` har den ikke tvungen håndtering og vi behøver hverken skrive `try/catch` eller `throws` i eksemplet.

### 14.8.3 try - finally

Efter (eller i stedet for) en `catch`-blok kan man definere en `finally`-blok. Den bliver *altid* udført, selv når undtagelsen ikke fanges, eller der hoppes ud af metoden på anden vis:

```

public void metode1()
{
    try
    {
        ...
        if (...) return;
        if (...) throw new IllegalArgumentException(...);
        ...
    }
    catch (NullPointerException e)
    {
        System.out.println("Intern fejl:");
        e.printStackTrace();
    }
    finally
    {
        System.out.println("Dette bliver altid udført.");
    }
    System.out.println("Slut på metode1()");
}

```

"Slut på metode1()" springes måske over, hvis metoden returnerer før enden, eller der hoppes ud på grund af en uhåndteret undtagelse (if-sætninger i `try`-blokken). Men selvom metoden ikke afsluttes normalt, bliver "Dette bliver altid udført." alligevel *altid* udskrevet.

En `try-finally`-blok kan være praktisk, hvis man har noget oprydningsskode, man vil være sikker på bliver udført. Bruger man `finally`, bør man kommentere sin kode godt, for det er ikke alle Javaprogrammører, der kender dette fif!

### 14.8.4 Selv definere undtagelsestyper

Synes man ikke, at de foruddefinerede undtagelser passer til ens behov, kan man selv definere sine egne slags undtagelser. Det gøres ved at definere en klasse, der arver fra f.eks `Exception` (eller f.eks. `RuntimeException`, hvis ens undtagelse ikke skal have tvungen håndtering).

# 15 Datastrømme og filhåndtering

Indhold:

- At forstå datastrømme
- At læse og skrive filer
- At analysere tekstfiler og udtrække data
- Overblikket over og sammenhængen mellem alle datastrøm-klasserne

Kapitlet forudsættes i kapitel 16, Netværkskommunikation og 18, Serialisering.

Forudsætter kapitel 14, Undtagelser.

En fil er et arkiv på et lagermedium, hvori der er gemt data. På lagermediet gemmes en række 0'er og 1-taller (bit) i grupper á 8 bit, svarende til en byte (et tal mellem 0 og 255).

Data kan være gemt *binært*, sådan at de kun kan læses af et program eller styresystemet. Det gælder f.eks. en .exe-fil eller et dokument gemt i et proprietært binært format som f.eks. Word. De kan også være gemt som *tekst* uden formatering. Det gælder f.eks. filer, der ender på .txt, .html og .java. Oplysningerne i tekstfiler kan læses med en teksteditor<sup>1</sup>. Det er op til programmet, der læser/skriver i filen at afgøre, om indholdet er tekst eller binært.

I Java behandles filer som datastrømme. En datastrøm er et objekt, som man enten henter data fra (læser fra en datakilde, f.eks. en fil) eller<sup>2</sup> skriver data til (et datamål).

Denne arbejdsmåde gør, at forskellige datakilder og -mål kan behandles ensartet og at det er let at udskifte datakilden eller -målet med noget andet end en fil, f.eks. en forbindelse til netværket.

## 15.1 Skrive til en tekstfil

Klassen `FileWriter` bruges til at skrive en fil med tekstdata. I konstruktøren angiver man filnavnet:

```
FileWriter fil = new FileWriter("tekstfil.txt");
```

`FileWriter`-objektet er en datastrøm, hvis mål er filen. Nu kan man skrive tekstdata til filen med:

```
fil.write("Her kommer et tal:\n");  
fil.write(322+"\n");
```

`FileWriter`-objektets `write()`-metode er lidt besværlig at arbejde med, da den ikke understøtter linjeskift (som så i stedet må laves med `"\n"`, en streng med et linjeskift).

Det er mere bekvemt at lægge objektet ind i en `PrintWriter`. Et `PrintWriter`-objekt har `print()` og `println()`-metoder, som vi er vant til og man skal skrive til den præcis, som når man skriver til `System.out`:

```
PrintWriter ud = new PrintWriter(fil);  
ud.println("Her kommer et tal:");  
ud.println(322);
```

Når vi skriver til `PrintWriter`-objektet, sender det teksten videre til `FileWriter`-objektet, der skriver teksten i filen.

Det er vigtigt at lukke filen, når man er færdig med at skrive. Ellers kan de sidste data gå tabt! Det gør man ved at lukke datastrømmen, man skrev til:

```
ud.close();
```

---

1 Teksten kan dog stadig være kryptisk og uforståelig, som f.eks. .java-kildetekst er for en udenforstående.

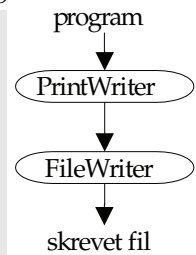
2 Det er dog muligt at åbne en fil for samtidig læsning og skrivning med klassen `RandomAccessFile`, beskrevet i Javadokumentationen.

Her er et samlet eksempel, der skriver nogle fiktive personers navn, køn og alder til en fil:

```
import java.io.*;
public class SkrivTekstfil
{
    public static void main(String[] arg) throws IOException
    {
        FileWriter fil = new FileWriter("skrevet fil.txt");
        PrintWriter ud = new PrintWriter(fil);
        for (int i=0; i<5; i++)
        {
            String navn = "person"+i;
            String køn;
            if (Math.random()>0.5) køn = "m"; else køn = "k";
            int alder = 10+(int) (Math.random()*60);

            ud.println(navn+" "+køn+" "+alder);
        }
        ud.close(); // luk så alle data skrives til disken
        System.out.println("Filen er gemt.");
    }
}
```

Filen er gemt.



Eventuelle IO-undtagelser (f.eks. ikke mere plads på disken) tager vi os ikke af, men sender dem videre til styresystemet (main()-metoden er erklæret som "throws IOException").

Efter at programmet har kørt, findes filen "skrevet fil.txt" på disken, med indhold:

```
person0 m 34
person1 m 26
person2 m 24
person3 k 51
person4 k 16
```

## 15.2 Læse fra en tekstfil

Lad os læse filen ovenfor og skrive den ud til skærmen.

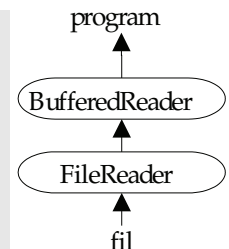
Til det formål bruger vi et **FileReader**-objekt som datakilde. Igen pakker vi det ind i et andet objekt, denne gang af klassen **BufferedReader**. **BufferedReader** gør det mere bekvemt, da indlæsning kan ske linje for linje med metoden **readLine()**. Når der ikke er flere data, returnerer **readLine()** null.

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

public class LaesTekstfil
{
    public static void main(String[] arg) throws IOException
    {
        FileReader fil = new FileReader("skrevet fil.txt");
        BufferedReader ind = new BufferedReader(fil);

        String linje = ind.readLine();
        while (linje != null)
        {
            System.out.println("Læst: "+linje);
            linje = ind.readLine();
        }
    }
}
```

```
Læst: person0 m 34
Læst: person1 m 26
Læst: person2 m 24
Læst: person3 k 51
Læst: person4 k 16
```



## 15.3 Analysering af tekstdata

Ofte er det ikke nok bare at indlæse data, de skal også kunne behandles bagefter.

For eksempel kunne det være sjovt at udregne aldersgennemsnittet i LaesTekstfil.java. Det kræver, at vi først opdeler data i bidder, for at finde kolonnen med aldrene og derefter konverterer dem til tal, der kan regnes på.

### 15.3.1 Opdele strenge og konvertere bidderne til tal

String-klassen har metoden `split()`, der kan dele en streng op i et array bidder. Således vil

```
String tekst = "Hej, kære venner!"
String[] bidder = tekst.split(" ");
```

opdele teksten efter mellemrum og give tre bidder tekst, hhv. "Hej,", "kære" og "venner!".

Argumentet til `split()` er et *regulært udtryk*, hvilket vil sige at man kan lave meget sofistikerede opdelinger (læs mere om regulære udtryk i javadokumentationen til `split()`).

Integer- og Double-klasserne har metoderne hhv. `parseInt()` og `parseDouble()` til at omsætte en streng til et tal<sup>1</sup>. De får en streng som parameter og returnerer den ønskede type:

```
int i = Integer.parseInt("542");
double d = Double.parseDouble("3.14");
```

Eksponentiel notation (hvor 9.8E3 betyder 9800) og andre talsystemer end talsystemet forstås også. F.eks. giver `Integer.parseInt("00010011",2)` tallet 19 (som er 00010011 i det binære talsystem) og `Integer.parseInt("1F",16)` giver 31 (1F i det hexadecimale talsystem):

```
d = Double.parseDouble("9.8E3");           // d = 9800
i = Integer.parseInt("00010011",2);         // i = 19
i = Integer.parseInt("1F",16);              // i = 31
```

Her er et samlet eksempel, der læser filen og finder summen af alle personernes alder:

```
import java.io.*;

public class LaesTekstfil2
{
    public static void main(String[] arg) throws IOException
    {
        BufferedReader ind = new BufferedReader(new FileReader("skrevet fil.txt"));
        String linje = ind.readLine();
        int alderssum = 0;
        while (linje != null)
        {
            String[] bidder = linje.split(" ");           // opdel i bidder efter mellemrum
            alderssum = alderssum + Integer.parseInt(bidder[2]); // brug tredje bid
            linje = ind.readLine();
        }
        System.out.println("Summen af aldrene er: "+alderssum);
    }
}

Summen af aldrene er: 151
```

### 15.3.2 Indlæsning af tekst med Scanner-klassen

For et tekstbaseret (ikke-grafisk) program skal uddata som bekendt skrives til `System.out`.

Det modsvarende objekt til at læse fra tastaturet, `System.in`, er en byte-baseret (binær) datastrøm. Det er nemmest at pakke den ind i et `Scanner`-objekt, som vist afsnit 2.3.1:

```
import java.util.*;

...
Scanner tastatur = new Scanner(System.in);
int alder = tastatur.nextInt();           // læs ét tal
String navn = tastatur.next();            // læs tekst til første mellemrum
String linje = tastatur.nextLine();       // læs (resten af) en hel linje
```

<sup>1</sup> Tilsvarende findes `Byte.parseByte()`, `Long.parseLong()`, `Float.parseFloat()` osv.

## Eksempel: Statistik

Lad os lave et statistikprogram. Vi tæller antallet af personer (linjer i filen) og summen af aldrene. Linjerne analyseres og lægges ind i variablerne navn, køn og alder.

```
import java.io.*;
import java.util.*;
public class LaesTekstfilOgLavStatistik
{
    public static void main(String[] arg)
    {
        int antalPersoner = 0;
        int sumAlder = 0;

        try
        {
            Scanner sc = new Scanner(new FileReader("skrevet fil.txt"));

            while (sc.hasNext())
            {
                try
                {
                    String navn = sc.next(); // læs tekst til første mellemrum
                    String køn = sc.next(); // læs tekst til næste mellemrum
                    int alder = sc.nextInt(); // læs ét tal

                    System.out.println(navn+" er "+alder+" år.");
                    antalPersoner = antalPersoner + 1;
                    sumAlder = sumAlder + alder;
                }
                catch (Exception u)
                {
                    System.out.println("Fejl. Linjen springes over.");
                    u.printStackTrace();
                }
                sc.hasNextLine(); // hop til næste linje
            }

            System.out.println("Aldersgennemsnittet er: "+sumAlder/antalPersoner);
        }
        catch (FileNotFoundException u)
        {
            System.out.println("Filen kunne ikke findes.");
        }
        catch (Exception u)
        {
            System.out.println("Fejl ved læsning af fil.");
            u.printStackTrace();
        }
    }
}

person0 er 34 år.
person1 er 26 år.
person2 er 24 år.
person3 er 51 år.
person4 er 16 år.
Aldersgennemsnittet er: 30
```

Undervejs kan der opstå forskellige undtagelser. Hvis filen ikke eksisterer, udskrives "Filen kunne ikke findes" og programmet afslutter. En anden mulig fejl er, at filen er tom. Så vil der opstå en aritmetisk undtagelse (division med nul), når vi dividerer med antalPersoner og "Fejl ved læsning af fil" udskrives.

Under analyseringen af linjen, kan der også opstå flere slags undtagelser: Konverteringen til heltal kan gå galt og der kan være for få eller for mange bidder. Hvis disse fejl opstår, fortsætter programmet efter catch-blokken med, at læse næste linje af inddata.

Da sumAlder og antalPersoner ændres sidst i try-blokken, vil de kun blive opdateret, hvis hele linjen er i orden og statistikken udregnes derfor kun på grundlag af de gyldige linjer.

## 15.4 Binær læsning og skrivning

Arbejder man med binære data (f.eks. lyd- eller billedfiler), skal det ske binært. En anden grund til at arbejde med binære data er, at oversættelsen fra binære til tekstdata tager tid.

Det følgende eksempel kopierer en fil (bog.html) binært til en anden fil:

```
import java.io.*;

public class KopierFil
{
    public static void main(String[] arg) throws IOException
    {
        InputStream is = new FileInputStream("bog.html");
        OutputStream os = new FileOutputStream("kopieretBog.html");

        // brug evt. buffere i læsning og skrivning (mere effektivt)           punkt A
        // is = new BufferedInputStream(is);
        // os = new BufferedOutputStream(os);

        // husk starttidspunkt, så vi kan måle hvor lang tid det tager
        long starttid = System.currentTimeMillis();

        // læs og skriv én byte ad gangen (ret ineffektivt)                   punkt B
        int b = is.read();
        while (b != -1)
        {
            os.write(b);
            b = is.read();
        }

        is.close();
        os.close();
        long sluttid = System.currentTimeMillis();
        System.out.println("Kopiering tog " + (sluttid-starttid)*0.001 + " sek.");
    }
}
```

Kopiering tog 4.713 sek.

Programmet, som det umiddelbart ser ud (punkt A er kommenteret ud), udfører opgaven ved at læse én byte fra filen, skrive den til den anden fil, læse en ny byte o.s.v. Det går ret langsomt, filen bog.html, der fylder ca. 100 kb, tager knap fem sekunder at kopiere.

### 15.4.1 Optimering af ydelse

Her udføres programmet igen, med punkt A kommenteret ind, så der bruges buffere til læsning og skrivning. Programmet udskriver da:

Kopiering tog 0.089 sek.

Vi opnår altså en hastighedsforøgelse på over en faktor halvtreds (!) ved at bruge buffere. Bufferne tager højde for vores en-byte-ad-gangen-kopiering og bevirker, at læsning og skrivning sker i klumper á et par kilobyte, hvilket er langt mere effektivt.

Vi kunne også selv sørge for, at data bliver læst i større klumper, ved at bruge et array (beskrevet i kapitel 8). Erstatte vi punkt B med det følgende, går det endnu hurtigere:

```
// læs og skriv i større klumper (mere effektivt)
byte[] data = new byte[4096]; // 4 kilobyte
int lgd = is.read(data);
while (lgd != -1)
{
    os.write(data, 0, lgd);
    lgd = is.read(data);
}
```

Kopiering tog 0.0060 sek.

Denne tid (75 gange hurtigere) er uafhængig af, om der bruges buffere eller ej (i punkt A).

---

**Brug buffere eller sørg for, at dit program behandler data i større klumper**

---



## 15.5 Appendiks

I pakken `java.io` findes omkring 40 klasser, der kan læse eller skrive binære eller tegnbaserede data fra et væld af datakilder eller -mål og på et væld af forskellige måder. Der henvises til javadokumentationen for en nærmere beskrivelse af de enkelte klasser.

Næsten alle metoderne i klasserne kan kaste en `IOException`-undtagelse, som skal fanges i en `try-catch`-blok (eller kastes videre som beskrevet i kapitlet om undtagelser).

### 15.5.1 Navngivning

Datastrømmene kan ordnes i fire grupper og den konsistente navngivning gør dem lettere at overskue:

**InputStream**-objekter læser binære data.

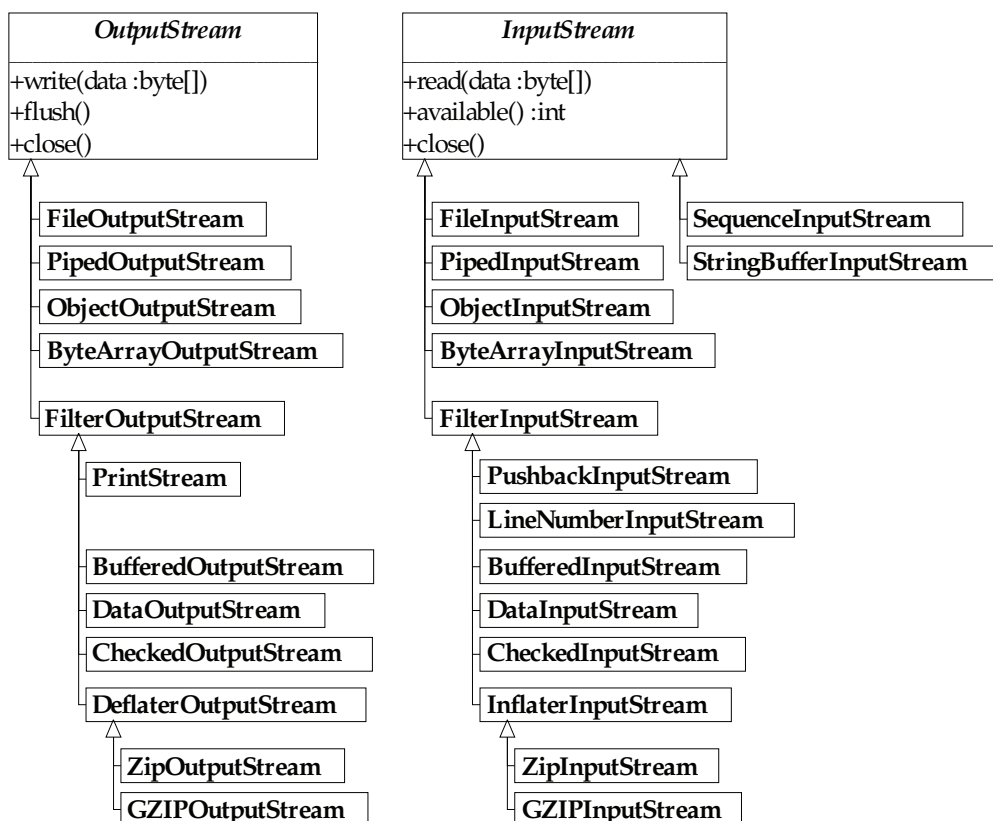
**OutputStream**-objekter skriver binære data.

**Reader**-objekter læser tekstdata.

**Writer**-objekter skriver tekstdata.

### 15.5.2 Binære data ( -OutputStream og -InputStream)

Byte-baserede data som f.eks. billeder, lyde eller andre binære programdata håndteres af klasser, der arver fra `InputStream` eller `OutputStream`.

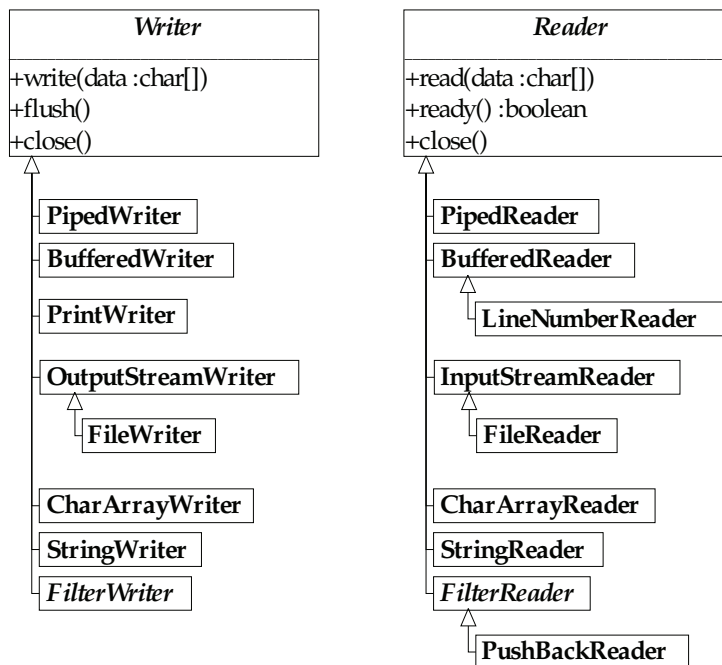


Af klassesdiagrammet ses, at metoderne i `InputStream` og `OutputStream` læser og skriver byte-data: `write(byte[])` på `OutputStream` skriver et array (en række) af byte. Arvingerne har lignende metoder (disse er ikke vist).

InputStream og OutputStream er tegnet i kursiv. Det er fordi de er *abstrakte* klasser og det betyder, at man ikke kan oprette InputStream og OutputStream-objekter direkte med f.eks. `new InputStream()`. I stedet skal man bruge en af nedarvingerne. Abstrakte klasser bliver behandlet i kapitel 21, Avancerede klasser.

### 15.5.3 Tekstdata ( -Writer og -Reader)

Tegn-baserede data bruges til tekstfiler, til at læse brugerinput og til meget netværks-kommunikation. Dette håndteres af klasserne, der nedarver fra Reader og Writer.



Af klassediagrammet ses, at alle metoderne i Reader og Writer læser og skriver tegndata (datatype char). Tegn repræsenteres i Java som 16-bit unikode-værdier og man kan derfor arbejde med, ikke blot det vesteuropæiske tegnsæt, men også det østeuropæiske, det kinesiske alfabet, det japanske, det kyrilliske, det græske o.s.v..

### 15.5.4 Fillæsning og -skrivning (File- )

Klasserne til filhåndtering er `FileInputStream`, `FileReader`, `FileOutputStream` og `FileWriter`.

### 15.5.5 Streng (String- )

Med `StringReader` kan man læse data fra en streng, som om det kom fra en datastrøm. Det kan være praktisk til f.eks. at simulere indtastninger fra tastaturet under test eller input fra en fil (sml. afsnit 15.3.2, Indlæsning af tekst med Scanner-klassen).

```
StringReader tegnlæser = new StringReader("Jacob\n4\n5.14\n");
BufferedReader ind = new BufferedReader( tegnlæser );
```

`StringWriter` er en datastrøm, der gemmer data i et `StringBuilder`-objekt (se afsnit 3.10.1). Når man er færdig med at skrive, kan man få den samlede streng ud ved at kalde `toString()`.

## 15.5.6 Arrays (ByteArray- og CharArray- )

Et array er en liste eller række af noget (se kapitel 8). Ligesom man kan behandle en streng som en datastrøm, kan man også arbejde med et array som datakilde eller -mål. Klasserne `CharArrayReader` og `CharArrayWriter` hhv. læser og skriver fra et array af tegn, mens `ByteArrayInputStream` og `ByteArrayOutputStream` læser og skriver binært i et array af byte.

## 15.5.7 Læse og skrive objekter (Object- )

Det er muligt at skrive hele objekter ned i en datastrøm med `ObjectOutputStream`. Objekterne bliver da "serialiseret", dvs. dets data gemmes i datastrømmen. Refererer objektet til andre objekter, bliver disse også serialiseret og så fremdeles. Dette er nyttigt til at gemme en hel graf af objekter på disken, for senere at hente den frem igen. Emnet vil blive behandlet mere i kapitel 18, Serialisering.

## 15.5.8 Dataopsamling (Buffered- )

Klasserne `BufferedInputStream`, `BufferedReader`, `BufferedOutputStream` og `BufferedWriter` sørger for en buffer (et opsamlingsområde) til datastrømmen. Det sikrer mere effektiv indlæsning, fordi der bliver læst/skrevet større blokke data ad gangen.

`BufferedReader` sørger også for, at man kan læse en hel linje af datastrømmen ad gangen.

## 15.5.9 Gå fra binære til tegnbaserede datastrømme

Nogen gange står man med en binær datastrøm og ønsker at arbejde med den, som om den var tekstbaseret. Der er to klasser, der konverterer til tegnbaseret indlæsning og -udlæsning:

**`InputStreamReader`** er et `Reader`-objekt, der læser fra en `InputStream` (byte til tegn).

**`OutputStreamWriter`** er et `Writer`-objekt, der skriver til en `OutputStream` (tegn til byte).

## 15.5.10 Filtreringsklasser til konvertering og data-behandling

Klasserne, der arver fra `FilterOutputStream` og `FilterInputStream`, sørger alle for en eller anden form for behandling og præsentation, der letter programmørens arbejde:

**`LineNumber`**-klasser tæller antallet af linjeskift i datastrømmen, men lader den ellers være uændret.

**`Pushback`**-klasser giver mulighed for at skubbe data tilbage i datastrømmen (nyttigt, hvis man af en eller anden grund kan "komme til" at læse for langt).

**`SequenceInputStream`** sætter to eller flere datakilder i forlængelse af hinanden.

**`Piped`**-klasserne letter datakommunikationen mellem to tråde (samtidige programudførelsespunkter i et program) ved at sætte data "i kø" sådan, at en tråd kan læse fra datastrømmen og en anden skrive.

**`Checked`**-klasserne (i pakken `java.util.zip`) udregner en checksum på data. Det kan være nyttigt til at undersøge, om nogen eller noget har ændret data (f.eks. en cracker eller en dårlig diskette). Man skal angive et checksum-objekt, f.eks. `Adler32` eller `CRC32`.

**`Zip`**-klasserne (i `java.util.zip`) læser og skriver zip-filer (lavet af f.eks. `WinZip`). De er lidt indviklede at bruge, da de er indrettet til at håndtere pakning af flere filer.

**GZIP**-klasserne (i `java.util.zip`) komprimerer og dekomprimerer data med Lempel-Ziv-kompression, kendt fra filer, der ender på `.gz` på UNIX-systemer (især Linux). Er nemmere at bruge end **Zip**-klasserne, hvis man kun ønsker at pakke én fil.

## 15.5.11 Brug af på filtreringsklasser

Filtreringsklasser skydes ind som ekstra "indpakning" mellem de andre datastrømme. F.eks. kan

```
PrintWriter ud = new PrintWriter( new FileOutputStream("fil.txt"));
```

ændres til også at komprimere uddata, simpelthen ved at skyde et `GZIPOutputStream`-objekt ind:

```
PrintWriter ud = new PrintWriter( new GZIPOutputStream(  
    new FileOutputStream("fil.txt.gz"));
```

Herunder læser vi en fil og udregner filens checksum og antallet af linjer i filen.

```
import java.io.*;  
import java.util.zip.*;  
  
public class UndersoegFil  
{  
    public static void main(String[] arg) throws IOException  
    {  
        FileInputStream fil = new FileInputStream("skrevet fil.txt");  
        BufferedInputStream bstrøm = new BufferedInputStream(fil);  
        CRC32 checksum = new CRC32();  
        CheckedInputStream chkstrøm = new CheckedInputStream(bstrøm,checksum);  
        InputStreamReader txtstrøm = new InputStreamReader(chkstrøm);  
        LineNumberReader ind = new LineNumberReader(txtstrøm);  
  
        String linje;  
        while ((linje=ind.readLine())!= null) System.out.println("Læst: "+linje);  
  
        System.out.println("Antal linjer: " +ind.getLineNumber());  
        System.out.println("Checksum (CRC):" +checksum.getValue());  
    }  
}  
  
Læst: person0 k 43  
Læst: person1 k 10  
Læst: person2 k 16  
Læst: person3 k 11  
Læst: person4 k 21  
Antal linjer: 5  
Checksum (CRC):3543848051
```

Læg mærke til, hvordan vi hæfter datastrøm-objekterne sammen i en kæde ved hele tiden at bruge det forrige objekt som parameter til konstruktørerne: Filindlæsning, buffering, checksum, gå fra binær til tekstbaseret indlæsning (`InputStreamReader`) og linjetælling.

While-løkken er skrevet meget kompakt med en tildeling (`linje=ind.readLine()`) og derefter en sammenligning, om værdien af tildelingen var null (`(..) != null`).

## 15.6 Test dig selv

- 1) Beskriv de forskellige kategorier af datastrømme.
- 2) Hvilken slags undtagelser kan opstå, når man bruger datastrømme?
- 3) Hvad skal man altid huske at gøre, når man skriver data, f.eks. til en fil?

## 15.7 Resumé

- De fire hovedkategorier for datastrømme er: `InputStream` (læser binære data), `OutputStream` (skriver binære data), `Reader` (læser tekstdata) og `Writer` (skriver tekstdata). De fleste kategorier har filtre, der behandler datastrømmen på en eller anden måde.
- Næsten alle metoder på datastrøm-objekter kan kaste `IOException`, der skal håndteres.
- **Husk:** For at være sikker på, at data bliver gemt på det underliggende medium, skal man altid lukke en datastrøm (med `close()`-metoden), når man er færdig med at skrive til den!

## 15.8 Opgaver

Prøv eksemplerne fra kapitlet og:

- 1) Udvid `LaesTekstfilOgLavStatistik.java` sådan, at linjer, der starter med "#", er kommentarer, som ignoreres og afprøv, om programmet virker.
- 2) Lav et program, der læser fra en tekstfil, `skyld.txt`, og udskriver summen af tallene i hver linje med navnet foranstillet (f.eks. Anne: 450). Filen kunne f.eks. indeholde:  
Anne 300 150  
Peter 18 300 900 -950  
Lis 1000 13.5
- 3) Skriv programmet `Grep.java`, der læser en fil og udskriver alle linjer, der indeholder en bestemt delstreng (vink: `Ret LaesTekstfil.java` – en linje undersøges for en delstreng med `substring(...)`).
- 4) Skriv programmet `Diff.java`, der sammenligner to tekstfiler linje for linje og udskriver de linjer, der er forskellige.
- 5) Ret `SkrivTekstfil.java` til `SkrivKomprimeretTekstfil.java`, der gemmer data komprimeret med `GZIPOutputStream` (se appendiks).
- 6) Lav den tilsvarende `LaesKomprimeretTekstfilOgLavStatistik.java`.
- 7) Kør `KopierFil` på din maskine og se, hvor lang tid det tager (husk at lægge en fil med navn `bog.html` på ca. 100 kb det rette sted, eller ret filnavnet i programmet).  
Prøv derefter, hvor du bruger buffere for mere effektiv læsning og skrivning.  
Prøv igen, hvor programmet læser og skriver 4 kb ad gangen.  
Gør det nu nogen forskel, om du bruger buffere? Hvorfor/hvorfor ikke?

# 15.9 Avanceret

## 15.9.1 Klassen RandomAccessFile

I stedet for at bruge datastrømme, kan man også åbne en fil på mere traditionel vis med klassen `RandomAccessFile`, der kan både læse og skrive i samme fil.

Filen opfører sig stort set som en række byte: Man har en fil-pointer, der husker, hvor langt man er nået i filen, og som kan sættes til et vilkårligt sted i filen (med metoden `seek()`), hvorefter der kan læses eller skrives data (som strenge, byte eller nogen af de simple typer).

## 15.9.2 Filhåndtering (klassen File)

Klassen `File`, der repræsenterer en fil eller en mappe, kan bruges til at navigere i filsystemet, slette eller omdøbe filer eller mapper og aflæse eller sætte deres attributter.

Det følgende eksempel lister alle filerne i den aktuelle mappe. Er der nogle undermapper, listes indholdet af dem også (`listMappe()` kalder sig selv, så her anvendes rekursion).

```
import java.io.*;
public class ListFilerRekursivt
{
    public static void main(String[] arg)
    {
        File m = new File("."); // objekt der repræsenterer den aktuelle mappe
        listMappe(m);
    }

    private static void listMappe(File mappe)
    {
        File[] filer = mappe.listFiles();

        for (int i=0; i<filer.length; i++)
        {
            File f = filer[i];
            System.out.println(f);    // udskriv filens/mappens navn og sti

            if (f.isDirectory())
                listMappe(f);        // kald listMappe() rekursivt
        }
    }
}

./LaesTekstfil.java
./SkrivTekstfil.java
./ListFilerRekursivt.java
./class-filer
./class-filer/LaesTekstfil.class
./class-filer/SkrivTekstfil.class
./class-filer/ListFilerRekursivt.class
./bog.html
./kopieretBog.html
```

Programmet er startet fra en mappe, der bl.a. indeholdt `LaesTekstfil.java`, `SkrivTekstfil.java` og andre filer og undermappen 'class-filer', der indeholdt de tilsvarende .class-filer.

## 15.9.3 Platformuafhængige filnavne

Da kørslen er sket under Linux/UNIX, er sti-separatoren tegnet '/', mens det under Windows/DOS ville være '\'. Derfor bør man bruge variabelen `File.separator`, der giver den pågældende platforms sti-separatortegn, og ikke angive / eller \ direkte i stinavne.

Derudover findes `System.getProperty("user.home")`, der angiver brugerens hjemmemappe og `System.getProperty("user.dir")`, der giver stien til den aktuelle mappe.

En liste over de tilgængelige drevbogstaver kan fås med kaldet `FileSystem.listRoots()`.

# 16 Netværkskommunikation

Indhold:

- At koble til en tjeneste på en fjernmaskine
- At udbyde tjenester på netværket
- URL-klassen og dens muligheder
- Eksempler: Hente en hjemmeside og en webserver

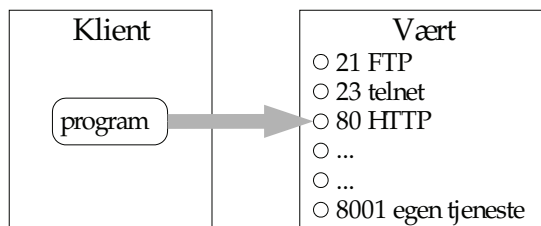
Kapitlet forudsættes ikke i resten af bogen.

Forudsætter 15, Datastrømme og filhåndtering.

Alle maskiner på et TCP-IP-netværk, f.eks. internettet, har et IP-nummer. Det er en talrække på fire byte<sup>1</sup>, der unikt identificerer en maskine på nettet, f.eks. 195.215.15.20.

Normalt bruger man en navnetjeneste (eng.: Domain Name Service – DNS), der sammenholder alle numre med navne, der er nemmere at huske, f.eks. [www.cv.ihk.dk](http://www.cv.ihk.dk) eller [www.esperanto.dk](http://www.esperanto.dk). Adressen localhost (IP-nummer 127.0.0.1) er speciel ved altid at pege på den maskine, man selv sidder ved.

Kommunikation mellem to maskiner sker ved, at værtsmaskinen (eng.: host) gør en tjeneste (eng.: service) tilgængelig på en bestemt port, hvorefter klienter kan åbne en forbindelse til tjenesten.



Hjemmesider (HTTP-tjenesten) er tilgængelige på port 80. Filoverførsel (FTP-tjenesten) er på port 21 og hvis man vil logge ind på maskinen (telnet-tjenesten), er det port 23.

I det følgende vil vi vise, hvordan man bruger og udbyder HTTP-tjenesten til hjemmesider, men andre former for netværkskommunikation foregår på lignende måder.

## 16.1 At forbinde til en port

Man opretter en forbindelse ved at oprette et Socket-objekt<sup>2</sup> og angive værtsmaskinen og porten i konstruktøren. Vil man f.eks. kontakte webserveren på [www.esperanto.dk](http://www.esperanto.dk), skriver man:

```
Socket forbindelse = new Socket("www.esperanto.dk", 80);
```

Hvis alt gik godt, har Socket-objektet (forbindelsen eller "soklen") nu kontakt med værtsmaskinen (ellers har den kastet en undtagelse).

Nu skal vi have fat i datastrømmene fra os til værten og fra værten til os:

```
OutputStream binærUd = forbindelse.getOutputStream();
InputStream binærInd = forbindelse.getInputStream();
```

Hvis vi vil sende/modtage binære data, kan vi nu bare gå i gang: `binærUd.write()` sender en byte eller en række af byte til værten og `binærInd.read()` modtager en eller flere byte.

Hvis det er tekstkommunikation, er `PrintWriter` og `BufferedReader` (der arbejder med tegn og strenge som beskrevet i kapitel 15) dog nemmere at bruge:

```
PrintWriter ud = new PrintWriter(binærUd);
BufferedReader ind = new BufferedReader(new InputStreamReader(binærInd));
```

Nu kan vi f.eks. bede om startsidens (svarende til <http://www.esperanto.dk/index.html>) ved at sende "GET /index.html HTTP/0.9", "Host: [www.esperanto.dk](http://www.esperanto.dk)" og en blank linje:

```
ud.println("GET /index.html HTTP/0.9");
ud.println("Host: www.esperanto.dk");
ud.println();
ud.flush();
```

Kaldet til `flush()` sikrer, at alle data er sendt til værten, ved at tømme eventuelle buffere.

- 1 For den nye version af IP-protokollen, IPv6, som man regner med vil slå igennem omkring år 2005, er det 16 byte. Den gamle vil dog blive understøttet mange år frem.
- 2 Netop HTTP kan egentligt klares nemmere med URL-klassen (se senere). Vi bruger Socket-klassen i det følgende, da den kan anvendes til alle former for netværkskommunikation.



Nu sender værten svaret, der kan læses fra inddatastrømmen<sup>3</sup>:

```
String s = ind.readLine();
while (s != null) {
    System.out.println("modt: "+s);
    s = ind.readLine();
}
```

While-løkken bliver ved med at læse linjer. Når der ikke er flere data (værten har sendt alle data og lukket forbindelsen), returnerer `ind.readLine()` null og løkken afbrydes.

Her er hele programmet:

```
import java.io.*;
import java.net.*;
public class HentHjemmeside
{
    public static void main(String[] arg)
    {
        try {
            Socket forbindelse = new Socket("www.esperanto.dk",80);
            OutputStream binærUd = forbindelse.getOutputStream();
            InputStream binærInd = forbindelse.getInputStream();
            PrintWriter ud = new PrintWriter(binærUd);
            BufferedReader ind = new BufferedReader(new InputStreamReader(binærInd));
            ud.println("GET /index.html HTTP/0.9");
            ud.println("Host: www.esperanto.dk");
            ud.println();
            ud.flush(); // send anmodning afsted til værten
            String s = ind.readLine();
            while (s != null) { // readLine() giver null når datastrømmen lukkes
                System.out.println("modt: "+s);
                s = ind.readLine();
            }
            forbindelse.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
modt: HTTP/1.1 200 OK
modt: Date: Tue, 17 Apr 2001 13:06:06 GMT
modt: Server: Apache/1.3.12 (Unix) (Red Hat/Linux) PHP/4.0.2 mod_perl/1.24
modt: Last-Modified: Thu, 05 Mar 1998 17:28:16 GMT
modt: Content-Length: 896
modt: Content-Type: text/html
modt:
modt: <HTML><HEAD><TITLE>Esperanto.dk</TITLE>
modt: <META name="description" content="Den officielle danske hjemmeside om
modt: plansproget esperanto. Oficiala dana hejmpagxo pri la planlingvo Esperanto.">
modt: <META name="keywords" content="Esperanto, Danmark, Danio, Esperanto-nyt,
modt: Zamenhof, bogsalg, plansprog, lingvistik, libroservo, planlingvo, lingvistiko">
modt: </HEAD>
modt: <FRAMESET cols="22%,*">
modt: <FRAME NAME="menu" SRC="da/menu.htm" MARGINWIDTH=0>
modt: <FRAME NAME="indhold" SRC="da/velkomst.htm">
modt: <NOFRAMES>
modt: Velkommen til Esperanto.dk!<p>
modt: Gå til <a href="da/velkomst.htm">velkomst-siden</a>,
modt: eller til <a href="da/menu.htm">indholdsfortegnelsen</a>.
modt: </NOFRAMES>
modt: </FRAMESET>
modt: </HTML>
```

Det ses, at svaret starter med et hoved med metadata, der beskriver indholdet (dato, værtenes styresystem, hvornår dokumentet sidst blev ændret, længde, type).

Derefter kommer en blank linje og så selve indholdet (HTML-kode).

Dette er i overensstemmelse med måden, som data skal sendes på ifølge HTTP-protokollen. Protokollen er løbende blevet udbygget. En af de tidligste (og dermed simpleste) var HTTP/0.9, mens de fleste moderne programmer bruger HTTP/1.1.

<sup>3</sup> I virkeligheden tager det noget tid, før data når frem til klienten, men så vil *read*-metoderne *bloke-re*, dvs. vente på, at der er data.

## 16.2 At lytte på en port

For at lave et program, der fungerer som vært (dvs. som andre maskiner/programmer kan forbinde sig til), opretter man et `ServerSocket`-objekt, der accepterer anmodninger på en bestemt port:

```
ServerSocket værtssokkel = new ServerSocket(8001);
```

Nu lytter vi på port 8001. Så er det bare at vente på, at der kommer en anmodning:

```
Socket forbindelse = værtssokkel.accept();
```

Kaldet af `accept()` venter på, at en klient forbinder sig og når det sker, returnerer kaldet med en forbindelse til klienten i et `Socket`-objekt.

Derefter kan vi arbejde med forbindelsen ligesom før. Ligesom når to mennesker snakker sammen, har det ikke den store betydning, hvem der startede samtalen, når den først er kommet i gang.

I tilfældet med HTTP-protokollen er det defineret, at klienten først skal spørge og værten (webserveren) derefter svare, så vi læser først en anmodning

```
String anmodning = ind.readLine();  
System.out.println("Anmodning: "+anmodning);
```

... og sender derefter et svar, tømmer databufferen og lukker forbindelsen:

```
ud.println("HTTP/0.9 200 OK");  
ud.println();  
ud.println("<html><head><title>Svar</title></head>");  
ud.println("<body><h1>Kære bruger</h1>");  
ud.println("Du har spurgt om "+anmodning+", men der er intet her.");  
ud.println("</body></html>");  
ud.flush();  
forbindelse.close();
```

Bemærk, at `ud.flush()` skal ske, før vi lukker forbindelsen, ellers går svaret helt eller delvist tabt; `Socket`-objektet ved ikke, om datastrømmen har nogle data liggende, der ikke er blevet sendt endnu.

Herunder ses det fulde program.

```

import java.io.*;
import java.net.*;
public class Hjemmesidevaert
{
    public static void main(String[] arg)
    {
        try {
            ServerSocket vartssokkel = new ServerSocket(8001);
            while (true)
            {
                Socket forbindelse = vartssokkel.accept();
                PrintWriter ud = new PrintWriter(forbindelse.getOutputStream());

                BufferedReader ind = new BufferedReader(
                    new InputStreamReader(forbindelse.getInputStream()));

                String anmodning = ind.readLine();
                System.out.println("Anmodning: "+anmodning);

                ud.println("HTTP/0.9 200 OK");
                ud.println();
                ud.println("<html><head><title>Svar</title></head>");
                ud.println("<body><h1>Kære bruger</h1>");
                ud.println("Du har spurgt om "+anmodning+", men der er intet her.");
                ud.println("</body></html>");
                ud.flush();
                forbindelse.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

---

```

Anmodning: GET / HTTP/0.9
Anmodning: GET / HTTP/1.1
Anmodning: GET /xx.html HTTP/1.1

```

Du kan afprøve programmet ved at ændre på HentHjemmeside.java til at spørge 'localhost' på port 8001 eller ved i en netlæser at åbne adressen <http://localhost:8001/xx.html>

## 16.3 URL-klassen

HTTP-forespørgsler kan egentligt klares nemmere ved, at man bruger URL-klassen, der er indrettet til netop dette (og som tager højde for eventuelle brandmure og proxyer) og som tillader at arbejde på et mere overordnet niveau, uden at kende til detaljerne i HTTP-protokollen. Herunder er HentHjemmeside igen, men hvor URL-klassen bruges i stedet.

```

import java.io.*;
import java.net.*;
public class HentHjemmesideMedURL
{
    public static void main(String[] arg)
    {
        try {
            URL url = new URL("http://www.esperanto.dk");
            InputStream binærInd = url.openStream();
            BufferedReader ind = new BufferedReader(new InputStreamReader(binærInd));
            String s = ind.readLine();
            while (s != null)
            {
                System.out.println("modt: "+s);
                s = ind.readLine();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

---

```

modt: <HTML><HEAD><TITLE>Esperanto.dk</TITLE>
modt: <META name="description" content="Den officielle danske hjemmeside om
plansproget esperanto. Oficiala dana hejmpagxo pri la planlingvo Esperanto.">
...

```

Faktisk er URL-klassen meget kapabel og understøtter mere end HTTP-protokollen.

Man kan åbne en fil på filsystemet med f.eks.:

```
new URL("file:sti/filnavn.txt")
```

Man kan endda åbne et jar- eller zip-arkiv (her arkiv.jar) og læse en fil fra det:

```
new URL("jar:file:arkiv.jar!/fil_i_arkivet.txt")
```

Man kan også bruge anonym FTP til at hente filer og liste mapper:

```
new URL("ftp://sunsite.dk/")
```

... eller logge ind med brugernavn og adgangskode:

```
new URL("ftp://brugernavn:adgangskode@ftp.vært.dk/fil.txt")
```

URL-klassen understøtter ikke at gemme filer (hverken på disk eller over FTP/HTTP).

## 16.4 Opgaver

- 1) Lav Hjemmesidevaert om, så den, afhængig af anmodningen, kan give tre forskellige svar.
- 2) Skriv HentHjemmeside om, så den spørger den lokale maskine ('localhost') port 8001 og brug den til at teste Hjemmesidevaert (der køres i en separat proces).
- 3) Lav en virtuel opslagstavle. Den skal bestå af klasserne Opslagstavletjeneste, som udbyder tjenesten (brugport 8002) og Opslagstavleklient, som forbinder sig til tjenesten. Opslagstavletjeneste skal understøtte to former for anmodninger: 1) TILFØJ, der føjer en besked til opslagstavlen og 2) HENTALLE, der sender alle opslag til klienten. Afprøv begge slags anmodninger fra Opslagstavleklient.
- 4) Lav din egen mellemvært (eng.: proxy – betyder egentlig 'stråmand'), der modtager en HTTP-forespørgsel og spørger videre for klienten.
- 5) Prøv hver af de andre eksempler på URL-adresser i HentHjemmesideMedURL.

# 16.5 Avanceret

## 16.5.1 FTP-kommunikation

FTP-protokollen er noget mere indviklet end HTTP-protokollen, men giver mulighed for langt flere ting – bl.a. at gemme data på værtsmaskinen. Det fører for vidt at gennemgå FTP-protokollen her, men du kan lære en del om, hvordan den fungerer ved at lege med programkoden herunder.

```
public class BenytFtpForbindelse {
    public static void main(String[] a) throws Exception
    {
        FtpForbindelse f = new FtpForbindelse();
        // bemærk - vær altid MEGET FORSIGTIG med at angive adgangskoder i en fil!!
        f.forbind("pingo.cv.ihk.dk","nordfalk","adgangskoden");

        f.sendKommando("HELP");    // få liste over kommandoer som tjenesten kender
        f.modtagTekst("LIST");      // få liste over filer på værten

        String indhold = "Indhold af en lille fil med navnet:\ntil.txt";
        f.sendTekst("STOR fil.txt", indhold);    // gem en tekstfil på værten

        indhold = f.modtagTekst("RETR fil.txt"); // hent filen igen
        System.out.println("Fil hentet med indholdet: "+indhold);
    }
}
```

```
modt: 220 ProFTPD 1.2.5rc1 Server [pingo.cv.ihk.dk]
send: USER nordfalk
modt: 331 Password required for nordfalk.
send: PASS adgangskoden
modt: 230 User nordfalk logged in.
...
```

Her følger forklaring på noget af udskriften fra programmet. Linjer, der starter med 'modt:', er tekst modtaget fra værten, mens linjer, der starter med 'send:', kommer fra os.

Listen af kommandoer, som FTP-tjenesten på værten forstår, fås ved at sende HELP:

```
send: HELP
modt: 214-The following commands are recognized (* =>'s unimplemented).
modt: USER PASS ACCT* CWD XCWD CDUP XCUP SMNT*
modt: QUIT REIN* PORT PASV TYPE STRU MODE RETR
modt: STOR STOU* APPE ALLO* REST RNFR RNTD ABOR
modt: DELE MDTM RMD XRMD MKD XMKD PWD XPWD
modt: SIZE LIST NLST SITE SYST STAT HELP NOOP
modt: 214 Direct comments to root@pingo.cv.ihk.dk.
```

Bemærk, at kommandoerne til værten ikke svarer til det, en bruger plejer at skrive i et FTP-program. F.eks svarer kommandoen 'STOR fil.txt', sendt til værten, til at brugeren har skrevet kommandoen 'put fil.txt' for at sende filen fil.txt.

Data (filer og fillister) overføres gennem en separat netværksforbindelse. Linjer, der starter med 'data:', kommer gennem denne forbindelse (som man kalder dataforbindelsen).

Listen over filer og mapper i den aktuelle mappe fås med LIST:

```
send: PASV
modt: 227 Entering Passive Mode (192,168,122,147,4,100).
send: LIST
modt: 150 Opening ASCII mode data connection for file list
data: drwxr-xr-x 3 j      jano      72 Jun 26 2001 GNUstep
data: drwxr-xr-x 5 j      jano      624 Apr  9 12:16 Linux-kursus
data: drwxr-xr-x 6 j      jano      328 Feb  4 14:30 OpenOffice.org
data: -rw-r--r-- 1 j      jano      57350 Dec 12 20:25 løsninger.zip
data: -rw-r--r-- 1 j      jano      16198 Mar  4 15:12 netscape.ps
modt: 226 Transfer complete.
```

Kommandoen 'PASV' klargør en såkaldt 'passiv' dataoverførsel, hvor værten kvitterer med en IP-adresse og et portnummer (delt i to byte, her 4\*256+100, d.v.s. port 1124), som klienten forbinder sig til og sender eller modtager dataene fra.

Herunder er FtpForbindelse-klassen, som du kan benytte i dine egne programmer.

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

/** Denne klasse er af pladshensyn skrevet meget kompakt. Den beste måde at
    forstå den er at prøve den fra et program, f.eks BenytFtpForbindelse, og
    bruge trinvis gennemgang til at følge med i hvordan den fungerer. */

public class FtpForbindelse
{
    private Socket kontrol;
    private PrintStream ud;
    private BufferedReader ind;

    /** Modtager værtens svar, der godt kan løbe over flere linjer. Sidste linje
        er en svarkode på tre cifre, uden en bindestreg '-' på plads nummer 4 */
    private String læsSvar() throws IOException
    {
        while (true) {
            String s = ind.readLine();
            System.out.println("modt: "+s);
            if (s.length()>=3 && s.charAt(3)!='-' && Character.isDigit(s.charAt(0))
                && Character.isDigit(s.charAt(1)) && Character.isDigit(s.charAt(2)))
                return s;    // afslut løkken og returner sidste linje med statuskode
        }
    }

    public String sendKommando(String kommando) throws IOException
    {
        System.out.println("send: "+kommando);
        ud.println(kommando);
        ud.flush();    // sørg for at data sendes til værten før vi læser svar
        return læsSvar();
    }

    public void forbind(String vært, String bruger, String kode) throws IOException
    {
        kontrol = new Socket(vært,21);
        ud = new PrintStream(kontrol.getOutputStream());
        ind = new BufferedReader(new InputStreamReader(kontrol.getInputStream()));
        læsSvar();    // Læs velkomst fra vært
        sendKommando("USER "+bruger);    // Send brugernavn
        sendKommando("PASS "+kode);    // Send adgangskode
    }

    /** Få en forbindelse beregnet til at overføre data (filer) til/fra værten */
    private Socket skafDataforbindelse() throws IOException
    {
        String maskineOgPortnr = sendKommando("PASV");
        StringTokenizer st = new StringTokenizer(maskineOgPortnr, "(,)", false);
        if (st.countTokens() < 7) throw new IOException("Ikke logget ind");
        st.nextToken();    // spring over 5 bidder før portnummer kommer
        st.nextToken();    st.nextToken();    st.nextToken();
        int portNr = 256*Integer.parseInt(st.nextToken())
            + Integer.parseInt(st.nextToken());
        return new Socket(kontrol.getInetAddress(), portNr);    // forbind til porten
    }

    public void sendTekst(String kommando, String data) throws IOException
    {
        Socket df = skafDataforbindelse();
        PrintStream dataUd = new PrintStream( df.getOutputStream() );
        sendKommando(kommando);    // f.eks STOR fil.txt
        dataUd.print(data);
        dataUd.close();
        df.close();
        læsSvar();
    }
}
```

```

public String modtagTekst(String kommando) throws IOException
{
    Socket df = skafDataforbindelse();
    BufferedReader dataInd = new BufferedReader(new InputStreamReader(
                                                df.getInputStream()));
    sendKommando(kommando); // f.eks LIST eller RETR fil.txt
    StringBuilder sb = new StringBuilder();
    String s = dataInd.readLine();
    while (s != null) {
        System.out.println("data: "+s);
        sb.append(s+"\n");
        s = dataInd.readLine();
    }
    dataInd.close();
    df.close();
    læsSvar();
    return sb.toString(); // returnér en streng med de data vi fik fra værten
}
}

```

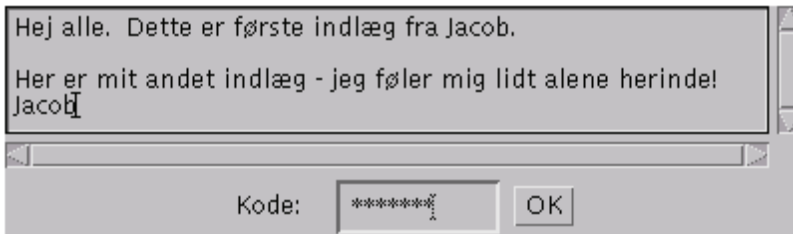
## 16.5.2 Brug af FTP fra en applet

Som beskrevet i Avanceret-afsnittet i kapitel 10, Appletter kan en applet, af sikkerhedsgrunde, kun forbinde sig til den vært, den selv kommer fra og ikke gemme filer på brugerens filsystem.

Det gør det svært at lade en applet huske noget fra gang til gang, hvor siden besøges og umuliggør kommunikation mellem forskellige brugere, med mindre appletten har mulighed for at aktivere et program på værtsmaskinen, der gemmer de nødvendige data. Dette kan desværre også være svært at få lov til hos almindelige internetudbydere.

En løsning er at lave en applet, der kan gemme filer på værtsmaskinen med FTP, hvor brugeren skal angive FTP-adgangskoden (det er nemt for en kompetent person at afkode .class-filen, så det er en *dårlig* ide at lægge adgangskoden i kildeteksten, også selvom den på en eller anden måde er kamufleret). Det forudsætter dog, at man har en begrænset mængde brugere og at man stoler på dem.

Herunder bruges FtpForbindelse til, at lave en lille opslagstavle i en applet.



Når en bruger kommer ind på siden, skal han angive en adgangskode for at se meddelelserne<sup>1</sup>, hvorefter han kan rette i dem og gemme dem igen med FTP.

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

public class FtpOpslagstavleapplet extends JApplet implements ActionListener {
    BorderLayout borderLayout1 = new BorderLayout();
    JTextArea textAreaOpslag = new JTextArea();
    JPanel panelBundlinje = new JPanel();
    JLabel labelKode = new JLabel();
    JPasswordField passwordField = new JPasswordField();
    JButton buttonOk = new JButton();
    FtpForbindelse f;

    public void init() {
        this.setLayout(borderLayout1);
        textAreaOpslag.setText("Opslagstavlen\n\nSkriv koden og tryk OK");
        labelKode.setText("Kode:");
        passwordField.setColumns(8);
        passwordField.setEchoChar('*'); // Skjul adgangskoden
        buttonOk.setText("OK");
        buttonOk.addActionListener(this);
        panelBundlinje.add(labelKode);
        panelBundlinje.add(passwordField);
        panelBundlinje.add(buttonOk);
        this.add(textAreaOpslag, BorderLayout.CENTER);
        this.add(panelBundlinje, BorderLayout.SOUTH);
    }

    public void actionPerformed(java.awt.event.ActionEvent e) {
        try {
            if (f == null) {
                f = new FtpForbindelse();
                String vart = getCodeBase().getHost();
                if (vart.length()==0) vart = "pingo.cv.ihk.dk"; // test-vært
                f.forbind(vart,"jano",new String(passwordField.getPassword()));
                textAreaOpslag.setText(f.modtagTekst("RETR opslag.txt"));
            } else {
                f.sendTekst("STOR opslag.txt",textAreaOpslag.getText());
            }
        } catch (Exception ex) {
            ex.printStackTrace();
            textAreaOpslag.setText("Fejl: "+ex);
        }
    }
}
```

1 Meddelelser kunne også hentes med URL-klassen, sådan at de kunne ses uden adgangskode. Husk da, at protokollerne FTP og HTTP er ret forskellige, blandt andet går FTP ud fra brugerens hjemkatalog, mens hjemmeside-adresser ofte ligger i et underkatalog, der hedder public\_html. F.eks. svarer "PUT public\_html/opslag/opslag.txt" fra FTP-protokollen i HTTP-protokollen til URL'en "http://pingo.cv.ihk.dk/~jano/opslag/opslag.txt" på min maskine.



# 17 Flertrådet programmering

Indhold:

- Forstå tråde
- Eksempel på en flertrådet webserver

Kapitlet forudsættes ikke i resten af bogen.

Forudsætter kapitel 12, Interfaces (kapitel 16, Netværskommunikation og kapitel 9, Grafiske programmer bruges i nogle eksempler).

Når man kommer ud over den grundlæggende programmering, ønsker man tit at lave programmer, som udfører flere opgaver løbende. Det kan f.eks. være et tekstbehandlings-program, hvor man ønsker at kunne gemme eller udskrive i baggrunden, mens brugeren redigerer videre, eller man ønsker løbende stavekontrol samtidig med, at brugeren skriver. Skrivningen må ikke blive forsinket af, at programmet sideløbende forbereder en udskrift eller kontrollerer stavningen. Disse delprocesser (kaldet tråde) har lav prioritet i forhold til at håndtere brugerens input og vise det på skærmen og selvom de midlertidigt skulle gå i stå, skal de andre dele af programmet køre videre.

Et flertrådet program er et program med flere tilsyneladende samtidige programudførelsespunkter (i virkeligheden vil CPU'en skifte meget hurtigt rundt mellem punkterne og udføre lidt af hver).

## 17.1 Princip

Det er ret let at programmere flere tråde i Java. Man opretter en ny tråd med et objekt i konstruktøren:

```
Thread tråd;  
tråd = new Thread(obj);
```

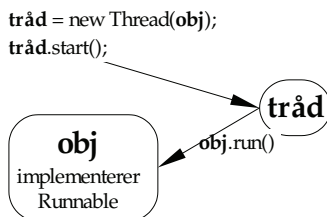
Objektet obj skal implementere interfacet Runnable, f.eks.:

```
public class UdførbartObjekt implements Runnable  
{  
    public void run() // kræves af Runnable  
    {  
        // her starter den nye tråd med at køre  
        // ...  
    }  
}
```

Tråden er nu klar til at udføre run()-metoden på objektet, men den er ikke startet endnu. Man starter den ved at kalde start()-metoden på tråd-objektet:

```
tråd.start();  
// her kører den oprindelige tråd videre, mens den nye tråd kører i obj.run()  
// ...
```

Derefter vil der være to programudførelsespunkter: Et vil være i koden efter kaldet til start() og den anden vil være ved begyndelsen af run()-metoden i objektet.



*En tråd oprettes med et objekt, der implementerer Runnable-interfacet.  
Når start() kaldes på tråden, vil den begynde at udføre run() på objektet.*

### 17.1.1 Eksempel

Herunder definerer vi klassen SnakkesagligPerson. Objekter af typen SnakkesagligPerson kan køre i en tråd (implements Runnable). I konstruktøren angives navnet på personen og hvor lang tid der går, mellem at personen taler.

Når run() udføres, skriver den personens navn + bla bla bla ud, så ofte som angivet.

```

public class SnakkesagligPerson implements Runnable
{
    private String navn;
    private int ventetid;

    public SnakkesagligPerson(String n, int t)
    {
        navn = n;
        ventetid = t;
    }

    public void run()
    {
        for (int i=0; i<5; i++)
        {
            System.out.println(navn+": bla bla bla "+i);
            try { Thread.sleep(ventetid); } catch (Exception e) {} // vent lidt
        }
    }
}

```

Da `Thread.sleep()` kan kaste undtagelser af typen `InterruptedException`, er vi nødt til at indkapsle koden i en try-catch-blok (denne undtagelse forekommer aldrig i praksis).

Vi kan nu oprette en snakkesalig person, der siger noget hvert sekund:

```
SnakkesagligPerson p = new SnakkesagligPerson("Brian",1000);
```

... og en tråd, der er klar til at udføre `p.run()` og lade personen snakke:

```
Thread t = new Thread(p);
```

... og til sidst startes tråden, så personen snakker:

```
t.start();
```

Her ses et samlet eksempel, der opretter 3 snakkesalige personer, Jacob, Troels og Henrik og lader dem snakke i munden på hinanden (i hver sin tråd).

```

public class BenytSnakkesagligePersoner
{
    public static void main(String[] arg)
    {
        SnakkesagligPerson p = new SnakkesagligPerson("Jacob",150); // opret Jacob
        Thread t = new Thread(p); // Ny tråd, klar til at udføre p.run()
        t.start(); // .. Nu starter personen med at snakke...

        p = new SnakkesagligPerson("Troels",400); // opret Troels
        t = new Thread(p);
        t.start();

        // Det kan også gøres meget kompakt:
        new Thread(new SnakkesagligPerson("Henrik",200)).start(); // opret Henrik
    }
}

```

```

Jacob: bla bla bla 0
Troels: bla bla bla 0
Henrik: bla bla bla 0
Jacob: bla bla bla 1
Henrik: bla bla bla 1
Jacob: bla bla bla 2
Troels: bla bla bla 1
Henrik: bla bla bla 2
Jacob: bla bla bla 3
Henrik: bla bla bla 3
Jacob: bla bla bla 4
Troels: bla bla bla 2
Henrik: bla bla bla 4
Troels: bla bla bla 3
Troels: bla bla bla 4

```

Bemærk, at udførelsen af `main()`, der faktisk sker i en fjerde tråd, afsluttes med det samme, men at programmet kører videre, indtil de tre tråde er færdige med deres opgaver; Java fortsætter med at udføre et program, så længe der er tråde, der stadig er aktive, dvs. ikke har returneret fra `run()`.

## 17.2 Ekstra eksempler

Se afsnit 9.4.2, Animationer med en separat tråd, for et eksempel på et grafisk program med flere tråde.

### 17.2.1 En flertrådet webserver

Herunder har vi lavet en flertrådet webserver (sammenlign med Hjemmesidevaert i kapitel 16). For at gøre det nemmere at se, hvad der foregår, lader vi hver anmodning vente i 10 sekunder, før den afslutter.

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

public class Anmodning implements Runnable
{
    private Socket forbindelse;

    Anmodning(Socket forbindelse)
    {
        this.forbindelse = forbindelse;
    }

    public void run()
    {
        try {
            PrintWriter ud = new PrintWriter(forbindelse.getOutputStream());
            BufferedReader ind = new BufferedReader(
                new InputStreamReader(forbindelse.getInputStream()));

            String anmodning = ind.readLine();
            System.out.println("start " + new Date() + " " + anmodning);

            ud.println("HTTP/0.9 200 OK");
            ud.println();
            ud.println("<html><head><title>Svar</title></head>");
            ud.println("<body><h1>Svar</h1>");
            ud.println("Tænker over " + anmodning + "<br>");
            for (int i=0; i<100; i++)
            {
                ud.print(".<br>");
                ud.flush();
                Thread.sleep(100);
            }
            ud.println("Nu har jeg tænkt færdig!</body></html>");
            ud.flush();
            forbindelse.close();
            System.out.println("slut " + new Date() + " " + anmodning);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Når der kommer en anmodning, oprettes et Anmodning-objekt, der snakker med klienten og behandler forespørgslen og en ny tråd knyttes til anmodningen.

```
import java.net.*;
public class FlertraadetHjemmesidevaert
{
    public static void main(String[] arg)
    {
        try {
            ServerSocket vartssokkel = new ServerSocket(8001);

            while (true)
            {
                Socket forbindelse = vartssokkel.accept();
                Anmodning a = new Anmodning(forbindelse);
                new Thread(a).start();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

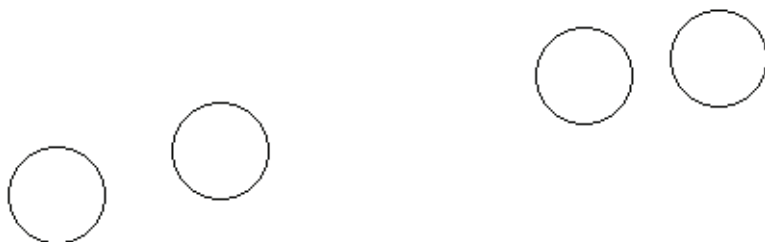
```
start Tue Nov 28 15:37:31 GMT+01:00 2000 GET /xx.html HTTP/1.0
start Tue Nov 28 15:37:38 GMT+01:00 2000 GET /yy.html HTTP/1.0
start Tue Nov 28 15:37:42 GMT+01:00 2000 GET /zz.html HTTP/1.0
slut Tue Nov 28 15:37:42 GMT+01:00 2000 GET /xx.html HTTP/1.0
slut Tue Nov 28 15:37:49 GMT+01:00 2000 GET /yy.html HTTP/1.0
start Tue Nov 28 15:37:50 GMT+01:00 2000 GET /qq.html HTTP/1.0
slut Tue Nov 28 15:37:53 GMT+01:00 2000 GET /zz.html HTTP/1.0
slut Tue Nov 28 15:38:01 GMT+01:00 2000 GET /qq.html HTTP/1.0
```

Programmet er afprøvet ved at åbne adressen <http://localhost:8001/xx.html> hhv. yy, zz og qq.html i en netlæser. Man ser, at anmodningerne xx, yy og zz behandles samtidigt.

## 17.2.2 Et flertrådet program med hoppende bolde

Lad os lave et program med nogle bolde, der hopper rundt. Hver bold kører i sin egen tråd.

Når en bold oprettes, får den i konstruktøren overført start-koordinater og et Graphics-objekt, som den husker. Den opretter og starter en tråd, som kører run()-metoden.



Vi kan så oprette et vindue, få dets Graphics-objekt og oprette nogle bolde med det, som vist herunder:

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

public class FlertraadetGrafik
{
    public static void main(String[] arg)
    {
        JFrame f = new JFrame();
        f.setSize(400,150);
        f.setBackground(Color.WHITE);
        f.setVisible(true);
        Graphics g = f.getGraphics();
        new Bold(g, 0, 0);
        new Bold(g, 50,10);
        new Bold(g,100,50);
        new Bold(g,150,90);
    }
}
```

```

import java.awt.*;

public class Bold implements Runnable
{
    double x, y, fartx, farty;
    Graphics g;

    public Bold(Graphics g1, int x1, int y1)
    {
        g = g1;
        x = x1;
        y = y1;

        fartx = Math.random();
        farty = Math.random();
        Thread t = new Thread(this);
        t.start();
    }

    public void run()
    {
        for (int tid=0; tid<5000; tid++)
        {
            // Tegn bolden over med hvid på den gamle position
            g.setColor(Color.WHITE);
            g.drawOval((int) x, (int) y, 50, 50);

            // Opdater positionen med farten
            x = x + fartx;
            y = y + farty;

            // Tegn bolden med sort på den nye position
            g.setColor(Color.BLACK);
            g.drawOval((int) x, (int) y, 50, 50);

            // ændr boldens hastighed lidt nedad
            farty = farty + 0.1;

            // Hvis bolden er uden for det tilladte område skal den
            // rettes hen mod området
            if (x < 0) fartx = Math.abs(fartx);
            if (x > 400) fartx = -Math.abs(fartx);
            if (y < 0) farty = Math.abs(farty);
            if (y > 100) farty = -Math.abs(farty);

            // Vent lidt
            try { Thread.sleep(10); } catch (Exception e) {};
        }
    }
}

```

## 17.3 Opgaver

- 1) Udvid FlertraadetGrafik med andre figurer end bolde.
- 2) Skriv et program, der udregner primtal (se kapitel 2 for inspiration). Samtidig med, at programmet regner, skal det kunne kommunikere med brugeren og give ham mulighed for at afslutte programmet og udskrive de primtal, der er fundet indtil nu.

## 17.4 Avanceret

### 17.4.1 Nedarvning fra Thread

I stedet for at implementere Runnable kan man også arve fra Thread. På den måde slipper man for at skulle oprette tråden adskilt fra objektet, hvis run()-metode skal udføres.

I eksemplet i afsnit 17.2.1 om den flertrådede webserver ville vi skrive:

```
public class Anmodning extends Thread
```

og i FlertraadetHjemmesidevaert skrive:

```
while (true)
{
    Socket forbindelse = værtssokkel.accept();
    new Anmodning(forbindelse).start();
}
```

Om man foretrækker at arve fra Thread eller implementere Runnable, er en smagssag. Husk dog, at der højst kan arves fra én klasse i Java, mens man kan implementere flere interfaces, så nogen gange kan den eneste mulighed være at implementere Runnable.

### 17.4.2 Synkronisering

Hvad sker der, når to tråde samtidig prøver at kalde System.out.println()? Man kunne være nervøs for, at de ville interferere, sådan at der kunne komme linjer a la

```
Jacob: bHenrik: bla bla bla 3
la bla bla 4
```

hvor Henriks tråd afbryder Jacobs tråd midt i udskrivningen, så teksten bliver "blandet" på en uhensigtsmæssig måde.

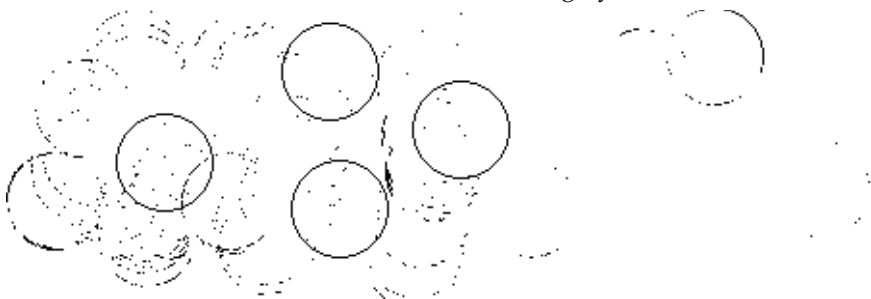
Mere generelt er der brug for at sikre, at kritiske dele af koden kun bliver udført af én tråd ad gangen, sådan at hvis en tråd forsøger at udføre metoder på et objekt, som er "reserveret" af en anden tråd, så kommer den "i kø" og må pænt vente på, at objektet bliver ledigt.

Dette gøres med synchronized-nøgleordet. Metoden println() er således synkroniseret<sup>1</sup>:

```
public synchronized void println(String s)
```

Dette gælder også de andre metoder i System.out, så højst én tråd være aktiv i højst én af disse metoder ad gangen og asynkrone blandinger af systemoutput forhindres således.

I eksemplet med de hoppende bolde kan man se, at programmet nogen gange "glemmer" at tegne en bold over med hvidt, især hvis man klikker og flytter musen over vinduet:



Det skyldes måden, vi sletter boldene på. Her er den kritiske kode i klassen Bold:

```
// Tegn bolden over med hvid på den gamle position
g.setColor(Color.WHITE);
g.drawOval((int) x, (int) y, 50, 50);
```

<sup>1</sup> Egentligt sker det med konstruktionen synchronized (this), som er beskrevet i næste afsnit.

Hvad sker der, hvis en anden tråd sætter farven til sort (fordi den vil tegne en bold) lige efter at denne tråd har sat farven til hvid? Jo, bolden tegnes over med sort i stedet for hvid og det ser ud som om, programmet har glemt at slette billedet.

Under tegneprocessen er vi altså nødt til at reservere `g`, sådan at ingen af de andre tråde uforvarende kommer til at ændre tegnefarven.

### 17.4.3 Synkronisering på objekter og semaforer

Man kan også synkronisere på et andet objekt end `det`, man lige står i (det er egentligt lige meget, hvilket objekt man synkroniserer på, så længe alle tråde bruger det samme):

```
synchronized (objektDerSynkroniseresPå)
{
    // dette udføres kun af een tråd af gangen
    ...
}
```

Hermed fortæller man trådene, at det er *det* objekt, der er "lyssignal" (semafor) for, om der er "grønt lys" til at gå ind i den synkroniserede blok, eller de skal vente på, at en anden tråd har forladt det. Koden:

```
public synchronized void println(String s)
{
    // kode her...
}
```

synkroniserer implicit på objektet selv og kunne faktisk lige så godt skrives som:

```
public void println(String s)
{
    synchronized (this)
    {
        // kode her...
    }
}
```

### 17.4.4 wait() og notify()

Alle objekter har metoderne `wait()` og `notify()`, som kan bruges til kommunikation mellem tråde. Kalder man `wait()` på et objekt, går den kaldende tråd i stå, indtil en (anden) tråd "vækker" denne ved at kalde `notify()` eller `notifyAll()` på samme objekt. Begge tråde skal være synkroniserede på det objekt, der på denne måde bruges som 'lyssignal'.

### 17.4.5 Prioritet

Hver tråd har en prioritet, der kan læses med `getPriority()` og sættes med `setPriority()`. Konstanterne `Thread.MIN_PRIORITY`, `Thread.NORM_PRIORITY` og `Thread.MAX_PRIORITY` angiver hhv. lav, normal og høj prioritet.

### 17.4.6 Opgaver

- 1) Ret i `Bold`-klassen sådan, at vi sikrer, at max. en bold bliver tegnet ad gangen (synkroniser på et objekt, alle bolde har til fælles – for eksempel `Graphics`-objektet `g`).
- 2) Omskriv `Bold` til at arve fra `Thread` i stedet for at implementere `Runnable`.



# 18 Serialisering af objekter

Indhold:

- Hente og gemme objekter i en fil
- Gemme egne klasser – interfacet Serializable og nøgleordet transient

Kapitlet forudsættes i kapitel 19, RMI.

Forudsætter kapitel 15, Datastrømme og filhåndtering (kapitel 12, Interfaces og kapitel 7 om klassemetoder er en fordel).

Når et program afslutter, kan det være, at man ønsker at gemme data til næste gang, programmet starter.

Man kan selvfølgelig skrive programkode, der gemmer og indlæser alle variablerne i de objekter, der skal huskes, men der findes en nemmere måde.

Java har en mekanisme, kaldet *serialisering*, der består i, at objekter kan omdannes til en byte-sekvens (med datastrømmen `ObjectOutputStream`), der f.eks. kan skrives til en fil<sup>1</sup>. Denne bytesekvens kan senere, når man har brug for objekterne igen, deserialiseres (gendannes i hukommelsen med datastrømmen `ObjectInputStream`). Dette kunne f.eks. ske, når programmet starter næste gang.

## 18.1 Hente og gemme objekter

Her er en klasse med to klassemetoder, der henter og gemmer objekter i en fil:

```
import java.io.*;
public class Serialisering
{
    public static void gem(Object obj, String filnavn) throws IOException
    {
        FileOutputStream datastrøm = new FileOutputStream(filnavn);
        ObjectOutputStream objektstrøm = new ObjectOutputStream(datastrøm);
        objektstrøm.writeObject(obj);
        objektstrøm.close();
    }

    public static Object hent(String filnavn) throws Exception
    {
        FileInputStream datastrøm = new FileInputStream(filnavn);
        ObjectInputStream objektstrøm = new ObjectInputStream(datastrøm);
        Object obj = objektstrøm.readObject();
        objektstrøm.close();
        return obj;
    }
}
```

Du kan benytte klassen fra dine egne programmer. Her er et program, der læser en liste fra filen `venner.ser`<sup>2</sup>, tilføjer en indgang og gemmer listen i filen igen.

```
import java.util.*;
public class HentOgGem
{
    public static void main(String[] arg) throws Exception
    {
        ArrayList<String> l;
        try {
            l = (ArrayList<String>) Serialisering.hent("venner.ser");
            System.out.println("Læst: "+l);
        } catch (Exception e) {
            l = new ArrayList();
            l.add("Jacob");
            l.add("Brian");
            l.add("Preben");
            System.out.println("Oprettet: "+l);
        }

        l.add("Ven"+l.size());
        Serialisering.gem(l,"venner.ser");
        System.out.println("Gemt: "+l);
    }
}
```

```
Oprettet: [Jacob, Brian, Preben]
Gemt: [Jacob, Brian, Preben, Ven3]
```

Første gang, programmet kører, opstår der en undtagelse, fordi filen ikke findes. Den fanger vi og føjer "Jacob", "Brian" og "Preben" til listen. Derpå tilføjer vi "Ven3" og gemmer listen.

- 1 Eller netværket for den sags skyld.
- 2 Man bruger ofte filendelsen `.ser` til serialiserede objekter.

Næste gang er uddata:

```
Læst: [Jacob, Brian, Preben, Ven3]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4]
```

Køres programmet igen, ser man, at der hver gang tilføjes en indgang:

```
Læst: [Jacob, Brian, Preben, Ven3, Ven4]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4, Ven5]
Læst: [Jacob, Brian, Preben, Ven3, Ven4, Ven5]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4, Ven5, Ven6]
Læst: [Jacob, Brian, Preben, Ven3, Ven4, Ven5, Ven6]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4, Ven5, Ven6, Ven7]
Læst: [Jacob, Brian, Preben, Ven3, Ven4, Ven5, Ven6, Ven7]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4, Ven5, Ven6, Ven7, Ven8]
```

Hvis nogle af de serialiserede objekter indeholder datafelter, der er referencer til andre objekter, serialiseres disse også. Ovenfor så vi, at hvis man serialiserer en liste, bliver elementerne i listen også serialiseret. Dette gælder også, hvis disse elementer selv indeholder eller er lister og så fremdeles og så kan et helt netværk af objekter, med indbyrdes referencer, blive serialiseret. Man skal derfor være lidt påpasselig i sine egne programmer, det kan være, at man får for meget med.

## 18.2 Serialisering af egne klasser

Det er ikke alle klasser, der må/kan serialiseres. For eksempel giver det ikke mening at serialisere en datastrøm til en forbindelse over netværket (eller bare til en fil). Hvordan skulle systemet genskabe en netværksforbindelse, der har været gemt på harddisken i tre uger? Den anden ende af netværksforbindelsen vil formentlig være væk på det tidspunkt.

### 18.2.1 Interfacet Serializable

Serializable bruges til at markere, at objekter **godt må** serialiseres. Hvis en klasse implementerer Serializable, ved Java, at objekter af denne type godt kan serialiseres.

Derfor implementerer f.eks. ArrayList, Point, String og andre objekter beregnet til at holde data Serializable-interfacet, mens f.eks. FileWriter og Socket ikke gør, da de netop ikke må serialiseres (en Socket repræsenterer jo en netværksforbindelse til et program på en anden maskine og denne forbindelse ville alligevel være tabt, når objektet blev deserialiseret).

Prøver man at serialisere et objekt, der ikke implementerer Serializable, kastes undtagelsen `NotSerializableException` og serialiseringen afbrydes.

I interfacet Serializable er der ikke nogen metoder erklæret og det er derfor helt uforpligtende at implementere. Sådan et interface kaldes også et *markeringsinterface*, da det kun tjener til at markere klasser som, at man kan (eller ikke kan) gøre noget bestemt med dem.

### 18.2.2 Nøgleordet transient

Ud over, at der kan findes objekt-typer, som overhovedet ikke kan serialiseres, kan det også ske, at der er visse dele af et objekts data, man ikke ønsker serialiseret. Hvis et objekt indeholder midlertidige data (f.eks. fortrydelses-information i et tekstbehandlingsprogram), kan man markere de midlertidige datafelter i klassen med nøgleordet **transient**.

### 18.2.3 Versionsnummeret på klassen

Hvis du senere ændrer klassen og prøver at indlæse objekter gemt med den gamle udgave af klassen vil det gå galt! Det kan du løse ved at indsætte et `serialVersionUID` i klassen, som vist nedenfor. Så vil Java indlæse objektet og sætte evt. nye variabler til nul.

## 18.2.4 Eksempel

Eksemplet herunder viser en klasse, der kan serialiseres (implements Serializable), med en transient variabel (tmp). Hvis objektet serialiseres, vil a blive gemt, men tmp vil ikke.

Af bekvemmelighedsgrunde er der også lavet en toString()-metode.

```
import java.io.*;
public class Data implements Serializable
{
    public int a;
    public transient int tmp;    // transiente data bliver ikke serialiseret

    // Vigtigt: Sæt versionsnummer så objekt kan læses selvom klassen er ændret!
    private static final long serialVersionUID = 12345; // bare et eller andet nr.

    public String toString()
    {
        return "Data: a="+a+" tmp="+tmp;
    }
}
```

Her er et program, der læser en liste af Data-objekter, tilføjer et og gemmer den igen:

```
import java.util.*;
public class HentOgGemData
{
    public static void main(String[] arg) throws Exception
    {
        ArrayList<Data> l;
        try {
            l = (ArrayList<Data>) Serialisering.hent("data.ser");
            System.out.println("Læst: "+l);
        } catch (Exception e) {
            l = new ArrayList<Data>();
            System.out.println("Oprettet: "+l);
        }

        Data d = new Data();
        d.a = (int) (Math.random()*100);
        d.tmp = (int) (Math.random()*100);
        l.add(d);

        System.out.println("Gemt: "+l);
        Serialisering.gem(l, "data.ser");
    }
}
```

```
Oprettet: []
Gemt: [Data: a=88 tmp=2]
```

Køres programmet igen, fås:

```
Læst: [Data: a=88 tmp=0]
Gemt: [Data: a=88 tmp=0, Data: a=10 tmp=10]
Læst: [Data: a=88 tmp=0, Data: a=10 tmp=0]
Gemt: [Data: a=88 tmp=0, Data: a=10 tmp=0, Data: a=52 tmp=96]
Læst: [Data: a=88 tmp=0, Data: a=10 tmp=0, Data: a=52 tmp=0]
Gemt: [Data: a=88 tmp=0, Data: a=10 tmp=0, Data: a=52 tmp=0, Data: a=78 tmp=88]
```

Læg mærke til, at den transiente variabel tmp ikke bliver husket fra gang til gang.

## 18.3 Opgaver

- 1) Kør HentOgGemData nogle gange og se, at den husker data i en fil. Kig i venner.ser. Tilføj et ekstra felt til Data.java, oversæt og kør programmet. Hvad sker der? Hvorfor? Slet serialVersionUID og prøv igen. Hvad sker der? Hvorfor?
- 2) Ændr matadorspillet afsnit 5.3 sådan, at felterne og de to spillere gemmes i en fil (serialiseret ned i samme datastrøm), når de 20 runder er gået. Lav mulighed for at indlæse den serialiserede fil, så man kan spille videre på et senere tidspunkt.
- 3) Udvid programmet til, at brugeren angiver filnavnet, der skal hentes/gemmes i.

## 18.4 Avanceret

Mange vælger nutildags at serialisere til XML. Det kan `java.beans.XMLEncoder`, men den gemmer ikke variabler, men *egenskaber* (se afsnit 11.2), dvs du skal lave `get()`- `set()`-metoder for de variabler, du ønsker, skal gemmes. Prøv `XMLEncoder` med `Serialisering.java`:

```
java.beans.XMLEncoder objektstrøm = new java.beans.XMLEncoder(datastrøm);
...
java.beans.XMLDecoder objektstrøm = new java.beans.XMLDecoder(datastrøm);
```

Se også: <http://java.sun.com/javase/6/docs/api/java/beans/XMLEncoder.html>.

### 18.4.1 Serialisere det samme objekt flere gange

Serialisering virker ved, at `ObjectOutputStream` husker, hvilke objekter der allerede er gået gennem datastrømmen. Hvert objekt serialiseres kun én gang, så hvis det samme objekt kommer igen, registreres blot et ID, der henviser til det allerede serialiserede objekt.

Det giver en u hensigtsmæssig bivirkning, hvis man serialiserer et objekt, derpå ændrer det og derefter serialiserer det igen, som vist nedenfor:

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

public class GemDetSammeIgen {
    public static void main(String[] arg) throws Exception {
        ArrayList l;
        try {
            FileInputStream datastrøm = new FileInputStream("venner2.ser");
            ObjectInputStream objektstrøm = new ObjectInputStream(datastrøm);
            l = (ArrayList) objektstrøm.readObject();
            System.out.println("Læst1: "+l);

            l = (ArrayList) objektstrøm.readObject();
            System.out.println("Læst2: "+l);

            objektstrøm.close();
        } catch (Exception e) {
            l = new ArrayList();
            l.add("Jacob");
            System.out.println("Oprettet: "+l);
        }
        l.add("Ven"+l.size());

        FileOutputStream datastrøm = new FileOutputStream("venner2.ser");
        ObjectOutputStream objektstrøm = new ObjectOutputStream(datastrøm);

        objektstrøm.writeObject(l);
        System.out.println("Gemt1: "+l);

        // ændr nu l og serialiser igen
        l.add("EkstraVen"+l.size());
        // l = (ArrayList) l.clone(); // løsning1: opret et andet objekt
        // objektstrøm.reset(); // løsn2: nulstil liste af allerede skrevne obj
        objektstrøm.writeObject(l); // ellers bliver den nye tilstand af l ikke gemt
        System.out.println("Gemt2: "+l);
        objektstrøm.close();
    }
}
```

```
Oprettet: [Jacob]
Gemt1: [Jacob, Ven1]
Gemt2: [Jacob, Ven1, EkstraVen2]
```

Næste gang er uddata:

```
Læst1: [Jacob, Ven1]
Læst2: [Jacob, Ven1]
Gemt1: [Jacob, Ven1, Ven2]
Gemt2: [Jacob, Ven1, Ven2, EkstraVen3]
```

Og næste gang:

```
Læst1: [Jacob, Ven1, Ven2]
Læst2: [Jacob, Ven1, Ven2]
Gemt1: [Jacob, Ven1, Ven2, Ven3]
Gemt2: [Jacob, Ven1, Ven2, Ven3, EkstraVen4]
```

Man ser, at den ændrede tilstand af listen (at EkstraVennerne er kommet til) ikke gemmes. ObjectOutputStream tænker "dét ArrayList-objekt har jeg allerede serialiseret, så det behøver jeg ikke kigge nærmere på, jeg henviser bare til dets ID".

Der er to løsninger på problemet:

- 1) Send et nyt objekt. Det kunne man f.eks gøre ved at oprette en kopi af listen:  
`l = (ArrayList) l.clone();`
- 2) Fortæl datastrømmen, at den skal nulstille listen af allerede serialiserede objekter:  
`objektstrøm.reset();`

Løsning 1) er faktisk den mest effektive, da det kun er ArrayList-objektet (og en EkstraVen), der bliver skrevet igen, mens løsning 2) skriver samtlige data igen.

Hvad enten man vælger det ene eller andet, bliver uddata nu som forventet:

```
Oprettet: [Jacob]
Gemt1: [Jacob, Ven1]
Gemt2: [Jacob, Ven1, EkstraVen2]
```

```
Læst1: [Jacob, Ven1]
Læst2: [Jacob, Ven1, EkstraVen2]
Gemt1: [Jacob, Ven1, EkstraVen2, Ven3]
Gemt2: [Jacob, Ven1, EkstraVen2, Ven3, EkstraVen4]
```

```
Læst1: [Jacob, Ven1, EkstraVen2, Ven3]
Læst2: [Jacob, Ven1, EkstraVen2, Ven3, EkstraVen4]
Gemt1: [Jacob, Ven1, EkstraVen2, Ven3, EkstraVen4, Ven5]
Gemt2: [Jacob, Ven1, EkstraVen2, Ven3, EkstraVen4, Ven5, EkstraVen6]
```

## 18.4.2 Selv styre serialiseringen af en klasse

Klasser, der har brug for speciel håndtering under serialisering og deserialisering, kan definere både metoderne `writeObject()` og `readObject()`. Da vil disse blive kaldt og dermed slås den normale serialisering fra og man får fuld kontrol over, hvordan det foregår.

Eksempelvis kunne vi have skrevet Data-klassen som:

```
import java.io.*;
public class DataMedStyretSerialisering implements Serializable
{
    public int a;
    public int tmp;    // tmp behøves ikke mere at erklæres transient,
                      // da vi selv sørger for serialiseringen

    public String toString()
    {
        return "Data: a="+a+" tmp="+tmp;
    }

    private void writeObject(ObjectOutputStream out) throws IOException
    {
        out.writeInt(a);    // gem kun a, ikke tmp
    }

    private void readObject(ObjectInputStream in) throws IOException,
        ClassNotFoundException
    {
        a = in.readInt();
    }
}
```

Nu behøver vi ikke mere erklære `tmp` på nogen speciel måde, da vi selv skriver koden, der foretager serialiseringen.

Ønsker man også at bruge standardmekanismen for serialisering, kan den fra `writeObject()` kaldes med `out.defaultWriteObject()` (og tilsvarende `in.defaultReadObject` fra `readObject()`). Et trick er da, at erklære alle felter, man selv vil sørge for serialiseringen af, for transient og så selv kan sørge for dem i `writeObject()` og `readObject()`.

# 19 RMI - objekter over netværk

Indhold:

- Forstå principperne i RMI
- Kalde metoder i fjerne objekter

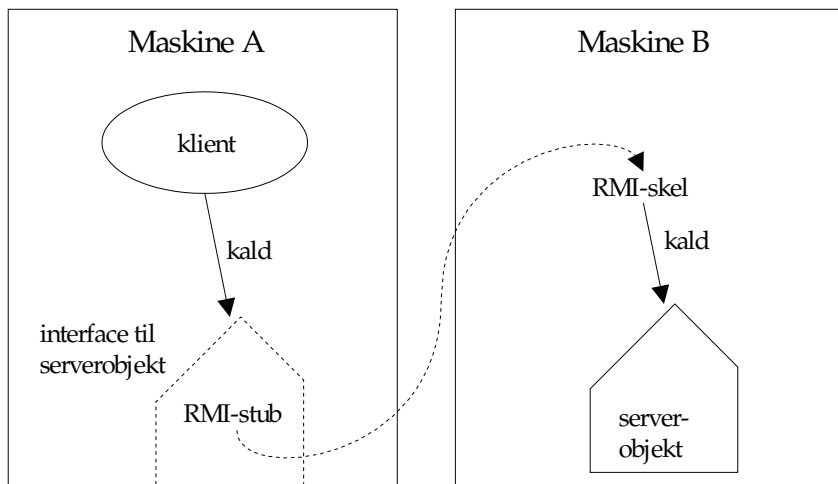
Kapitlet forudsættes ikke i resten af bogen.

Forudsætter kapitel 12, Interfaces, kapitel 18, Serialisering og kendskab til netværk.

Med RMI (Remote Method Invocation) kan man arbejde med objekter, der eksisterer i en anden Java virtuel maskine (ofte på en anden fysisk maskine), *som om de var lokale objekter*.

## 19.1 Principper

Herunder er tegnet, hvad der sker, når en klient på maskine A laver et kald til et serverobjekt (værts-objekt), der er i maskine B.



Serverobjektet findes slet ikke på maskine A, i stedet er der en såkaldt *RMI-stub*, der repræsenterer det. Når der sker et kald til RMI-stubben på maskine A, sørger den for at transportere kaldet og alle parametre til maskine B, hvor serverobjektet bliver kaldt, som om det var et lokalt kald. Serverobjektets svar bliver transporteret tilbage til RMI-stubben, der returnerer det til klienten.

Denne proces foregår helt automatisk og er usynlig for klienten såvel som serverobjektet.

RMI benytter serialisering til at transportere parametre og returværdi mellem maskinerne, så man skal huske, at alle objekter, der sendes over netværket, skal implementere *Serializable*-interfacet og at variabler, der ikke skal overføres, skal mærkes med nøgleordet *transient*.

Der skal være defineret et interface (kaldet fjerninterfacet) til de metoder på serverobjektet, som skal være tilgængelige for klienten. Serverobjekt skal implementere dette interface.

## 19.2 I praksis

Lad os forestille os, at serveren har et konto-objekt, hvor man kan overføre penge, spørge om saldo og få bevægelserne. Disse metoder skal være tilgængelige over netværket, så vi definerer et fjerninterface til kontoen (her kaldt *KontoI*):

```
import java.util.ArrayList;

public interface KontoI extends java.rmi.Remote
{
    public void overførsel(int kroner) throws java.rmi.RemoteException;
    public int saldo() throws java.rmi.RemoteException;
    public ArrayList bevægelser() throws java.rmi.RemoteException;
}
```

Fjerninterfacet skal arve fra interfacet *java.rmi.Remote* og alle metoder skal kunne kaste undtagelsen *java.rmi.RemoteException*.



## 19.2.1 På serversiden

På serversiden skal vi implementere Konto-interfacet og programmere den funktionalitet, der skjuler sig bag det i et serverobjekt, der skal arve fra `UnicastRemoteObject`. Klassenavnet ender normalt på `Impl` (for at vise, at det er implementationen af fjerninterfacet).

```
import java.util.ArrayList;
import java.rmi.server.UnicastRemoteObject;

public class KontoImpl extends UnicastRemoteObject implements KontoI
{
    public int saldo;
    public ArrayList bevægelser;

    public KontoImpl() throws java.rmi.RemoteException
    {
        // man starter med 100 kroner
        saldo = 100;
        bevægelser = new ArrayList();
    }

    public void overførsel(int kroner)
    {
        saldo = saldo + kroner;
        String s = "Overførsel på "+kroner+" kr. Ny saldo er "+saldo+" kr.";
        bevægelser.add(s);
        System.out.println(s);
    }

    public int saldo()
    {
        System.out.println("Der spørges om saldoen. Den er "+saldo+" kr.");
        return saldo;
    }

    public ArrayList bevægelser()
    {
        System.out.println("Der spørges på alle bevægelser.");
        return bevægelser;
    }
}
```

Nu skal vi oprette et serverobjekt og registrere vores tjeneste under et navn i RMI:

```
import java.rmi.Naming;
public class Kontoserver
{
    public static void main(String[] arg) throws Exception
    {
        // Enten: Kør programmet 'rmiregistry' fra mappen med .class-filerne, eller:
        java.rmi.registry.LocateRegistry.createRegistry(1099); // start i server-JVM

        KontoI k = new KontoImpl();
        Naming.rebind("rmi://localhost/kontotjeneste", k);
        System.out.println("Kontotjeneste registreret.");
    }
}

Kontotjeneste registreret.
...
```

Programmet afslutter ikke, men venter på, at noget henvender sig, for at bruge tjenesten.

For at registreringen kan foregå, skal der køre en RMI-navnetjeneste, der holder styr på, hvilke tjenester, der udbydes under hvilke navne og formidler kontakten til dem. Det er et lille program, der hedder `rmiregistry` og som normalt lytter på port 1099. Det nemmeste er dog at starte navnetjenesten i samme JVM som serverobjektet kører, som vist ovenfor<sup>1</sup>.

---

<sup>1</sup> Vil du hellere køre `rmiregistry` i et separat vindue, så se øvelsen sidst i kapitlet.

## 19.2.2 På klientsiden

På klientsiden skal vi slå serverobjektet op i RMI-tjenesten og derefter bruge det objekt, vi får retur, som om det var serverobjektet selv (i virkeligheden er det RMI-stubben):

```
import java.rmi.Naming;

public class Kontoklient
{
    public static void main(String[] arg) throws Exception
    {
        KontoI k =(KontoI) Naming.lookup("rmi://localhost/kontotjeneste");
        k.overførsel(100);
        k.overførsel(50);
        System.out.println( "Saldo er: "+ k.saldo() );
        k.overførsel(-200);
        k.overførsel(51);
        System.out.println( "Saldo er: "+ k.saldo() );
        java.util.ArrayList bevægelser = k.bevægelser();

        System.out.println( "Bevægelser er: "+ bevægelser );
    }
}
```

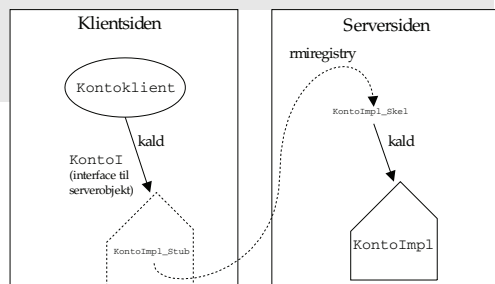
```
Saldo er: 250
Saldo er: 101
Bevægelser er: [Overførsel på 100 kr. Ny saldo er 200 kr., Overførsel på 50 kr.
Ny saldo er 250 kr., Overførsel på -200 kr. Ny saldo er 50 kr., Overførsel på 51
kr. Ny saldo er 101 kr.]
```

Sammen med Kontoklient skal ligge fjerninterfacet KontoI.

Mens kontoklienten kører, kommer der følgende uddata fra Kontoserver:

```
Overførsel på 100 kr. Ny saldo er 200 kr.
Overførsel på 50 kr. Ny saldo er 250 kr.
Der spørges om saldoen. Den er 250 kr.
Overførsel på -200 kr. Ny saldo er 50 kr.
Overførsel på 51 kr. Ny saldo er 101 kr.
Der spørges om saldoen. Den er 101 kr.
Der spørges på alle bevægelser.
```

På figuren til højre ses de enkelte klassers funktioner. Bemærk at Java automatisk genererer stub- og skel-klasserne.



## 19.3 Opgaver

Start serveren og kør klienten et par gange.

### 19.3.1 Server og klient to forskellige steder

Når ovenstående fungerer, så ret i Kontoklient, sådan at klienten kontakter en anden maskine. Er værtsmaskinens IP-nummer f.eks. 192.168.1.42, retter du i Kontoklient til:

```
KontoI k =(KontoI) Naming.lookup("rmi://192.168.1.42/kontotjeneste");
```

### 19.3.2 Starte separat 'rmiregistry'

Fjern kaldet til `java.rmi.registry.LocateRegistry.createRegistry(1099)` fra Kontoserver og start i stedet `rmiregistry` i et separat vindue. Husk at det skal kende definitionen af serverobjekter og evt. klasser, der overføres med RMI, så `CLASSPATH` skal sættes eller `rmiregistry` skal startes fra den samme mappe, som bytekoden findes. Ligger `.class`-filerne i `C:\jbp\project\mitProjekt\classes`, kunne du åbne en DOS-kommandoprompt<sup>1</sup> og skrive:

```
cd C:\jbp\project\mitProjekt\classes
rmiregistry
```

<sup>1</sup> I Windows fås en DOS-prompt ved at klikke i menuen Start, vælge 'Kør...' og skrive 'cmd'.

# 20 JDBC - databaseadgang

Indhold:

- Få kontakt til en database fra Java
- Kommandoer til en database
- Forespørgsler til en database

Kapitlet forudsættes ikke i resten af bogen.

Forudsætter kapitel 14, Undtagelser og lidt kendskab til databaser og databasesproget SQL (Structured Query Language) og at du har en fungerende database, som du ønsker adgang til fra Java.

Adgang til en database fra Java sker gennem et sæt klasser, der under et kaldes JDBC (Java DataBase Connectivity) – en platformuafhængig pendant til Microsoft ODBC (Open DataBase Connectivity). Klasserne ligger i pakken `java.sql`, så kildetekstfiler, der arbejder med databaser, skal starte med:

```
import java.sql.*;
```

## 20.1 Kontakt til databasen

At få kontakt til databasen er måske det sværeste skridt. Det består af to led:

1) Indlæse databasedriveren

2) Etablere forbindelsen

Indlæsning af driveren sker, ved at bede systemet indlæse den pågældende klasse, der derefter registrerer sig selv i JDBC-systemets driver-manager. Er det f.eks. en Oracle-database, skriver man:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Ofte skal man have en jar-fil (et Java-ARKiv, en samling klasser pakket i zip-formatet) med en driver fra producenten. Nyeste drivere kan findes på <http://java.sun.com/jdbc>.

For en Oracle-database hedder filen `classes12.zip` og passer til en bestemt udgave af Oracle-databasen. I JDeveloper, er den som standard med i projektets klassesti. Ellers skal den føjes til CLASSPATH (i JBuilder 2006 under Project Properties, Paths, Required Libraries).

Herefter kan man oprette forbindelsen med (for en Oracle-database):

```
Connection forb = DriverManager.getConnection(
    "jdbc:oracle:thin:@oracle.cv.ihk.dk:1521:student", "brugernavn", "adgangskode");
```

Første parameter er en URL til databasen. Den består af en protokol ("`jdbc`"), underprotokol ("`oracle`") og noget mere, der afhænger af underprotokollen. I dette tilfælde angiver det, at databasen ligger på maskinen `oracle.cv.ihk.dk` port 1521 og hedder `student`.

Anden og tredje parameter er brugernavn og adgangskode.

### 20.1.1 JDBC-ODBC-broen under Windows

Med Java under Windows følger en standard JDBC-ODBC-bro med, så man kan kontakte alle datakilder defineret under ODBC. Denne driver indlæses med:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Når forbindelsen oprettes, angiver man den ønskede datakildes navn. Husk, at datakildens navn (her "`datakilde1`") først skal være defineret i Windows' Kontrolpanel under ODBC:

```
Connection forb = DriverManager.getConnection("jdbc:odbc:datakilde1");
```

Bemærk, at ODBC er en ret langsom protokol. Har du brug for bedre ydelse bør du finde en driver, der kommunikerer direkte med databasen, i stedet for at bruge JDBC-ODBC.

## 20.2 Kommunikere med databasen

Når vi har en forbindelse, kan vi oprette et "statement"-objekt, som vi kan sende kommandoer og forespørgsler til databasen med:

```
Statement stmt = forb.createStatement();
```

Der kan opstå forskellige undtagelser af typen `SQLException`, der skal fanges.

### 20.2.1 Kommandoer

SQL-kommandoer, der ikke giver et svar tilbage i form af data, som `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE` og `DROP TABLE`, sendes med `executeUpdate()`-metoden.

Her opretter vi f.eks. tabellen "kunder" og indsætter et par rækker:

```
import java.sql.*;
public class SimpeltDatabaseeksempel
{
    public static void main(String[] arg) throws Exception
    {
        // Udskift med din egen databasedriver og -URL
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection forb = DriverManager.getConnection("jdbc:odbc:datakilde1");
        Statement stmt = forb.createStatement();

        stmt.executeUpdate("create table KUNDER (NAVN varchar(32), KREDIT float) ");

        stmt.executeUpdate("insert into KUNDER values('Jacob', -1799)");
        stmt.executeUpdate("insert into KUNDER values('Brian', 0)");
    }
}
```

Oftest har man data gemt i nogle variabler. Så skal man sætte en streng sammen, der giver den ønskede SQL-kommando:

```
String navn = "Hans";
int kredit = 500;

// indsæt data fra variablerne navn og kredit
stmt.executeUpdate("insert into KUNDER values('"+navn+"', '"+kredit+"')");
```

## 20.2.2 Forespørgsler

SQL-forespørgslen SELECT udføres med metoden `executeQuery()`:

```
ResultSet rs = stmt.executeQuery("select NAVN, KREDIT from KUNDER");
```

Den giver et `ResultSet`-objekt, der repræsenterer svaret på forespørgslen (for at få alle kolonner kunne vi også skrive "select \* from KUNDER"). Data hentes fra objektet således:

```
while (rs.next())
{
    String navn = rs.getString("NAVN");
    double kredit = rs.getDouble("KREDIT");
    System.out.println(navn+" "+kredit);
}
```

Man kalder altså `next()` for at få næste række i svaret, læser de enkelte celler ud fra kolonnenavnene (eller kolonnenumrene, regnet fra 1 af), hvorefter man går videre til næste række med `next()` osv. Når `next()` returnerer false, er der ikke flere rækker at læse.

## 20.3 Adskille database- og programlogik

Det er en god idé at indkapsle databearbejdet ét sted, f.eks. i en klasse, sådan at resten af programmet kan fungere, selvom databasens adresse eller struktur skulle ændre sig.

Ofte vil man have en klasse for hver tabel i databasen, sådan at hvert objekt kommer til at svare til en række. Herunder har vi lavet klassen `Kunde`, svarende til tabellen `KUNDER`:

```
public class Kunde
{
    String navn;
    double kredit;

    public Kunde(String n, double k)
    {
        navn = n;
        kredit = k;
    }

    public String toString()
    {
        return navn+": "+kredit+" kr.";
    }
}
```

Klassen, der varetager forbindelsen til databasen, bør have metoder, der svarer til de kommandoer og forespørgsler, resten af programmet har brug for. Hvis databasen ændrer sig, er det kun denne klasse, der skal rettes i:

```
import java.sql.*;
import java.util.*;

public class Databaseforbindelse
{
    private Statement stmt;

    public Databaseforbindelse() throws Exception
    {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection forb = DriverManager.getConnection(
            "jdbc:oracle:thin:@oracle.cv.ihk.dk:1521:student", "brugernavn", "kode");
        stmt = forb.createStatement();
    }

    public void sletAlleData() throws SQLException
    {
        stmt.execute("truncate table KUNDER");
    }

    public void opretTestdata() throws SQLException
    {
        try { // hvis tabellen allerede eksisterer opstår der en SQL-udtagelse
            stmt.executeUpdate(
                "create table KUNDER (NAVN varchar(32), KREDIT float) ");
        } catch (SQLException e) {
            System.out.println("Kunne ikke oprette tabel: "+e);
        }
        stmt.executeUpdate("insert into KUNDER values('Jacob', -1799)");
        stmt.executeUpdate("insert into KUNDER values('Brian', 0)");
    }

    public void indsæt(Kunde k) throws SQLException
    {
        stmt.executeUpdate("insert into KUNDER (NAVN,KREDIT) values('"
            + k.navn + "', " + k.kredit + ")");
    }

    public ArrayList<Kunde> hentAlle() throws SQLException
    {
        ArrayList<Kunde> alle = new ArrayList<Kunde>();
        ResultSet rs = stmt.executeQuery("select NAVN, KREDIT from KUNDER");
        while (rs.next())
        {
            // brug kolonneindeks i stedet for kolonnenavn
            Kunde k = new Kunde( rs.getString(1), rs.getDouble(2));
            alle.add(k);
        }
        return alle;
    }
}
```

Klassen lader kalderen om at håndtere de mulige undtagelser. Det er fornuftigt, da det også er kalderen, der skal fortælle fejlen til brugeren og evt. beslutte, om programmet skal afbrydes.

Her er et program, der bruger Databaseforbindelse. Først opretter det forbindelsen og henter alle poster, dernæst sletter det alt og indsætter en enkelt post. Hvis der opstår en fejl, udskrives "Problem med database" og programmet afbrydes.

```
import java.util.*;

public class BenytDatabaseforbindelse
{
    public static void main(String[] arg)
    {
        try {
            Databaseforbindelse dbf = new Databaseforbindelse();

            dbf.opretTestData(); // fjern hvis tabellen allerede findes
            ArrayList<Kunde> l = dbf.hentAlle();
            System.out.println("Alle data: "+l);
            dbf.sletAlleData();

            dbf.indsæt( new Kunde("Kurt",1000) );
            System.out.println("Alle data nu: "+ dbf.hentAlle());

        } catch(Exception e) {
            System.out.println("Problem med database: "+e);
            e.printStackTrace();
        }
    }
}

Alle data: [Jacob: -1799.0 kr., Brian: 0.0 kr.]
Alle data nu: [Kurt: 1000.0 kr.]
```

## 20.4 Opgaver

- 1) Ændr SimpletDatabaseeksempel, så den også laver en SQL-forespørgsel.
- 2) Udvid Databaseforbindelse, så den kan søge efter en kunde ud fra kundens navn (antag, at navnet er en primærnøgle, så der ikke kan være flere kunder med samme navn).
- 3) Udvid Databaseforbindelse, så den kan give en liste med alle kunder med negativ kredit.
- 4) Lav et program, der holder styr på en musiksamling vha. en database. Databasen skal have tabellen UDGIVELSER med kolonnerne år, navn, gruppe og pladeselskab. Opret en tilsvarende klasse, der repræsenterer en Udgivelse (int år, String navn, String gruppe, String pladeselskab). Lav en passende Databaseforbindelse og et (evt. grafisk) program, der arbejder med musikdatabasen.
- 5) Ret databasen i forrige opgave til at have tabellen UDGIVELSER med kolonnerne år, navn og gruppeID, tabellen GRUPPER med kolonnerne gruppeID, navn, pladeselskab. Hvordan skal Databaseforbindelse ændres? Behøves der blive ændret i resten af programmet? Hvorfor?
- 6) Udvid programmet, så hver gruppe har en genre som f.eks. rock, tekno, klassisk (tabellen GRUPPER udvides med genreID og tabellen GENRER oprettes med kolonnerne genreID og navn).

## 20.5 Avanceret

Det følgende er rart (men ikke nødvendigt) at vide, når man arbejder med databaser.

### 20.5.1 Forpligtende eller ej? (commit)

Normalt er alle SQL-kommandoer gennem JDBC *automatisk forpligtende* (eng. auto-committing): Ændringerne kan ikke annulleres på et senere tidspunkt.

Ønsker man at slå det fra, kan man kalde `setAutoCommit(false)` på forbindelsen. Derefter er transaktioner (SQL-kommandoer) ikke forpligtende, d.v.s. ændringerne er ikke synlige for andre brugere og de kan annulleres ved at kalde `rollback()`. Når man ønsker at transaktionerne skal træde i kraft, kalder man `commit()`. Først herefter er transaktionerne endeligt udført på databasen og dermed synlige for andre brugere. Eksempel:

```
try {
    forb.setAutoCommit(false);
    Statement stmt = forb.createStatement();

    stmt.execute("truncate table KUNDER");
    stmt.executeUpdate("insert into KUNDER values('Jacob', -1799)");
    stmt.executeUpdate("insert into KUNDER values('Brian', 0)");

    // flere transaktioner ...
    System.out.println("Alt gik godt, gør ændringerne forpligtende");
    forb.commit();
}
catch (Exception e)
{
    e.printStackTrace();
    System.out.println("Noget gik galt med databasen! Annullerer ændringerne...");
    forb.rollback();
}
finally
{
    // Husk at sætte auto-commit tilbage, af hensyn til andre transaktioner
    forb.setAutoCommit(true);
}
```

Denne facilitet er nyttig i forbindelse med en serie af transaktioner, der enten alle skal gennemføres eller alle skal annulleres.

### 20.5.2 Optimering

Et programs ydeevne begrænses næsten altid af en eller højst to faktorer (flaskehalse).

Inden du læser videre, så bemærk lige en generel regel omkring optimering af ydelse:

---

**Under programmeringen er det i de fleste tilfælde spild af tid, at tænke på optimering – det er bedre, at koncentrere sig om funktionaliteten og, når programmet er skrevet næsten helt færdigt, identificere flaskehalsene og optimere disse dele af koden**

---

Optimering af kørselstiden bør dog ske tidligere, hvis:

- Det forventes at forårsage større strukturelle ændringer i programmet. Optimeringer der forventes at indvirke på programmets struktur, bør ske i design-fasen, inden programmeringen påbegyndes.
- Det forventes at give kortere udviklingstid (fordi de løbende afprøvninger af programmet kan udføres hurtigere).



## På forhånd forberedt SQL

I eksemplerne vist indtil nu, har strengen med SQL-kommandoen skulle fortolkes, hver gang kaldet foretages. I stedet kan man bruge metoden `prepareStatement()`, hvor man angiver SQL-kommandoen én gang og derefter kan udføre kommandoen flere gange:

```
PreparedStatement indsæt, hent;  
indsæt = forb.prepareStatement("insert into KUNDER values(?,?)");  
hent = forb.prepareStatement("select NAVN, KREDIT from KUNDER");
```

Ovennævnte forberedelser sker typisk kun én gang under opstart af programmet. Derefter kan de forberedte kommandoer bruges igen og igen med forskellige parametre:

```
indsæt.setString(1, "Jacob")  
indsæt.setInt(2, -1799)  
indsæt.execute();  
  
indsæt.setString(1, "Brian")  
indsæt.setInt(2, 0)  
indsæt.execute();  
  
ResultSet rs = hent.executeQuery();  
...
```

## Kalde lagrede procedurer i databasen

Større databaser understøtter procedurer lagret i databasen ('stored procedures'). De kan udføres hurtigt, da databasen på forhånd kan optimere, hvordan SQL-kaldene skal foregå.

En lagret procedure kan kaldes med et `CallableStatement` (her forestiller vi os, at der på forhånd er oprettet procedurerne **indsaetkunde** og **hentkunder** i databasen):

```
CallableStatement indsætP = forb.prepareCall("call indsaetkunde(?, ?)");  
CallableStatement hentP = forb.prepareStatement("?= hentkunder");
```

Resten af arbejdet foregår som med `PreparedStatement`:

```
indsætP.setString(1, "Jacob")  
indsætP.setInt(2, -1799)  
indsætP.execute();  
indsætP.setString(1, "Brian")  
indsætP.setInt(2, 0)  
indsætP.execute();  
  
ResultSet rs = hentP.executeQuery();  
...
```

## Brug af en optimal databasedriver

JDBC-drivere findes i fire typer og det kan være afgørende for ydeevnen, hvilken driver man bruger.

- Type 1 er JDBC-ODBC-broen. Dette er den langsomste, da den kører gennem ODBC og derudover er den platformsspecifik (Windows).
- Type 2 er drivere, hvor JDBC-laget kalder funktioner skrevet i f.eks. maskinkode, C eller C++ til den specifikke platform.
- Type 3 er platformsuafhængige (ren Java-) drivere, der benytter en databaseuafhængig kommunikationsprotokol.
- Type 4 er platformsuafhængige (ren Java-) drivere, der er skrevet til at kommunikere med en specifik database.

De hurtigste drivere er type 2, men ofte er type 4-drivere næsten lige så hurtige.

En liste med over hundrede tilgængelige drivere kan findes på <http://java.sun.com/jdbc>.

## 20.5.3 Metadata

Metadata betyder egentligt "uden for data", men kunne lige så godt forstås som "data om data". Der findes metadata på to niveauer: Metadata om databasen (DatabaseMetaData) og metadata om svaret på en forespørgsel (ResultSetMetaData).

Metadata om databasen fås ved at kalde `getMetaData()` på forbindelses-objektet:

```
DatabaseMetaData dmd = forb.getMetaData();
```

Dette objekt har metoder, der giver information om:

- Producenten  
`getDatabaseProductName()`, `getDatabaseProductVersion()`, `getDriverName()`, `getDriverVersion()`, ...
- Databasens begrænsninger og hvad den understøtter  
`getMaxRowSize()`, `getMaxStatementLength()`, `getMaxTableNameLength()`, ...
- Databasens indhold (skemaer, tabeller, ...) – i form af et `ResultSet` med en liste:  
`getSchemas()`, `getCatalogs()`, `getTableTypes()`, `getTables()`, `getColumns()`, ...

Metadata om svaret på en forespørgsel fås ved at kalde `getMetaData()` på `ResultSet`-objektet. Dette objekt har metoder, som beskriver svaret, bl.a. `getColumnCount()`, der giver antal kolonner i svaret og `columnName()`, der giver navnet på en kolonne:

```
Statement stmt = forb.createStatement();
ResultSet rs = stmt.executeQuery("select * from KUNDER");
ResultSetMetaData rsmd = rs.getMetaData();

int antalKolonner = rsmd.getColumnCount();
System.out.println("Der er "+antalKolonner+" kolonner i tabellen KUNDER.");

for (int i=0; i<antalKolonner; i++)
{
    System.out.println("Kolonne nr. "+i+" har navn:"+rsmd.getColumnName(i));
}
```

## 20.5.4 Opdatering og navigering i ResultSet-objekter

`ResultSet` har metoder, som understøttes af de fleste JDBC-drivere, hvor til at bevæge sig aktivt rundt i svaret og endda opdatere databasen gennem svaret. Det gøres med f.eks.:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("select NAVN, KREDIT from KUNDER");
```

Derefter kan man navigere rundt i `ResultSet`-objektet med f.eks.:

```
rs.absolute(3);      // går til 3. række i svaret (regnet fra 1 af)
rs.previous();       // går en række tilbage (modsatte af next())
rs.first();          // går til starten af svaret (svarende til rs.absolute(1))
rs.relative(3);      // går 3 rækker frem, d.v.s. til 4. række
int r = rs.getRow(); // giver hvilken række vi nu er i (her returneres 4)
```

Man kan ændre i data med f.eks.:

```
rs.updateString("NAVN", "Jakob"); // ændrer kundens navn til Jakob
rs.updateRow();                   // opdaterer rækken i databasen

rs.moveToInsertRow();             // flyt til speciel indsættelses-række
rs.updateString("NAVN", "Søren"); // sæt navn
rs.updateDouble("KREDIT", 1000);  // sæt kredit
rs.insertRow();                  // indsæt rækken i databasen
rs.moveToCurrentRow();            // gå væk fra speciel indsættelsesrække
```

Derudover findes `cancelRowUpdates()`, der annullerer opdateringer i en række, `deleteRow()`, der sletter den aktuelle række fra både svaret og databasen, `refreshRow()`, der opfrisker `ResultSet`-objektet med de nyeste data.

# 21 Avancerede klasser

Indhold:

- Nøgleordene abstract og final
- Indre klasser, herunder lokale klasser og anonyme klasser
- Brug af indre klasser og anonyme klasser til at lytte efter hændelser
- Brug af anonyme klasser til at oprette bl.a. tråde i en håndevending

Indre klasser er en forudsætning for at forstå den måde, mange værktøjer laver kode, til at håndtere hændelser.

Forudsætter kapitel 5, Nedarvning, kapitel 12, Interfaces (og kapitel 13, Hændelser, og kapitel 17, Flertrådet programmering, for at forstå nogle af eksemplerne) og et ønske om at vide mere om emner, som først bliver relevante, når man laver større programmer.

## 21.1 Nøgleordet **abstract**

Noget der er erklæret **abstract** er ikke implementeret og skal defineres i en nedarvning. Det skrives i kursiv i UML-notationen.

### 21.1.1 Abstrakte klasser

En abstrakt klasse erklæres således

```
public abstract class X
{
    public void a()
    {
        //..
    }
}
```

Det er ikke tilladt at oprette objekter fra en abstrakt klasse

```
public static void main(String[] arg)
{
    X x = new X(); // ulovligt! X er abstrakt
}
```

I stedet skal man arve fra klassen

```
public class Y extends X
{
}
```

og lave objekter fra den nedarvede klasse:

```
public static void main(String[] arg)
{
    X x;           // lovligt

    x = new Y();    // lovligt, Y er ikke abstrakt
}
```

Basisklasserne for IO-systemet, `InputStream` og `OutputStream` er abstrakte, fordi programøren altid skal bruge en mere konkret klasse, f.eks. `FileInputStream` (se afsnit 15.5.2).

I Matador-eksemplet afsnit 5.3 kunne vi have erklæret klassen `Felt` for abstrakt, da der ikke måtte oprettes 'nøgne' `Felt`-objekter, men kun nedarvinger som `Start` og `Helle`.

Det er lovligt (og nyttigt i visse tilfælde) at have variabler af en abstrakt klasse (det svarer til, at det er lovligt og nyttigt at have variabler af en interface-type). F.eks. er det i Matador-eksemplet ret vigtigt at have en `Felt`-variabel, der kan referere til alle slags felter.

### 21.1.2 Abstrakte metoder

En metode erklæret **abstract** har et metodehoved, men ingen krop. Den kan kun erklæres i en abstrakt klasse

```
public abstract class X
{
    public abstract void a();
}
```

Nedarvede klasser skal definere de abstrakte metoder (eller også selv være abstrakte)

```
public class Y extends X
{
    public void a()
    {
        //..
    }
}
```

I Matador-eksemplet afsnit 5.3 kunne vi have erklæret metoden `landet()` for abstrakt og på den måde sikre, at den blev defineret i alle nedarvinger.

```
/** Superklassen for alle matadorspillels felter */
public abstract class Felt
{
    String navn;

    public void passeret(Spiller sp)
    {
        System.out.println(sp.navn+" passerer "+navn);
    }

    public abstract void landet(Spiller sp);
}
```

Hvis vi glemte at definere `landet()` i en nedarving (eller måske kom til at stave metodenavnet forkert), ville oversætteren stoppe os og komme med en fejlmeddelelse.

En klasse, hvor alle metoder er defineret abstrakte, kaldes en *ren abstrakt klasse*. Sådant en klasse har i de fleste tilfælde nogenlunde samme rolle som et interface<sup>1</sup>.

## 21.2 Nøgleordet final

Noget, der er erklæret **final**, kan ikke ændres. Både variabler, metoder og klasser kan erklæres final.

### 21.2.1 Variabler erklæret final

En variabel, der er erklæret **final**, kan ikke ændres, når den først har fået en værdi.

```
public class X
{
    public final int a=10;

    //..
    // forbudt: a=11;
}
```

Herover kan `a`'s værdi ikke ændres i den efterfølgende kode.

Det kan lette overskueligheden at vide, hvilke variabler, der er konstante. Desuden udføres programmet lidt hurtigere.

Foran en objekt- eller klassevariabel bestemmer **final** ikke adgang/synlighed, men kan bruges sammen med **public**, **protected** og **private**.

**final** kan også bruges på lokale variabler (hvor **public**, **protected** og **private** aldrig kan bruges):

```
public static void main(String[] arg)
{
    final ArrayList l = new ArrayList();

    //l = new ArrayList(); // ulovligt! l kan ikke ændres.
}
```

Bemærk: Når vi arbejder med objekter, er variablerne jo referencer til objekterne. En variabel erklæret **final** kan ikke ændres til at referere til et andet objekt, men objektet kan godt få ændret sin indre tilstand, f.eks. gennem et metodekald:

```
l.add("Hans"); // lovligt, l refererer stadig til samme objekt
```

---

<sup>1</sup> Forskellen på en ren abstrakt klasse og et interface er, at klassen godt kan indeholde variabler, mens et interface kun har konstanter. Derudover kan man implementere flere interfaces, men kun arve fra én klasse.

## 21.2.2 Metoder erklæret final

En metode erklæret final kan ikke tilsidesættes i en nedarving.

```
public class X
{
    public final void a()
    {
        // ..
    }
}
```

```
public class Y extends X
{
    public void a() // ulovligt! a() er final
    {
        //..
    }
}
```

Den virtuelle maskine kan optimere final metoder, så kald til dem sker en smule hurtigere.

## 21.2.3 Klasser erklæret final

En klasse erklæret final må man overhovedet ikke arve fra (og alle dens metoder bliver dermed final).

```
public final class X
{
    // ..
}
```

```
public class Y extends X // ulovligt! X er final
{
}
```

## 21.3 Indre klasser

Indre klasser er mindre "hjælpeklasser" defineret inde i en anden klasse. Dette afsnit handler om de forskellige måder, at definere indre klasser på og de forhold, der her gør sig gældende.

Siden Java version 1.1 har der eksisteret 3 slags indre klasser:

- (Almindelige) indre klasser
- Lokale klasser
- Anonyme klasser

Der er flere fordele ved at benytte indre klasser (visse undtagelser bliver forklaret sidst i afsnittet):

- Den indre klasse er knyttet til den ydre klasse og kan kun anvendes i denne.  
Man behøver derfor ikke bekymre sig for sammenhængen med resten af programmet. Det kan give et mere overskueligt program at lægge klasser, der alligevel har meget stærk binding (er meget afhængige af hinanden) inden i hinanden (indkapsling).
- Den indre klasse kan arbejde direkte på den ydre classes variabler og metoder, også de private.  
Det skyldes, at et objekt af en indre klasse, altid hører til et objekt af den ydre klasse.

## 21.4 Almindelige indre klasser

En almindelig indre klasse er en klasse, der erklæres på linje med objektvariabler og metoder:

```
public class YdreKlasse
{
    class IndreKlasse
    {
    }
}
```

Programkoden i den indre klasse kan anvende alle den ydre classes variabler og metoder – også de private. Den indre klasse er knyttet til et objekt af den ydre klasse.

## 21.4.1 Eksempel - Linjetegning

Man benytter ofte indre klasser i forbindelse med at lytte efter hændelser. Her kommer Linjetegning-eksemplet fra kapitel 13 igen, men hvor vi lader en indre klasse lytte efter musklik.

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

public class LinjetegningIndre extends JPanel
{
    // Selv private variabler har den indre klasse adgang til
    private Point trykpunkt;
    private Point slippunkt;

    public LinjetegningIndre()
    {
        Linjelytter lytter = new Linjelytter();
        this.addMouseListener(lytter);
    }

    // En indre klasse
    class Linjelytter implements MouseListener
    {
        public void mousePressed (MouseEvent event)
        {
            trykpunkt = event.getPoint();           // sæt variablen i det ydre objekt
        }

        public void mouseReleased (MouseEvent event)
        {
            slippunkt = event.getPoint();
            repaint();                             // kald det ydre objekts metode
        }

        public void mouseClicked (MouseEvent event) {} // kræves af MouseListener
        public void mouseEntered (MouseEvent event) {} // kræves af MouseListener
        public void mouseExited (MouseEvent event) {} // kræves af MouseListener
    }
    // slut på indre klasse

    // en metode i den ydre klasse
    public void paintComponent (Graphics g)
    {
        super.paintComponent(g);                  // tegn først baggrunden på panelet
        g.drawString("1:"+trykpunkt+" 2:"+slippunkt,10,10);
        if (trykpunkt != null && slippunkt != null)
        {
            g.setColor(Color.BLUE);
            g.drawLine(trykpunkt.x, trykpunkt.y, slippunkt.x, slippunkt.y);
        }
    }
}
```

Læg mærke til, at den indre klasse uden videre har adgang til den ydre classes variabler og metoder (sammenlign med Linjetegning fra afsnit 13.2).

```
import javax.swing.*;
public class BenytLinjetegningIndre
{
    public static void main(String[] arg)
    {
        JFrame vindue = new JFrame( "LinjetegningIndre" );
        vindue.add( new LinjetegningIndre() );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        vindue.setSize(500,150);
        vindue.setVisible(true);
    }
}
```



## 21.5 Lokale indre klasser

En lokal klasse er defineret i en blok programkode ligesom en lokal variabel.

```
public class YdreKlasse
{
    public void metode()
    {
        // ...

        class Lokalklasse
        {
            // metoder og variabler her ...
        }

        Lokalklasse objektAfLokalklasse = new Lokalklasse();

        // ...
    }
}
```

Lokale klasser er kun synlige og anvendelige i den blok, hvor de er defineret. Ligesom lokale variabler er de ikke synlige uden for metoden (og nøgleordene public, private, protected og static foran klassen har derfor ingen mening).

Lokale klasser kan benytte alle variabler og metoder, der er synlige inden for blokken. Dog skal lokale variabler, der er erklæret i den omgivende metode, være erklæret final, dvs. være konstante, før de kan bruges i den lokale klasse.

Lokale klasser bruges ret sjældent (men de er gode at forstå, før man går videre til anonyme klasser).

Nedenstående er et eksempel på en lokal klasse, der benytter variabler defineret i den ydre klasse:

```
public class YdreKlasseMedLokalklasse
{
    private int a1 = 1;           // Objektvariabler behøver ikke være final

    public void prøvLokaltObjekt(final int a2) // Bemærk: final
    {
        final int a3 = 3;           // Bemærk: final

        class Lokalklasse {           // definér lokal klasse
            int a4 = 4;
            public void udskriv()
            {
                System.out.println( a4 );
                System.out.println( a3 );
                System.out.println( a2 );
                System.out.println( a1 );
            }
        } // slut på lokal klasse

        Lokalklasse lokal = new Lokalklasse(); // opret lokalt objekt fra klassen
        lokal.udskriv();
    }

    public static void main(String[] arg){
        YdreKlasseMedLokalklasse ydre = new YdreKlasseMedLokalklasse();
        ydre.prøvLokaltObjekt(2);
    }
}
4
3
2
1
```

## 21.6 Anonyme indre klasser

En anonym klasse er en klasse uden navn, som der oprettes et objekt ud fra der, hvor den defineres (altså en unavngiven lokal klasse).

```
public class YdreKlasse
{
    public void metode()
    {
        // ... programkode for metode

        X objektAfAnonymKlasse = new X()
        {
            void metodeIAnonymKlasse()
            {
                // programkode
            }
            // flere metoder og variabler i anonym klasse
        };

        // mere programkode for metode
    }
}
```

Lige efter new angives det, hvad den anonyme klasse arver fra, eller et interface, der implementeres (i dette tilfælde X). Man kan ikke definere en konstruktør til en anonym klasse (den har altid standardkonstruktøren). Angiver man nogen parametre ved new X(), er det parametre til superklassens konstruktør.

Fordelen ved anonyme klasser er, at det tillades på en nem måde at definere et specialiseret objekt præcis, hvor det er nødvendigt – det kan være meget arbejdsbesparende.

### 21.6.1 Eksempel - filtrering af filnavne

Følgende program udskriver alle javafiler i den aktuelle mappe. Det sker ved at kalde list()-metoden på et File-objekt og give det et FilenameFilter-objekt som parameter.

Interfacet FilenameFilter har metoden accept(File dir, String filnavn), som afgør, om en fil skal tages med i listen (se evt. Javadokumentationen).

```
import java.io.*;
public class FilnavnfiltreringMedAnonymKlasse
{
    public static void main(String[] arg)
    {
        File f = new File( "." );           // den aktuelle mappe
        FilenameFilter filter;

        filter = new FilenameFilter()
        { // En anonym klasse
            public boolean accept( File f, String s ) // En metode
            {
                return s.endsWith( ".java" ); // svar true hvis fil ender på .java
            }
        } // slut på klassen
        ; // slut på tildelingen filter = new ...

        // brug objektet som filter i en liste af et antal filer
        String[] list = f.list( filter );

        for (int i=0; i<list.length; i=i+1) System.out.println( list[i] );
    }
}
```

```
YdreKlasseMedLokalKlasse.java
FilnavnfiltreringMedAnonymKlasse.java
LinjetegningAnonym.java
AnonymeTraade.java
A.java
BenytIndreKlasser.java
```

## 21.6.2 Eksempel - Linjetegning

Udviklingsværktøjer benytter ofte anonyme klasser i forbindelse med at lytte efter hændelser. Her er Linjetegning-eksemplet igen, hvor vi bruger en anonym klasse som lytter (sml. eksemplet i 21.4.1).

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

public class LinjetegningAnonym extends JPanel
{
    private Point trykpunkt;
    private Point slippunkt;

    public LinjetegningAnonym()
    {
        this.addMouseListener(
            new MouseListener()
        {
            public void mousePressed (MouseEvent event)
            {
                trykpunkt = event.getPoint();
            }

            public void mouseReleased (MouseEvent event)
            {
                slippunkt = event.getPoint();
                repaint();
            }

            public void mouseClicked (MouseEvent event) {}
            public void mouseEntered (MouseEvent event) {}
            public void mouseExited (MouseEvent event) {}
        } // slut på anonym klasse
    ); // slut på kald til addMouseListener()

    System.out.println("Anonymt lytter-objekt oprettet");
}

public void paintComponent(Graphics g)
{
    super.paintComponent(g);           // tegn først baggrunden på panelet
    g.drawString("1:"+trykpunkt+" 2:"+slippunkt,10,10);
    if (trykpunkt != null && slippunkt != null)
    {
        g.setColor(Color.BLUE);
        g.drawLine(trykpunkt.x, trykpunkt.y, slippunkt.x, slippunkt.y);
    }
}
```

Her er en klasse, der viser panelet:

```
import javax.swing.*;

public class BenytLinjetegningAnonym
{
    public static void main(String[] arg)
    {
        JFrame vindue = new JFrame( "LinjetegningAnonym" );
        vindue.add( new LinjetegningAnonym() );
        vindue.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        vindue.setSize(500,150);
        vindue.setVisible(true);
    }
}
```

## 21.6.3 Eksempel - tråde

Her gennemløber vi en løkke, der i hvert gennemløb opretter et Runnable-objekt fra en anonym klasse og en tråd, der kører på det. Objekterne får hvert sit nummer fra 1 til 5, som de udskriver 20 gange, før de slutter. For at få trådene til at kæmpe lidt om processortiden tæller de til 1.000.000 mellem hver udskrivning.

```
public class AnonymeTraade
{
    public static void main(String[] arg)
    {
        for (int i=1; i<=5; i=i+1)
        {
            // n bruges i den anonyme klasse
            final int n = i;

            Runnable r = new Runnable()
            {
                public void run()
                {
                    for (int j=0; j<20; j=j+1)
                    {
                        System.out.print(n);

                        // Lav noget, der tager tid
                        int x = 0;
                        for (int k=0; k<1000000; k=k+1) x=x+k;
                    }
                    System.out.println("Færdig med "+n+".");
                }
            };

            Thread t = new Thread(r);
            t.start();
        }
    }
}

111122221223311332441114433221144332211442255115544332115544332211Færdig med 1.
54442233332233Færdig med 2.
544335544Færdig med 3.
54Færdig med 4.
55555555Færdig med 5.
```

Man ser, hvordan objekt nummer 1, der blev startet først, også er det første, der afslutter.

Bemærk igen, hvordan den (anonyme) lokale klasse kun kan benytte lokale variabler i den omgivende metode, hvis de er erklæret final.

## 21.7 Resumé

De tre slags indre klasser er:

Navn	Beskrivelse
(almindelig) indre klasse	En klasse defineret i en klasse på linje med objektvariabler og -metoder. Er knyttet til et objekt af den ydre klasse (hvis den ikke er static).
lokal klasse	En klasse defineret i en metode i en klasse på linje med lokale variabler. Kan benytte de lokale variabler og parametre (defineret final) i den programblok, de er defineret i. Er kun kendt inden for blokken.
anonym klasse	Unavngivet klasse med samtidig oprettelse af et objekt.

Herunder er en beskrivelse af, hvor og hvordan de erklæres og bruges:

```
public class YdreKlasse
{
    class IndreKlasse
    {
        void metodeIIndreKlasse()
        { // programkode...
        }
        // flere metoder og variabler i indre klasse
    }

    public void ydreMetode()
    {
        // ... programkode for ydreMetode(), f.eks.
        IndreKlasse objektAfIndreKlasse = new IndreKlasse();
        objektAfIndreKlasse.metodeIIndreKlasse();

        class LokalKlasse
        {
            void metodeILokalKlasse()
            { // programkode...
            }
            // flere metoder og variabler i lokal klasse
        }

        // ... mere programkode for ydreMetode(), f.eks.
        LokalKlasse objektAfLokalKlasse = new LokalKlasse();
        objektAfLokalKlasse.metodeILokalKlasse();

        InterfaceEllerKlasse objektAfAnonymKlasse = new InterfaceEllerKlasse()
        {
            void metodeIAnonymKlasse()
            { // programkode...
            }
            // flere metoder og variabler i anonym klasse
        }

        // ... mere programkode for ydreMetode(),
    }
}
```

## 21.8 Opgaver

- 1) Tag TegnbareObjekter.java fra kapitel 12 og lav (i init()-metoden) fem forskellige objekter, der implementerer Tegnbar-interface (brug anonyme klasser). De fem objekter skal have forskellig måde at reagere på tegn() og sætPosition().
- 2) Læs afsnit eller dokumentationen til Comparator-interface i pakken java.util. Lav tre Comparator-objekter (vha. anonyme klasser), der sorterer strenge hhv. alfabetisk, omvendt alfabetisk og alfabetisk efter andet tegn i strengene. Lav en liste (ArrayList) med ti strenge og test din sortering med Collections.sort(liste, Comparator-objekt).

## 21.9 Avanceret

Man kan have indre klasser i indre klasser. Her er et (ikke specielt praktisk) eksempel:

```
public class A
{
    public String navn ="Klasse A";

    class B
    {
        public String navn="Klasse B";

        class C
        {
            public String navn="Klasse C";

            public void skriv()
            {
                System.out.println(navn);
                System.out.println(this.navn);
                System.out.println(C.this.navn);
                System.out.println(B.this.navn);
                System.out.println(A.this.navn);
            }
        }
    }
}
```

Er der sammenfald mellem navnene på nogle af variablerne i de ydre og indre klasser, kan der skelnes, ved at foranstille *this* og klassenavnet.

```
public class BenytIndreKlasser
{
    public static void main(String[] arg) {
        A a = new A();
        A.B b = a.new B();
        A.B.C c = b.new C();
        c.skriv();
    }
}
```

---

```
Klasse C
Klasse C
Klasse C
Klasse B
Klasse A
```

Bemærk den særlige måde, man opretter indre klasser på i forbindelse med en forekomst af den ydre klasse (det ligner næsten, at man kalder *new*-metoden på et ydre objekt for at skabe det indre objekt).

### 21.9.1 public, protected og private på en indre klasse

Som det ses ovenfor, kan indre klasser godt bruges uden for den ydre klasse. Adgang til brug af en indre klasse styres på samme måde som en variabel eller metode med *public* (alle har adgang), *protected* (adgang fra nedarvinger og alle klasser fra samme pakke), *in-genting* (kun adgang fra samme pakke) og *private* (kun adgang fra den ydre klasse).

Havde vi f.eks. rettet "class B" til "public class B" og tilsvarende med C, kunne BenytIndreKlasser have været i en anden pakke. Skrev vi *private* i stedet, skulle *main*-metoden være i klassen A. Se også afsnit 6.9 om pakker.

### 21.9.2 static på en indre klasse

En almindelig indre klasse er normalt knyttet til et objekt af den ydre klasse (ligesom en objektvariabel). Erklærer man den indre klasse *static*, mistes tilknytningen til det ydre objekt og den indre klasse kan derfor ikke mere anvende de ydre objektvariabler og objekt-metoder.

# 22 Objektorienteret analyse og design

Indhold:

- Analyse: Finde vigtige ord, brugssituationer, aktivitetsdiagrammer og skærm-billeder
- Design: Kollaborationsdiagrammer og klassediagrammer
- Eksempel: Skitse til et yatzy-spil

Kapitlet giver idéer til, hvordan en problemstilling kan analyseres, før man går i gang med at programmere.

Forudsætter kapitel 5, Nedarvning.

Når et program udvikles, sker det normalt i fem faser:

- 1) Kravene til programmet bliver afdækket.
- 2) Analyse – hvad det er for ting og begreber, programmet handler om.
- 3) Design – hvordan programmet skal laves.
- 4) Programmering.
- 5) Afprøvning.

Traditionel systemudvikling bygger vandfaldsmodellen: De fem faser udføres en efter en, sådan at en ny fase først påbegyndes, når den forrige er afsluttet. Hver fase udmøntes i et dokument, som de senere faser skal følge og som kan bruges til dokumentation af systemet.

Dette er i skarp modsætning til den måde, som en selvlært umiddelbart programmerer på. Her blandes faserne sammen i hovedet på programmøren, som skifter mellem dem, mens han programmerer. Resultatet er ofte et program, der bærer præg af ad-hoc-udbygninger og som er svært at overskue og vedligeholde – selv for programmøren selv<sup>1</sup>.

Den bedste udviklingsmetode findes nok et sted mellem de to ekstremer. Der dukker f.eks. altid nye ting op under programmeringen, som gør, at man må ændre sit design. Omvendt er det svært at programmere uden et gennemtænkt design.

Derfor er det ikke en god ide at bruge alt for lang tid på at lave fine tegninger og diagrammer – en blyantskitse er lige så god. Det er indholdet, der tæller, og ofte laver man om i sit design flere gange, inden programmet er færdigt. Dette gælder især, hvis man er i gang med at lære at programmere.

Nu til dags stræber mange mod en arbejdsform, hvor man på ca. fire uger (en iteration) udfører et "mini-vandfald" hvor alle de fem faser indgår. Det, man derved lærer, bruges så til at forfine kravene, analysen, designet og programmet, i næste iteration. Denne udviklingsform kaldes "Unified Process".

Dette kapitel viser gennem et eksempel (et Yatzy-spil) en grov skitse til objektorienteret analyse og design (forkortet OOAD). Det er tænkt som inspiration til, hvordan man kunne gribe sit eget projekt an ved at følge de samme trin.

## 22.1 Krav til programmet

Vi skal lave et Yatzy-spil for flere spillere. Der kan være et variabelt antal spillere, hvoraf nogle kan være styret af computeren. Computerspillerne skal have forskellige strategier (dum/tilfældig, grådig, strategisk), der vælges tilfældigt.

Krav skal afdække, kommunikere og huske behovene. Lav en nummereret kravliste:

- K1) Yatzy skal understøtte 2-6 spillere
- K2) Yatzy skal tjekke at reglerne er overholdt
- K3) Yatzy skal kunne styre nogle af spillerne
- K4) Yatzy skal lagre spillene når de er afsluttet
- K5) Yatzy skal kunne vise en hiscore-liste
- K6) Yatzy skal (kun) være på dansk

---

1 En helt anden arbejdsform, der prøver at gå med den umiddelbare impuls: at programmere med det samme, er *ekstremprogrammering* (XP). I denne form beskriver man først kodens ønskede opførsel i form af testtilfælde, der kan afprøves automatisk gennem hele forløbet. Så programmerer man to og to foran samme tastatur, indtil testene er opfyldt. Til sidst lægger man sig fast på et fornuftigt design og omstrukturerer programmet til at passe med det valgte design.



## 22.2 Objektorienteret analyse

Analysen skal beskrive, hvad det er for ting og begreber, programmet handler om. Analysefasens formål er, at afspejle virkeligheden mest muligt.

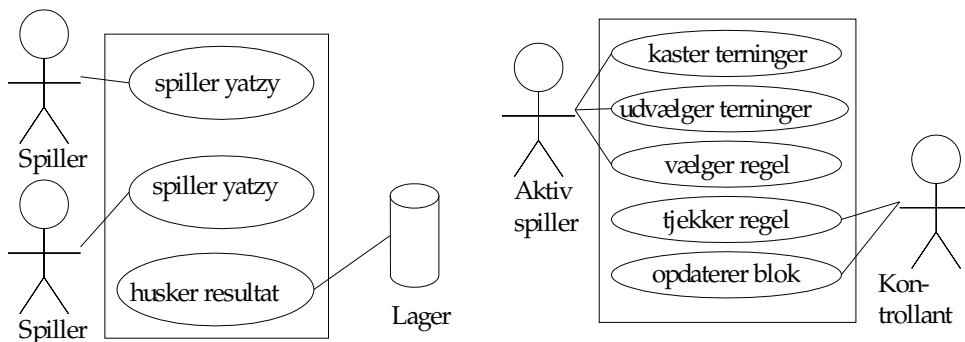
### 22.2.1 Skrive vigtige ord op

Det kan være en hjælp først at skrive alle de navneord (i ental) eller ting op, man kan komme i tanke om ved problemet. Ud for hver ting kan man notere eventuelle egenskaber (ofte tillægsord) og handlinger (ofte udsagnsord), knyttet til tingen.

- Yatzysspil – antal spillere
- Terning – værdi, kaste, holde
- Raflebæger – kombination, ryste, holde
- Blok – skrive spillernavn på, skrive point på
- Spiller – navn, type (computer/menneske)
- Computerspiller – strategi (dum/tilfældig, grådig, strategisk)
- Menneskespiller
- Regel (eller mulighed eller kriterium) – opfyldt, brugt, antal point
- Lager – hiscore

### 22.2.2 Brugssituationer - Hvem Hvad Hvor

Brugssituationer (eng.: Use Case) beskriver en samling af aktører og hvilke brugssituationer de deltager i. Man starter helt overordnet og går mere og mere i detaljer omkring hver brugssituation. Herunder to brugssituationer. Til venstre ses et meget overordnet, der beskriver to spillere og lageret som aktører. Til højre ses brugssituationen for en tur.



I stedet for diagrammer kan man også blot beskrive brugssituation efter et skema:

Primær aktør	<i>Brugeren eller systemet, det handler om i denne situation</i>
Interessenter	<i>Andre aktører, der har sekundær interesse i denne situation</i>
Startsituation	<i>Hvad skal være opfyldt før brugssituationen forekommer</i>
Slutsituation	<i>Resultatet af at brugssituationen udføres</i>
Hovedscenarie	<i>Sekvensen af handlinger, der fører aktøren fra startsituation frem til slutsituationen</i>
Afviigelser	<i>Afviigelser og undtagelser fra hovedscenariet</i>
Åbne spørgsmål	<i>Problemstillinger som der først tages stiling til på et senere tidspunkt (hvis man anvender en udviklingsmodel som f.eks. Unified Process hvor analysen gentages senere i udviklingsforløbet)</i>

Her er et eksempel:

### Brugssituation "SpilEnTur" i Yatzy

Primær aktør	Brugeren, hvis tur det er
Interessenter	Systemet
Startsituation	Det er brugerens tur
Slutsituation	Bruger har valgt felt i blokken og alle pointtal er opdateret
Hovedscenarie	<ol style="list-style-type: none"><li>1. Bruger trykker på "kast terninger"</li><li>2. Terninger, der ikke er holdt får en ny tilfældig værdi</li><li>3. Bruger vælger terninger der skal holdes</li><li>4. (punkt 1-3 gentages maks. 3 gange)</li><li>5. Systemet viser en liste af mulige felter i blokken</li><li>6. Bruger vælger et felt</li></ol>
Afvielser	2a. Alle terninger er holdt: Advarselsvindue dukker op: "Vil du afslutte kastene?"
Åbne spørgsmål	-

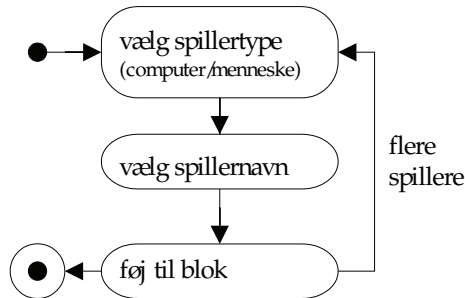
Man kan hævde, at Yatzy-spillet er på grænsen til at være for simpelt til at lave brugssituationer. Her er et mere realistisk eksempel på en brugssituation i et kurvetegningsprogram:

### Brugssituation "TegnGraf" i et kurvetegningsprogram

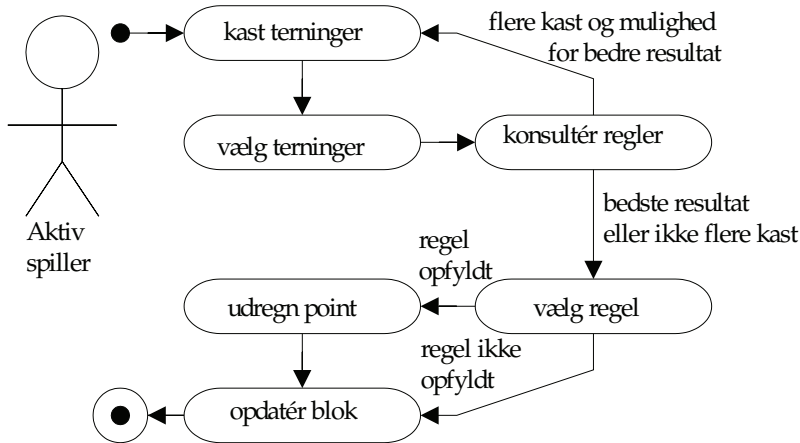
Primær aktør	Bruger
Interessenter	-
Startsituation	Et vindue skal være åbent og en funktion skal være tastet ind
Slutsituation	Grafens støttepunkter er beregnet og grafen vises i vinduet
Hovedscenarie	<ol style="list-style-type: none"><li>1. Brugeren aktiverer tegning af graf</li><li>2. Programmet åbner dialog og beder om mindste og største x-værdi</li><li>3. Bruger angiver start- og slutværdier og trykker OK</li><li>4. Programmet lukker dialog og beregner 50 funktionsværdien i 50 støttepunkter og bestemmer mindste og største funktionsværdi</li><li>5. Programmet tegner grafen som rette linjer mellem støttepunkterne</li></ol>
Afvielser	<p>3a: Brugeren afbryder indtastning</p> <ol style="list-style-type: none"><li>1. Programmet sletter eventuelle indtastninger og lukker dialog</li></ol> <p>3b: Brugeren har indastet andet end tal i startværdi eller slutværdi</p> <p>3c: Brugerens startværdi er mindre end slutværdien</p> <ol style="list-style-type: none"><li>1. Programmet kommer med en fejlmeddelelse og sletter indtastningerne</li></ol>
Åbne spørgsmål	Hvordan tegnes uendelig? F.eks. $1/x$ ? Skal man kunne se flere grafer samtidig?

## 22.2.3 Aktivitetsdiagrammer

Aktivitetsdiagrammer beskriver den rækkefølge, som adfærdsmønstre og aktiviteter foregår i. Eksempel: Aktiviteten "definere deltagere i spillet":



Herunder ses et diagram for spillets gang, "en tur":



## 22.2.4 Skærbilleder

Hvis skærbilleder er en væsentlig del af ens program, er det en god hjælp at tegne de væsentligste, for at gøre sig klart, hvilke elementer programmet skal indeholde.

Disse kan med fordel designes direkte med et Java-udviklingsværktøj. Herved opnår man en ide om, hvad der er muligt, samtidig med at den genererede kode ofte (men ikke altid!) kan genbruges i programmeringsfasen. Normalt kommer der en klasse for hvert skærbillede, så man kan også med det samme give dem sigende navne.

Når programmet startes, skal vælges 2-6 spillere, hvoraf nogle kan være computerspillere:

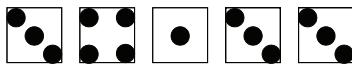
Navn:

☐ Computer  
☒ Menneske

*TilføjSpillervindue*

Under spillet skiftes spillerne til at få tur. For menneske-spillerne dukker dette billede op:

Søren



Hold ☒ ☐ ☐ ☒ ☒

Kast!
Færdig

*Turvindue*

Man kan holde på terningerne ved at klikke på afkrydsningsfelterne.

Når spilleren er færdig (efter max 3 kast), skal han/hun vælge, hvilken regel der skal bruges, ved at klikke på den i blok-vinduet:

	Jacob	Søren
Ettere	4	
Toere		
<b>Treere</b>		9
etc...		
<hr/>		
Sum		
Bonus		
Et par		
etc...		
<hr/>		
Sum		

*Blokvindue*

## 22.3 Objektorienteret design

Designets formål er at beskrive, hvordan programmet skal implementeres. I denne fase skal man bl.a. identificere de vigtigste klasser i systemet og lede efter ligheder mellem dem med henblik på nedrivning og genbrug.

### 22.3.1 Design af klasser

Et udgangspunkt for, hvordan man designer klasser, er at objekterne i programmet skal svare nogenlunde til de virkelige, oftest fysiske objekter fra problemstillingen:

---

**Navneord (substantiver) i ental bliver ofte til klasser**

**Klassenavne skal altid være i ental**

**Udsagnsord (verber) bliver ofte til metoder**

---

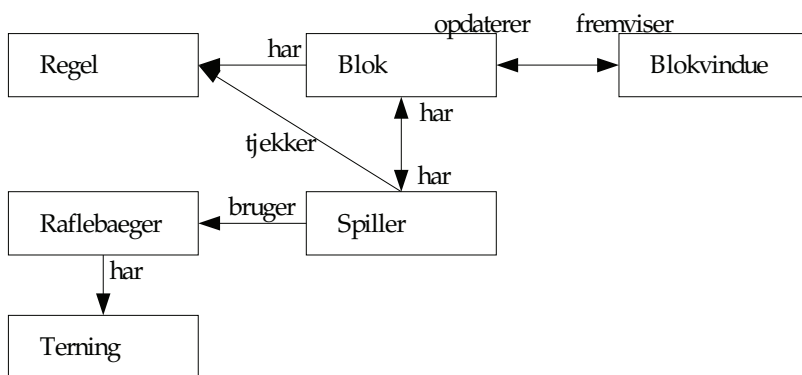
Det er vigtigt at huske, at dette kun er tommelfingerregler, som man ikke kan tage alt for bogstaveligt. Man bliver ofte nødt til at dreje tankegangen lidt for at få den til at passe i sit eget program.

F.eks. er en blyant eller et andet skriveredskab uundværlig i et virkeligt, fysisk Yatzy-spil (ellers kan man ikke skrive på blokken), men ingen erfarne programmører kunne drømme om at lave en Blyant-klasse og oprette Blyant-objekter, da blyanter slet ikke er vigtige for logikken i spillet.

## 22.3.2 Kollaborationsdiagrammer

Nyttige diagramformer under design er kollaborationsdiagrammer (samarbejdsdiagrammer), hvor man beskriver relationerne mellem klasserne eller objekterne på et overordnet plan.

Her er et eksempel:

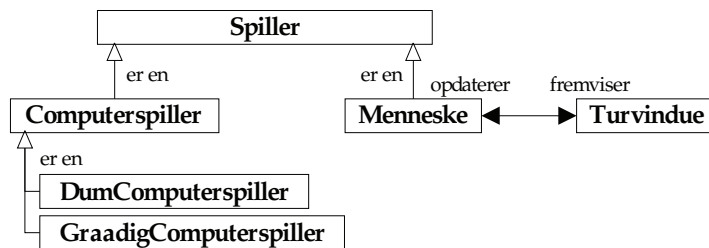


Har-relationer giver et vink om, at et objekt har en reference til (evt. ejer) et andet objekt:

- Raflebægeret har en reference til terningerne, ellers kan det ikke kaste dem. Terningerne kender ikke til raflebægerets eksistens.
- Blokken har nogle regler (en for hver række). Reglerne kender ikke til blokkens eller spillerens eksistens.
- Blokken har nogle spillere (en for hver søjle). Spillerne ved, de hører til en blok, hvor deres resultater skal skrives ind på.
- Blokkens data skal vises i et vindue. Der er brug for, at blokken kender til Blokvindue, vinduet, der viser blokken på skærmen, så det kan gentegnes, når blokken ændrer sig. Men vinduet har også brug for at kende til blokken, som indeholder de data, det skal vise.

Når spilleren tjekker regler, sker det gennem blok-objektet. Man kan forestille sig, at spilleren løber gennem alle blokkens regler og ser, om der er nogle, der passer, som han ikke har brugt endnu. Tjek af regler er altså ikke en har-relation, for spilleren har ikke en variabel, der refererer til reglerne.

Visse steder er der mange slags objekter, der kan indgå i samme rolle. Det gælder for eksempel Spiller i diagrammet ovenfor. Så kan man tegne et separat diagram, der viser rollerne.



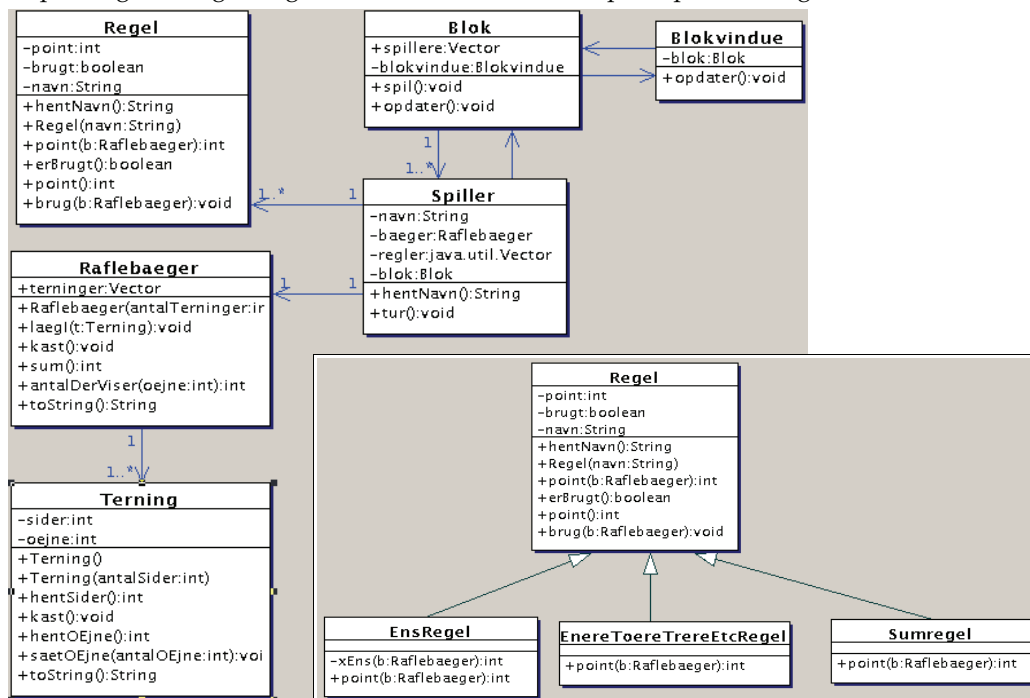
Er-en-relationer angiver generalisering eller specialisering (hvor nedrivning kan være en fordel). Det tegnes oftest med en hul pil.

Her er det lidt specielle, at én type spiller (nemlig Menneske) har et vindue tilknyttet. Dette vindue skal jo have adgang til at vise terningerne, så man skal huske at sørge for, at spillere har en reference til rafflebægeret.

## 22.3.3 Klassediagrammer

Herefter kan skitseres klassediagrammer, hvor man fastlægger nedrivning (er-en-relationerne), de vigtigste variabler og referencerne mellem objekterne (har-relationer) og de vigtigste metoder.

Dette kan eventuelt tegnes med et UML-udviklingsværktøj som samtidig kan generere kode til programmeringsfasen, f.eks. Oracle JDeveloper, ArgoUML, der kan hentes på <http://argouml.tigris.org/>, eller Poseidon for UML, på <http://www.gentleware.com/>:



Til højre ses, hvilke typer regel, der kunne forekomme.

# Engelsk-dansk ordliste

access	adgang, tilgang
applet	applet miniprogram der kan køre i en netlæser, f.eks. Firefox
array	tabel, række, array
assignment	tildeling a=5; er en tildeling af værdien 5 til variablen a
backslash	bagstreg ( \ - angives som '\\' i kildeteksten)
backspace	bak-tegnet ('\b')
block (statement)	blok en gruppering af kommandoer. Starter med { og slutter med }
boolean	logisk (boolsk) værdi sand (true) eller falsk (false).
browser	netlæser (f.eks. Netscape, Opera eller Internet Explorer)
bytecode	mellemkode binær kode i en .class-fil, genereret af Java-oversætteren fra en .java-kildetekst.
carriage return	vognretur ('\r')
cast	typekonvertering f.eks a = (int) f; konverterer værdien af f til et heltal før det tildeles a
character	tegn f.eks 'b', '9', '?'
character set	tegnsæt
checked exception	undtagelse med tvungen håndtering
class	klasse beskrivelsen af en objekttype
comment	kommentar tekst der forklarer programmet. Ignoreres af maskinen. En linje der starter med // bliver opfattet som en kommentar
compiler	oversætter f.eks fra Java-kildetekst til bytekode
concatenation	sammensætning (af tegnstreng) f.eks "Hej " + "verden" sammensætter to strenge
condition	betingelse a>5 er betingelsen i sætningen: if (a>5) System.out.println("a er 6 eller derover");
constructor	konstruktør
debugging	fejlfinding/aflusning af et program f.eks med trinvis gennemgang

declaration	erklæring int a; er en erklæring af, at a er et heltal.
decrement	nedtælling
deprecated	frarådet, forældet
digit	ciffer
encapsulation	indkapsling
enumeration	opremsning (af nogle elementer)
event	hændelse
event driven	hændelsesstyret
exception	undtagelse
expression	udtryk f.eks (a*5)/7
file	fil
floating-point number	kommatal i Java er kommatalstyperne float og double.
graphical user interface	grafisk brugergrænseflade
hardware	maskinel
host	værtsmaskine
identifier	navn f.eks a, public, class, int
increment	optælling
indentation	indrykning
inheritance	nedarvning
instance	instans, forekomst en forekomst af en klasse er et objekt, f.eks er "Hej" en forekomst af String
integer	heltal i Java er heltalstyperne int, long, short og byte.
interface	grænseflade
interpreter	fortolker
label	etikette
library	bibliotek
listener	lytter (efter hændelser)
location (in memory)	plads (i lager)
loop	løkke
memory	lager
method	metode f.eks har System.out metoden println()
nested	indlejret
newline	linjeskift ('\n')



open source	åben kildekode/kildetekst at kildeteksten er frit tilgængelig for forbedringer
overloading	overlæsning, navnesammenfald af metoder
override	tilsidesætte/omdefinere/overstyre/overskrive metode i en nedarvet klasse
package	pakke
parse	analysere, fortolke
pointer	reference/peger/hægte til et objekt et sted i lageret
polymorphism	polymorfi
reference	reference/peger/hægte til et objekt et sted i lageret
rounding	afrunding
scope	virkefelt
service	tjeneste
software	programmel
source code	kildetekst, kildekode
statement	sætning, ordre, kommando f.eks System.out.println("Hej"); eller a = 5;
stack trace	stakspor
stream	strøm
string	tekststreng
subclass	underklasse, subklasse, afledt klasse, nedarvet klasse
test	afprøvning
thread	tråd
token	brik, bid
truncation	nedrunding (3.7 bliver rundet ned til 3.0)
typecast	typekonvertering f.eks a = (int) f; konverterer værdien af f til et heltal før det tildeles a
underscore	understreg ('_')
unchecked exception	undtagelse uden tvungen håndtering
variable	variabel
visibility	synlighed
whitespace	blanktegn mellemrum, tabulatortegn og linjeskift er blanktegn



# Stikordsregister

## A

abstract 292  
abstrakte klasser 250, 292  
ActionListener 226, 228  
actionPerformed() 196, 226, 228  
adaptere 230  
addElement() 81, 93  
adgang til metoder/variabler 154  
adgangskontrol 154  
AffineTransform 183  
afkrydsningsfelter 198  
afledt klasse 125  
after() 92  
aktivitetsdiagrammer 307  
aktør 305  
alternativudtryk 63  
analyse 305  
AND 57  
anførelsestegn 91  
animationer 181  
anonyme klasser 230, 298  
antialias 182  
apostrof 91  
append() 94  
argumenter 71  
ArithmeticException 233  
aritmetiske operatorer 56  
array 166  
array versus ArrayList 169, 172  
ArrayIndexOutOfBoundsException 166, 232  
ArrayList 80, 93, 140, 169  
ArrayList 172  
arv 124, 220  
associative afbildninger 97  
AudioClip 189  
auto-commit 288  
autoboxing 93

## B

baggrundsfarve 178, 198  
bagstreg 91  
bak 91  
bankkonto 122  
BasicStroke 183  
basisklasse 125  
before() 92  
beregning af formel 162  
beregningsudtryk 34

betinget udførelse 39  
binær læsning og skrivning 248  
binære talsystem 246  
binært til tegnbaseret 251  
blok 42, 117  
BlueJ 27, 119  
Boks 100, 103, 105, 111, 141, 156, 242  
boolean 55  
boolesk variabel 38  
booleske udtryk 40  
BorderLayout 204  
Borland JBuilder 23  
break 64  
browser 188  
brugergænseflade 194  
brugssituationer 305  
buffer 248  
BufferedReader 245, 251  
byte 55  
ByteArrayInputStream 251  
bytekode 22, 32

## C

CallableStatement 289  
catch 233, 240  
char 55  
Character.isDigit() 158  
Character.isLetter() 158  
Character.isLowerCase() 158  
CharArrayReader 251  
Checked-filtreringsklasser 251  
checksum 251  
ClassCastException 131, 233  
ClassNotFoundException 235  
CLASSPATH 151  
clone() 278  
close() 244, 253  
CODEBASE 189, 192  
commit 288  
Comparator 216  
ComponentListener 228  
Connection 284  
containere 202  
continue 64  
cosinus 37  
createStatement() 284  
Cursor 178

## D

database 284  
databasedriver 284, 289  
DatabaseMetaData 290  
datakompression 252  
datastrømme 244, 249  
Date 85, 92  
design af klasser 308  
design-fane 194  
destroy() 190  
dialog-boks 202  
division 35  
do-while-løkken 63  
double 35, 55  
Double.parseDouble() 157, 246  
drawImage 176  
drawImage() 175  
drawLine() 175  
drawString 174  
drawString() 175  
drejninger 183  
drevbogstaver 254  
DriverManager 284

## E

Eclipse 24  
egenskaber 198  
eksekverbare jar-filer 153  
eksplicit typekonvertering 48, 56, 140  
eksponentiel notation 246  
elementtype 83, 93  
equals() 76, 95, 139  
er-en-relation 124, 133, 145, 310  
Error 241  
esperanto-dansk-ordbog 98  
etiket 198  
Event 222  
eventyrfortælling 82  
Exception 240  
executeQuery() 285  
extends 124

## F

FalskTerning 124, 126  
fange undtagelser 235  
farve 175  
fejl 50  
fejlmeddelelse 51  
Felt (matadorspil) 132, 133, 293  
File 254  
FilenameFilter 298  
FileNotFoundException 235  
FileReader 245, 250  
FileSystem.listRoots() 254  
FileWriter 244  
filhåndtering 254

filtrering af filnavne 298  
filtreringsklasser 251  
final 148, 293  
finally 242  
fjerninterface 280  
flerdimensionale arrays 171  
flertrådet programmering 266  
float 55  
FlowLayout 201, 203  
flueben 198  
flush() 256  
FocusListener 228  
Font 176  
for-løkken 45, 82  
forberedt SQL 289  
forbinde til en port 256  
foreach 82, 167, 169  
forgrundsfarve 178, 198  
formateringsstreng 59  
formelberegning 162  
formen af en klasse 117  
forpligtende SQL 288  
forældreklasse 125  
fraktaler 164  
fremviser 188  
FTP 260, 261

## G

Gade (matadorspil) 135, 144  
GeneralPath 183  
getConnection() 284  
getTime() 92  
grafisk brugergrænseflade 194, 222  
grafiske komponenter 194  
Graphics 175  
Graphics2D 182  
GridBagConstraints 206  
GridLayout 205  
Grund (matadorspil) 143  
grupper af klasser 150  
GUI 194  
GZIP-filtreringsklasser 252

## H

har-relation 108, 114, 122, 134, 145, 309  
HashMap 97  
Helle (matadorspil) 133  
heltal 33, 55  
heltalsdivision 56  
hexadecimale talsystem 246  
hjem 254  
hjemmesider 188  
HTML-dokument 188  
HTTP-tjeneste 256  
hændelser 222

## I

- if 40
- if-else 41
- Image 176
- implementere interface 212, 219
- implicit typekonvertering 49, 55
- import 150
- import af klasser 72
- indexOf() 76
- indkapsling 103, 285, 295
- indlejrede løkker 46
- indlæsning fra tastatur 236
- Indlæsning fra tastaturet 40, 236, 246
- indre klasser 295
- indrykning 42
- indtastningsfelt 199
- init() 190
- initComponents() 195
- initialisering uden for konstruktør 147
- InputStream 249
- InputStreamReader 251
- insert() 94
- insertElementAt() 81, 93
- Insets 206
- instruktioner 31
- int 33, 55
- Integer.parseInt() 157, 246
- interaktive programmer 196
- interface 212
- InterruptedException 267
- IOException 235
- ItemListener 228

## J

- J2SE 27
- JApplet 189
- jar-filer 152
- Java Web Start 192
- Java-arkiver 152
- java.awt 72, 150, 174, 177
- java.io 150, 249
- java.lang 150
- java.net 150, 257
- java.rmi 150, 280
- java.sql 150, 285
- java.text 150
- java.util 81, 150
- Java2D 182
- javac 32
- javadokumentationen 87
- javax.swing 150, 174, 176, 177, 183
- jbInit() 195
- JBuilder 23
- JButton 198, 227
- JCheckBox 198
- JComboBox 199

- JDBC 284
- JDeveloper 25
- JDialog 202
- JDK 27
- JFrame 178, 202
- JLabel 198
- JList 200
- JMenuBar 207
- JPanel 202
- JRadioButton 198
- JTable 209
- JTextArea 199
- JTextField 199
- JTree 209
- JWindow 202

## K

- kaste undtagelser 241
- KeyListener 227, 228
- kildetekst 31
- klassediagram 114, 132, 145, 310
- klassemetode 60
- klasser 68, 87, 116
- klassevariabler og -metoder 156, 160
- klipning 175
- kollaborationsdiagrammer 309
- kommandolinie-parametre 168
- kommatal 35, 55
- kommentarer 30, 53
- komponenter 194
- komprimering 252
- konstruktør 105
- konstruktør i underklasse 141, 143
- konstruktører 72, 116, 148
- Konto 122
- konvertere mellem arrays og lister 172
- koordinater 174
- kopiere en fil 248
- Kurvetegning 177
- kvadratrod 37
- køretidsfejl 51

## L

- layout 194
- layout-manager 203
- length() 76
- LineNumber-filtreringsklasser 251
- lineær transformation 183
- linieskift 91
- Linjelytter 230
- Linjetegning 223, 225
- LinjetegningAnonym 299
- LinjetegningIndre 296
- liste af objekter 80
- logisk variabel 38
- logiske fejl 50
- logiske udtryk 40, 57, 62

- lokale klasser 297
- lokale variable 101, 156, 158
- long 55
- lukke vindue 229
- LukProgram 229
- lytte på en port 258
- lytter 222
- læsLinje() 236
- læsTal() 236
- løkker 43

## M

- main()-metoden 31, 101, 158, 168
- manifest-fil 153
- matadorspil 132
- matematiske funktioner 37
- mellemvariabel 75
- memappe 254
- metadata 257, 290
- metodehovede 101, 118
- metodekald 38, 70, 71
- metodekrop 102
- metoder 38, 68, 73, 116, 160
- metoder i vinduer 178
- metoders returtype 74
- Microsoft 22
- modal 202
- MouseAdapter 230
- mouseClicked() 229
- mouseDragged() 225, 229
- MouseListener 222, 229
- MouseMotionListener 225, 229
- mouseMoved() 225, 229
- mousePressed() 229
- mouseReleased() 222, 229
- multipel arv 220
- multiplikation 35
- Muselytter 222
- musens udseende 178

## N

- navngivne løkker 64
- navngivningsregler 55
- nedarving 220
- nedarvning 124
- nedtælling 63
- NetBeans 26
- netlæser 188
- new-operatoren 69
- next() 285
- NOT 57
- notify() 272
- NotSerializableException 275
- NSidetTerning 112
- null 72, 135, 233
- null-layout 203
- NullPointerException 233

- nøgle 97
- nøgleindekserede tabeller 97
- nøgleordet static 156
- nøgleordet super 126, 141
- nøgleordet this 111

## O

- Object (stamklasse) 139
- ObjectOutputStream 251, 274
- objekter 68, 87
- objektorienteret analyse 305
- objektorienteret design 308
- objektreferencer 159
- objektrelationer 108, 122
- objektvariable 70, 100, 117, 156, 160
- ODBC 284, 289
- OOAD 304
- open source 28
- operatorer 56
- oprette objekt 69
- optimering 95, 248, 288
- optælling 63
- OR 57
- Oracle 25, 284
- OutputStream 249
- OutputStreamWriter 251
- override 125
- oversætte 32
- oversætterfejl 50

## P

- paintComponent() 174
- pakker 72, 150
- parametervariabler 158
- parametre 71, 87, 118, 158
- pause 136
- Person 114
- Piped-filtreringsklasser 251
- platformuafhængig 21, 32
- platformuafhængige filnavne 254
- Point 69, 88
- Polygon 176
- polymorfe variable 128
- polymorfi 129, 132, 138, 169, 216
- prepareCall() 289
- PreparedStatement 289
- prepareStatement() 289
- primitiv 60
- printf() 59
- printStackTrace() 234
- PrintWriter 244
- prioritet 272
- private 103, 154
- protected 148, 154
- public 103, 154
- Pushback-filtreringsklasser 251

## R

radioknapper 198  
RandomAccessFile 254  
readLine() 257  
readObject() 274, 278  
Rectangle 72, 89  
Rederi (matadorspil) 134, 144  
reference-typekonvertering 131  
referencer 159  
regneark 209  
regneudtryk 34  
rekursion 161, 254  
relationer mellem objekter 108, 122  
Remote 280  
removeElementAt() 81  
repaint() 178, 181  
replace () 76  
restdivision 56  
ResultSet 285, 290  
ResultSetMetaData 290  
return 102, 116  
returtype 74, 87, 101, 118  
reverse() 94  
RMI 280  
rmiregistry 281  
rollback() 288  
rotation 183  
run() 266  
Runnable 266  
række af data 166

## S

sammenligning af strenge 79  
sammenligningsoperatorer 40, 57  
sammensætte strenge 34  
Scanner 40, 246, 247  
semaforer 272  
serialisering 274, 280  
Serializable 275  
ServerSocket 258  
Shape 183  
short 55  
sikkerhed 192  
simple typer 55  
sinus 37  
skaleringer 183  
skilletegn 158  
skjule variabler/metoder 103  
skrifttype 178, 198  
skrive til en tekstfil 244  
sleep() 267  
SnakkesagligPerson 266  
Socket 256  
SocketException 235  
sortering 216  
specialtegn i strenge 91

Spiller (matadorspil) 133  
sprogfejl 50  
SQL 284  
SQLException 235, 284  
stakspor 234  
stamklasse 125  
stamklassen Object 139  
standardbiblioteket 87, 150, 157  
standardkonstruktør 106, 116, 142  
standardp 150  
Start (matadorspil) 134  
start() 190, 266  
Statement 284  
static 156  
statistik 167, 247  
statusfelt 189  
sti-separatortegn 254  
stifinder 209  
stop() 190  
stoppe programudførelsen 158  
strenge 76  
String 76, 90  
StringBuilder 94  
StringReader 250  
subklasse 125  
subrutiner 158  
substring() 76  
super 126, 141, 144  
superklasse 125  
switch 65  
synchronized 271  
synlighed af metoder/variabler 154  
syntaksfejl 50  
System.exit() 158  
System.out 157, 246

## T

tabulator 91  
talsystem 246  
Tastatur 40, 236  
tastetryk 227  
Tegnbar 212  
tegne grafik 175  
tekstdata 250  
tekststrenge 76  
Terning 107  
TextListener 229  
this 111, 122, 148  
Thread 266  
Thread.sleep() 267  
throw 241  
Throwable 241  
throws 235  
tildeling 33  
tilsidesætte 125  
tilsidesætte variabler 148  
titel på vindue 178

- tjeneste 256
- toLowerCase() 76
- toString() 139
- toUpperCase() 76
- transaktion 288
- transformation 183
- transient 275
- trappeudjævning 182
- trim() 90
- trinvis gennemgang 47
- try-catch-blok 233, 239
- tråde 266, 300
- typekonvertering 48, 55, 131, 140
- tællevariabel 44

## U

- udbygge en klasse 124
- udjævnede farveovergange 182
- uendelige løkker 47
- uforanderlig 77
- UML-notationen 69
- UML-udviklingsværktøj 310
- underklasse 125
- undtagelser 51, 232
- Unified Modelling Language 69
- Unified Process 305
- Unified Process" 304
- unicode 91
- UnknownHostException 235
- unreported exception 235
- URL 189

- URL-klassen 259
- user.dir 254
- user.home 254

## V

- valglister 199
- vandfaldsmodellen 304
- variabel-overskygning 148
- variabler 33
- variabler i interface 220
- vente 136
- virkefelt 158
- virtuel maskine 22
- vognretur 91
- void 87, 116
- vridninger 183
- værditypekonvertering 48, 55
- værtsmaskine 256

## W

- wait() 272
- webserver 258, 268
- while-løkken 43
- WindowListener 229
- writeObject() 274, 278

## Y

- Yatzyspil 305
- ydre klasse 295

## Z

- Zip-filtreringsklasser 251