Linux - Friheden til at programmere i Java

Version 0.7.20040516 - 2020-12-31

Christian Damsgaard

Jakob Nordfalk

Jonas Kongslund

og mange andre

Linux - Friheden til at programmere i JavaVersion 0.7.20040516 - 2020-12-31
af Christian Damsgaard, Jakob Nordfalk, Jonas Kongslund og og mange andre

Ophavsret © 2001-2005 Forfatterne som har ophavsret til bogen, men udgiver den under "Åben dokumentlicens (ÅDL) - version 1.0".

Bogen er nu nogenlunde klar - der mangler dog en del korrekturlæsning mht. struktur.

Indholdsfortegnelse

Forord ... xiii

1. Forord ... xiii
2. Linux-bøgerne ... xiii
3. Ophavsret ... xiv
4. Om forfatterne og bogens historie ... xiv
5. Vi siger tak for hjælpen ... xv
6. Typografi ... xvi

1. Indledning ... 1

1.1. Målgrupper ... 1
1.2. Om bogens kilde - http://javabog.dk ... 1
1.3. Javas historie ... 1
1.4. Hvad er en virtuel maskine ... 4
1.5. Programmering ... 5
1.5.1. Styresystemet ... 6
1.5.2. Hvorfor lære at programmere? ... 6
1.5.3. Et simpelt program ... 6
1.5.4. Hvordan lærer man at programmere ... 7
1.6. Fordele og ulemper ved Java ... 7
1.6.1. Enkelt ... 8
1.6.2. Objektorienteret ... 8
1.6.3. Platformuafhængigt ... 8
1.6.4. Netværksorienteret ... 8
1.6.5. Fortolket ... 8
1.6.6. Højtydende ... 8
1.6.7. Flertrådet ... 9
1.6.8. Robust ... 9
1.6.9. Sikkert ... 9
1.6.10. Dynamisk ... 9
1.6.11. Stor opbakning ... 9
1.6.12. Svagheder ved Java ... 10

2. Udviklingsmiljø ... 11

2.1. Indledning ... 11
2.2. Integrerede udviklingsmiljøer ... 11
2.2.1. Sun JDK ... 11
2.2.2. Borland JBuilder ... 11
2.2.3. Oracle JDeveloper ... 13
2.2.4. Sun Forte for Java ... 14
2.2.5. IBM VisualAge for Java ... 15
2.2.6. Andre ... 16
2.3. Opsætningsstyring ... 16
2.3.1. Ant ... 17
2.4. Debugging ... 19
2.5. Logning ... 19
2.6. Profilering ... 19
2.7. Dokumentation ... 20

2.7.1. Javadoc 20
2.8. Versionsstyring 25
2.9. Test 25
2.9.1. JUnit 25
3. Basal programmering 30
3.1. Det første javaprogram 30
3.1.1. Kommentarer 30
3.1.2. Klassedefinitionen 31
3.1.3. Oversættelse og kørsel af programmet 32
3.2. Variable 34
3.2.1. Heltal 34
3.2.2. Sammensætte strenge med + ... 36
3.2.3. Beregningsudtryk 37
3.2.4. Kommatal 38
3.2.5. Matematiske funktioner 40
3.2.6. Kald af metoder 42
3.2.7. Logiske variable 43
3.2.8. Opgaver 44
3.3. Betinget udførelse 44
3.3.1. if-else 46
3.3.2. Opgaver 47
3.4. Blokke 48
3.4.1. Indrykning 48
3.5. Løkker 49
3.5.1. while-løkken 49
3.5.2. for-løkken 53
3.5.3. Indlejrede løkker 54
3.5.4. Uendelige løkker 56
3.5.5. Opgaver 57
3.6. Værditypekonvertering 57
3.6.1. Eksplicit værdi-typekonvertering 58
3.6.2. Implicit værdi-typekonvertering 58
3.6.3. Misforståelser af værdi-typekonvertering 58
3.7. Fejl 59
3.7.1. Indholdsmæssige (logiske) fejl 59
3.7.2. Sproglige fejl 60
3.7.3. Køretidsfejl 61
3.8. Opgaver 62
3.8.1. Befordringsfradrag 62
3.8.2. Kurveprogram 62
3.9. Appendiks 62
3.9.1. Navngivningsregler 63
3.9.2. De simple typer 63
3.9.3. Værditypekonvertering 64
3.9.4. Aritmetiske operatorer 65
3.9.5. Regning med logiske udtryk 65
3.9.6. Sammenligningsoperatorer 66

3.9.7. Gode råd om programmering 67
4. Objekter 68
4.1. Objekter og klasser 68
4.2. Punkter (klassen Point) 70
4.2.1. Erklæring og oprettelse 70
4.2.2. Objektvariable 71
4.2.3. Metodekald 72
4.2.4. Eksempel 74
4.2.5. Import af standardklasser 75
4.3. Rektangler (klassen Rectangle) 75
4.3.1. Konstruktører 76
4.3.2. Metoder 77
4.3.3. Metoders returtype 78
4.3.4. Metoders parametre 80
4.4. Tekststrenge (klassen String) 80
4.4.1. Strenge er uforanderlige 83
4.4.2. Man behøver ikke bruge new til String-objekter 84
4.4.3. Navnesammenfald for metoder 84
4.4.4. At sætte strenge sammen med + ... 85
4.4.5. Sammenligning 86
4.4.6. Opgaver 87
4.5. Lister (klassen Vector) 87
4.5.1. Eksempel med Point 90
4.6. Ekstra eksempler 92
4.6.1. Blanding af kort med Vector 92
4.6.2. Datoer (klassen Date) 93
4.6.3. Opgaver 95
4.7. Appendiks 96
4.7.1. Point 96
4.7.2. Rectangle 97
4.7.3. String 98
4.7.4. Specialtegn i strenge 99
5. Definition af klasser 103
5.1. En Boks-klasse 103
5.1.1. Variabler 104
5.1.2. Brug af klassen 104
5.1.3. Metodedefinition 105
5.1.4. Flere objekter 106
5.2. Indkapsling 107
5.3. Konstruktører 109
5.3.1. Standardkonstruktører 111
5.3.2. Opgaver 112
5.4. En Terning-klasse 112
5.4.1. Opgaver 114
5.5. Relationer mellem objekter 114
5.5.1. En Raflebæger-klasse 114
5.5.2. Opgaver 117

5.6. Nøgleordet this 117
5.7. Ekstra eksempler 119
5.7.1. En n-sidet terning 119
5.7.2. Personer 122
5.7.3. Bankkonti 124
5.8. Opgaver 125
5.8.1. Fejlfinding 126
6. Nedarvning 129
6.1. At udbygge eksisterende klasser 129
6.1.1. Eksempel: En falsk terning 129
6.1.2. At udbygge med flere metoder og variabler 131
6.1.3. Nøgleordet super 133
6.2. Polymorfe variabler 133
6.2.1. Dispensation fra traditionel typesikkerhed 134
6.2.2. Polymorfi 135
6.2.3. Et eksempel på polymorfi: Brug af Raflebaeger 135
6.2.4. Hvilken vej er en variabel polymorf ? 137
6.2.5. Reference-typekonvertering 138
6.3. Eksempel: Et matador-spil 139
6.3.1. Polymorfi 148
6.4. Stamklassen Object 148
6.4.1. Referencer til objekter 149
6.5. Konstruktører i underklasser 150
6.6. Matadorspillet version 2 153
6.7. Opgaver 156
7. Pakker 158
7.1. At importere klassedefinitioner 158
7.2. Standardpakkerne 159
7.2.1. Pakken java.lang 159
7.3. Placering på filsystemet 160
7.4. At definere egne pakker 160
7.4.1. Eksempel på brugen af egne pakker 160
7.4.2. Navngive pakker 161
7.4.3. Pakke klasser i JAR-filer (Java-arkiver) 162
7.5. Opgaver 162
8. Lokale, objekt- og klassevariable 163
8.1. Eksempler i standardpakkerne 165
8.1.1. Klassevariable 165
8.1.2. Klassemetoder 165
8.2. Lokale variabler og parametre 166
8.2.1. Parametervariable 167
8.2.2. Rekursion 168

9. Arrays 170
9.1. Erklæring og brug 170
9.1.1. Eksempel: Statistik 171
9.2. Main-metoden 172
9.3. Arrays med startværdier 173
9.4. Gennemløb og manipulering 173
9.5. Array af objekter 174
9.5.1. Polymorfi 175
9.6. Arrays versus vektorer 175
9.7. Opgaver 176
10. Appletter og grafik 177
10.1. HTML-dokumentet 177
10.2. Javakoden 178
10.2.1. Metoder i appletter, som du kan kalde 179
10.2.2. Metoder i appletter, som systemet kalder 179
10.2.3. Eksempel 180
10.3. Opgaver 181
10.4. Appendiks 181
10.4.1. Metoder i appletter, som du kan kalde 181
10.4.2. Metoder, som systemet kalder 182
10.4.3. Klassen Graphics 183
11. Grafiske brugergrænseflader 185
11.1. Generering med et værktøj 185
11.1.1. Interaktive programmer 187
11.2. Komponenter 190
11.2.1. Button 191
11.2.2. Checkbox 191
11.2.3. Choice 192
11.2.4. TextField 192
11.2.5. TextArea 193
11.2.6. Label 193
11.2.7. List 194
11.2.8. Canvas 194
11.3. Eksempel 194
11.4. Containere 196
11.4.1. Panel 197
11.4.2. Applet 197
11.4.3. Window 197
11.4.4. Frame 197
11.4.5. Dialog 197
11.5. Relationer mellem klasserne 197
11.6. Layout-managere 198
11.6.1. FlowLayout 198
11.6.2. BorderLayout 199
11.6.3. GridBagLayout 199
11.7. Opgave: Matadorspillet grafisk 200
11.7.1. Vink 200

11.7.2. Flere vink 201
12. Interfaces - grænseflader til objekter 204
12.1. Definere et interface 204
12.2. Implementere et interface 205
12.2.1. Variabler af type Tegnbar 206
12.3. Eksempler med interfacet Tegnbar 206
12.3.1. En applet af Tegnbare objekter 208
12.4. Polymorfi 209
12.5. Interfaces i standardbibliotekerne 210
12.5.1. Sortering med Comparable 210
12.5.2. Flere tråde med Runnable 211
12.5.3. Lytte til musen med MouseListener 211
12.6. Opgaver 212
13. Hændelser i grafiske brugergrænseflader 213
13.1. Eksempel - LytTilMusen 213
13.2. Eksempel - Linjetegning 215
13.2.1. Linjetegning i én klasse 217
13.3. Ekstra eksempler 218
13.3.1. Lytte til musebevægelser 218
13.3.2. Lytte til en knap 219
13.4. Lyttere og deres metoder 220
13.4.1. ActionListener 220
13.4.2. ComponentListener 220
13.4.3. FocusListener 220
13.4.4. ItemListener 220
13.4.5. KeyListener 221
13.4.6. MouseListener 221
13.4.7. MouseMotionListener 221
13.4.8. TextListener 222
13.4.9. WindowListener 222
14. Undtagelser og køretidsfejl 223
14.1. Almindelige undtagelser 224
14.2. At fange og håndtere undtagelser 225
14.2.1. Undtagelsesobjekter og deres stakspor 226
14.3. Undtagelser med tvungen håndtering 227
14.3.1. Fange undtagelser eller sende dem videre 227
14.3.2. Konsekvenser af at sende undtagelser videre 228
14.4. Præcis håndtering af undtagelser 230
14.5. At fange flere slags undtagelser 233
14.6. Opgaver 234
15. Datastrømme og filhåndtering 235
15.1. Skrive til en tekstfil 235
15.2. Læse fra en tekstfil 237
15.3. Indlæsning fra tastatur 238
15.4. Analysering af tekstdata 238
15.4.1. Opdele inddata (StringTokenizer) 238

15.4.2. Konvertere til tal 240
15.4.3. DecimalFormat og DateFormat-klasserne 240
15.4.4. Samlet eksempel: Statistik 240
15.5. Appendiks 242
15.5.1. Navngivning 242
15.5.2. Binære data (-OutputStream og -InputStream) 242
15.5.3. Tekstdata (-Writer og -Reader) 243
15.5.4. Fillæsning og -skrivning (File-) 244
15.5.5. Strenge (String-) 244
15.5.6. Arrays (ByteArray- og CharArray-) 244
15.5.7. Læse/skrive objekter (Object-) 245
15.5.8. Dataopsamling (Buffered-) 245
15.5.9. Gå fra binære til tegnbaserede datastrømme 245
15.5.10. Filtreringsklasser til konvertering og databehandling 245
15.6. Ekstra eksempler 246
15.7. Opgaver 247
16. Netværkskommunikation 248
16.1. At forbinde sig til en port 248
16.2. At lytte på en port 251
16.3. Opgaver 253
17. Flertrådet programmering 254
17.1. Princip 254
17.1.1. Eksempel 255
17.2. Ekstra eksempler 257
17.2.1. En flertrådet webserver 257
17.2.2. En flertrådet applet med bolde 259
17.3. Opgaver 260
18. Serialisering af objekter 262
18.1. Hente og gemme objekter 262
18.2. Serialisering af egne klasser 264
18.2.1. Interfacet Serializable 264
18.2.2. Nøgleordet transient 264
18.2.3. Eksempel 264
18.3. Opgaver 266
19. RMI - objekter over netværk 267
19.1. Principper 267
19.2. Konkret 268
19.2.1. På serversiden 268
19.2.2. På klientsiden 269
20. JDBC - databaseadgang 272
20.1. Kontakt til databasen 272
20.1.1. JDBC-ODBC-broen under Windows 273
20.2. Kommunikere med databasen 273
20.2.1. Kommandoer 273
20.2.2. Forespørgsler 274
20.3. Samlet eksempel 274

20.4. Opgaver 276
21. Avancerede klasser 278
21.1. public, protected og private 278
21.1.1. Variabler og metoder 278
21.1.2. Klasser 279
21.2. Nøgleordet final 279
21.2.1. Variabler 279
21.2.2. Metoder 280
21.2.3. Klasser 281
21.3. Nøgleordet abstract 281
21.3.1. Klasser 281
21.3.2. Metoder 282
22. Indre klasser 283
22.1. Almindelige indre klasser 283
22.1.1. Eksempel - Linjetegning 284
22.2. Lokale klasser 285
22.3. Anonyme klasser 286
22.3.1. Eksempel - filtrering af filnavne 287
22.3.2. Eksempel - Linjetegning 288
22.3.3. Eksempel - tråde 289
22.4. Opgaver 290
23. Objektorienteret analyse og design 291
23.1. Krav til programmet 291
23.2. Objektorienteret analyse 292
23.2.1. Skrive vigtige ord op 292
23.2.2. Brugsmønstre 293
23.2.3. Aktivitetsdiagrammer 293
23.2.4. Skærmbilleder 294
23.3. Objektorienteret design 296
23.3.1. Kollaborationsdiagrammer 296
23.3.2. Klassediagrammer 298
24. Internationale programmer 300
24.1. Indledning 300
24.2. Lokalindstillinger 301
24.2.1. Oprettelse af en lokalindstilling 301
24.2.2. Tilgængelige lokalindstillinger 302
24.3. Ressourcebundter 303
24.3.1. Generelt om ressourcebundter 303
24.3.2. Lagring af tekst i ressourcefiler 304
24.3.3. Lagring af ressourcer i klasser 305
24.4. Parametriserede beskeder 306
24.5. Formatering af datoer og klokkeslæt 306
24.5.1. Prædefineret formater 306
24.5.2. Egne formater 307
24.6. Formatering af tal og beløb 309
24.6.1. Prædefineret formater 309

24.7. Tekster og tegn 311
24.7.1. Analyse af tegn 312
24.7.2. Sammenligning af strenge 312
24.7.3. Analyse af grænser i tekst..313
A. Revisionshistorie for bogen 314

Tabelliste

3-1. Simple variabeltyper i Java....................63
3-2. Java....................65
3-3. Java....................66
3-4. Java....................66
3-5. Java....................66
3-6. Java....................67
4-1. Java....................99
11-1. Java....................190
11-2. Java....................191
11-3. Java....................191
11-4. Java....................192
11-5. Java....................193
11-6. Java....................194
11-7. Java....................194
21-1. Adgangsregler....................278
24-1. Dato og klokkeslætmønstre....................308

Forord

1. Forord

Denne bog er blevet startet af Christian Damsgaard som syntes at den manglede i samlingen af "Frihedens" bøgerne. Meget hurtigt meldte Jonas Kongslund og Jacob Nordfalk sig på banen mht. at hjælpe med skrive arbejdet. Jakob Nordfalk har enddog doneret en del materiale fra sin egen Java bog "Objektorienteret programmering i Java" (http://javabog.dk).

Denne bog er oprindeligt skrevet og redigeret af Christian Damsgaard, Jonas Kongslund, Jakob Nordfalk, som har ophavsret på bogen.

Den største del at materialet i denne bog, består af det materiale som Jacob Nordfalk har doneret til bogen - det takker vi for. I fremtiden vil Jacobs og denne bog muligvis bevæge sig længere fra hinanden, og der vil komme nye afsnit til.

2. Linux-bøgerne

Bogen er en del af en serie, som kan findes på http://www.linuxbog.dk/

- *Linux – Friheden til at vælge installation* – Om at installere Linux.
- *Linux – Friheden til at lære Unix* – Om hvordan man bruger Linux' (og Unix') kommandolinjeværktøjer.
- *Linux – Friheden til at vælge grafisk brugergrænseflade* – Om alle de grafiske brugergrænseflader, der findes til Linux.
- *Linux – Friheden til at vælge programmer* – Om de programmer du kan få til Linux.
- *Linux – Friheden til systemadministration* – Om at administre sit eget linuxsystem.
- *Linux – Friheden til at programmere* – Programmering på Linux
- *Linux – Friheden til at programmere i C* – Om at programmere i sproget "C".
- *Linux – Friheden til at programmere i Java* – Om at programmere i sproget "Java".
- *Linux – Friheden til sikkerhed på internettet* – Om at sikre dit Linuxsystem mod indbrud fra internettet.
- *Linux – Friheden til egen webserver* – Om at sætte en webserver med databaser, CGI-programmer og andet godt op.
- *Linux – Friheden til at skrive dokumentation* – Om at skrive dokumentation (og andet) i SGML/DocBook, LaTeX eller andre formater.
- *Linux – Friheden til at vælge kontorprogrammer* – Kontorfunktioner på et Linux/KDE/OpenOffice.org-system.

- *Linux – Friheden til at vælge IT-løsning* – Om muligheder, fordele og ulemper ved at bruge Linux i sin IT-løsning.
- *Linux – Friheden til at vælge OpenOffice.org* – Om at bruge OpenOffice.org, både på Linux og på andre styresystemer.
- *Linux – Friheden til at vælge digital signatur* – Digital signatur på Linux.

3. Ophavsret

Denne bog er skrevet af Linux-brugere til Linux-brugere. Store dele af bogen er skrevet eller redigeret af enkelte forfattere, hvilket er nævnt i revisions-historien til bogen.

Bogen kan findes i opdateret form på http://www.linuxbog.dk/, mens prøve-udgaver kan findes på http://cvs.linuxbog.dk/.

Figur 1. ÅDL

Bogen er udgivet under "Åben dokumentlicens (ÅDL) – version 1.0" som kan læses på http://www.linuxbog.dk/licens.html. Du har bl.a. herved frit lov til at kopiere dette værk uændret på ethvert medium.

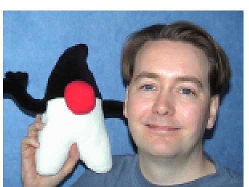
Kommentarer, ris og ros og specielt fejl og mangler bedes sendt til linuxbog@sslug.dk (mailto:linuxbog@sslug.dk), men er du medlem af SSLUG kan du i stedet for med fordel skrive til sslug-bog@sslug.dk (mailto:sslug-bog@sslug.dk).

4. Om forfatterne og bogens historie

Bogen er primært skrevet af nedenstående personer:

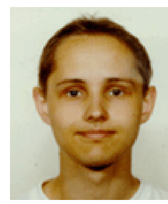
- Christian Damsgaard, BSc (hons) Software Engineering, De Montfort University Leicester

Figur 2. Christian Damsgaard



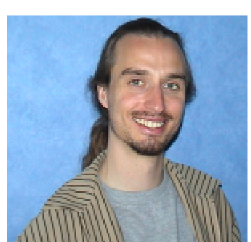
- Jonas Kongslund, Stud. MSc. Datalogi, Århus Universitet

Figur 3. Jonas Kongslund



- Jakob Nordfalk, cand.scient. i fysik, Københavns Universitet.

Figur 4. Jacob Nordfalk



5. Vi siger tak for hjælpen

Vi har haft stor glæde af mange SSLUG-medlemmers støtte, rettelser og forslag til forbedringer - bliv ved med dette. Specielt vil vi nævne:

- Peter Toft
- Erik Søe Sørensen
- Alfred Jensen
- Claus Madsen

Du kan i Appendiks A finde en liste over alle de revisioner, som bogen har været igennem.

Hvis du har ord du ikke forstår, så kan http://www.whatis.com være interessant. Her kan du slå mange computerord op dog kun på engelsk. I øvrigt kan bogens stikordsregister være interessant.

6. Typografi

Vi vil afslutte indledningen med at nævne den anvendte typografi.

- Navne på filer og kataloger skrevet som `foo.bar`
- Kommandoer, du udfører ved at taste, skrives som **help**
- Der er flere steder i bogen, hvor vi viser, hvad brugeren (som vi kalder "tyge") taster, og hvad Linux svarer. Det vil se ud som:

  ```
  [tyge@hven ~]$ Dette taster brugeren
  Dette svarer Linux.
  ```
- Der er tilsvarende flere steder i bogen hvor vi viser hvad systemadministratoren (root) taster, og hvad Linux svarer. Det vil se ud som:

  ```
  hven# Dette taster systemadministratoren
  Dette svarer Linux.
  ```
  Det vigtige her er at kommandofortolkeren bruger nummertegnet (#) til at markere at man har systemadministratorrettigheder.

Kapitel 1. Indledning

1.1. Målgrupper

Denne bog er tiltænkt to målgrupper:

- Folk, der aldrig har prøvet at programmere før. Disse kan starte med Kapitel 3, Basal programmering og læse derfra.
- Folk, der allerede kan programmere, og som har brug for et referenceværk, hvor man kan slå forskellige emner op, og få dem gennemgået skridt for skridt.

Som nybegynder er det vigtigt at du forstå begreber som filsystemet og kommandolinjen inden du går igang med denne bog. Du kan læse om disse to emner i de andre Friheden til... bøger.

1.2. Om bogens kilde - http://javabog.dk

Størstedelen af den bog du læser i lige nu stammer egentligt fra lærebogen "Objektorienteret programmering i Java" af Jacob Nordfalk. Den kan findes på adressen http://javabog.dk (http://javabog.dk/).

I den trykte bog af Jacob Nordfalk findes en del ting som ikke ligger gratis tilgængelig på nettet, bl.a.:

- "Test dig selv"-afsnit hvor man kan tjekke om man har forstået det væsentligste.
- Resuméer der på punktform giver overblik over kapitlet.
- Avancerede afsnit med ekstra stof til dem der vil vide mere.

Det kan anbefales at købe den trykte bog hvis du er begynder og har brug for en pædagogisk lærebog. På hjemmesiden http://javabog.dk (http://javabog.dk/) kan du også hente programeksempler, forslag til læseplan, undervisningsmateriale med ugesedler med øvelser, transparenter, løsningsforslag og meget andet.

1.3. Javas historie

Der var en gang - sådan starter mange eventyr. På sin egen måde har Java også været et eventyr, ikke kun for SUN Micro Systems (SUN) men også for den store skare af udviklere der i tidens løb har taget sproget til sig på godt og ondt.

Tilbage i 1992 var der nogle visionære personer ansat hos SUN, der ønskede at definere et programmeringssprog samt et afviklingsmiljø, som kunne anvendes til at afvikles i små diskrete enheder. Det primære årsag til at kikke på denne opgave var, at hver gang en virksomhed skulle at udvikle en ny enhed med en indbygget mikroprocessor, skulle de også til at udvikle oversættere, operativsystem, enhedsprogrammer mv. Ud over denne relative store opgave, skulle de enkelte udviklere også bruge en del tid på at sætte sig ind i den nye hardware platform, og evt. nyt assembler sprog (C bliver typisk kun anvendt til større enheder med mere hukommelse og hurtigere processor).

SUN's tanke var at definere et simpelt letvægts afviklingsmiljø (virtual machine) som enkelt kunne implementeres på forskellige typer af processorer. Derved kunne udviklerne koncentrere sig om at udvikle funktionalitet i stedet for at prøve at forstå hvordan den underliggende arktitektur var opbygget.

Udviklingen af det sprog, der skulle programmeres i, tog udgangspunkt i C og C++, specielt mht. syntaks. Dette ville gøre det let for målgruppen, som typisk allerede kendte til især C, at komme hurtigt igang med udvikling af software.

SUN's direktion besluttede at sætte en række midler af til et forskningsprojekt der skulle afdække mulighederne for en sådan platform. Projektet havde mange kodenavne, deriblandt "stealth project" og "project green". Efter et årstid havde gruppen fået defineret platformen og det var nu tid til at prøve tingene af i virkelighedens verden.

Test-projektet (kendt under navnet "Star Seven") var en simpel PDA (Personal Digital Assistent - som PalmPilot). Projektet var en succes, de havde bevist at de kunne udvikle det samme produkt på væsentligt kortere tid end det normalt ville havde taget dem (faktor 3 i forhold til normalt), men SUN besluttede at sig for at markedet endnu ikke kunne bære et produkt af denne type (SUN skævede selvfølgelig til Apples PDA - Newton - som Apple, på trods af at produktet var rimelig modent, ikke kunne sælge).

I stedet gik SUN i forhandlinger med bla. Times Warner om at implementere deres platform på såkaldte SetTop bokse, som er meget udbredte i bl.a. USA. Af ukendte årsager kunne Time Warner og SUN ikke blive enige, og aftalen blev aldrig til noget.

På det tidspunkt virkede det som om platformen ikke skulle blive til noget, projektet blev kørt lidt ud på et sidespor, men der blev arbejdet på livet løs i de små kontorer i Silicon Valley. På det tidspunkt hed programmeringssproget Oak - årsagen skulle efter sigende være, at James Gosling blev inspireret af et egetræ (eng. oak) der stod udenfor hans kontorvindue. Nu ville skæbnen blot at der allerede eksisterede et firma der havde navnebeskyttet (trademark) navnet Oak Technologies. De brugte noget tid på at brainstorme omkring navnet og kom på navne som: Silk, DNA, Aliva, Jolt, Ruby, WRL (WebRunnerLanguage), Lyric, Pepper, NetProse, Neon og Lingua Java (læs mere her: http://www.javaworld.com/jw-10-1996/jw-10-javaname.html (http://www.javaworld.com/jw-10-1996/jw-10-javaname.html)).

I starten af 1995 havde en af SUN's udviklere lavet en webbrowser i Java på en weekend (det siger nok mere om webbrowseren end om Java). Det specielle ved denne webbrowser (senere kendt som HotJava) var at den kunne afvikle Java indlejret i HTML-siden, hvilket var fuldstændigt nyt på dette tidspunkt.

HotJava blev vist frem på SUN World '95. Her blev det set af nogle folk fra Netscape, som på det tidspunkt var den altdominerende webbrowser på markedet, og de kunne godt lide tanken om at kunne afvikle programkode som en del af en HTML-side. Netscape købte en licens af SUN til at implementere en Java virtuel maskine, og den første version af Netscape med Java 1.1 var version 3. Det blev en bragende succes. I løbet af meget kort tid spredte kendskabet til Java sig og alle skulle lige pludselig til at programmere appletter (navnet på den slags Javaprogrammer der kan afvikles i en browser).

Siden 1995 er det gået hurtigt for Java. Version 1.1.0 blev hurtigt til version 1.1.2 osv. På nuværende tidspunkt er den seneste 1.x version fra SUN version 1.1.8_008. Version 1.1.x bliver ikke længere videreudviklet og har startet sit sidste liv - dvs. SUN retter ikke længere andet end meget kritiske fejl (dvs. ingen).

I starten af 1999 kom version 1.1.x's afløser - version 1.2.0. I denne version var der ændret på en del af arkitekturen i Java og rent markedsføringsmæssigt blev det kaldt "Java 2", hvilket den næste også hedder idag (Java 2 Standard Edition). Den seneste version 1.2.x fra SUN er version 1.2.2_010.

Den seneste endelige version af SUN hedder version 1.3.x (seneste 1.3.1_02) og i denne version er der også blevet plads til nogle nye features (dog ikke så mange som ved hoppet fra version 1.1.x til 1.2.x).

Den næste version af Java hedder version 1.4.0 og er i skrivende stund i beta 3 - dvs. næsten klar til udgivelse. Den nye version af Java er meget spændende mht. nye features.

SUN havde lige fra starten (1.1) valgt at lade oversætter og en version af den virtuelle maskine være tilgængelige gratis til nedhentning via internettet.

Det er stadig gratis at hente og bruge Java til både private og kommecielle formål. SUN leverer Java udviklingsmiljøer til følgende operativsystemer:

- Linux (forskellige distributioner)
- SUN Solaris
- Microsoft Windows 95,98,Me,NT,2000 & XP

Ud over disse versioner eksisterer der også et produkt der hedder JavaOS som bl.a. kan afvikles under DOS med DPMI-understøttelse. Denne version af Java bliver dog pt. ikke videreudviklet aktivt af SUN og kan ikke længere findes på deres hjemmeside.

Den seneste version af Java kan hentes fra SUN Java hjemmeside: http://java.sun.com/j2se (http://java.sun.com/j2se/).

Ud over SUN er der en række firmaer der også implementere Java til række platforme. Her er listen fra SUN's hjemmeside:

- AIX (IBM)

- DG/UX 4.2 (Data General Corporation)
- DYNIX/pt 4.2.2 (Intel)
- HP-UX (Hewlett-Packard)
- IRIX (Silicon Graphics)
- Linux (Blackdown)
- MacOS (Apple)
- NetWare (Novell)
- OpenVMS (Compaq Computer Corporation)
- OS/2 (IBM)
- OS 390 (IBM)
- OS 400 (IBM)
- SCO (SCO)
- True64 (Compaq Computer Corporation)
- UnixWare (SCO)
- VxWorks (Wind River Systems)
- Windows NT (Digital Equipment Corporation)

For mere information om de forskellige versioner kik på siden: http://java.sun.com/cgi-bin/java-ports.cgi (http://java.sun.com/cgi-bin/java-ports.cgi).

1.4. Hvad er en virtuel maskine

Java afvikles igennem en virtuel maskine. En virtuel maskine er, som navnet antyder, virtuel - dvs. ikke eksisterende. Det, SUN har gjort, er at definere et afviklingsmiljø, som passer godt med den måde, Java er skruet sammen. Afviklingsmiljøet skjuler den underliggende platform (både hvad angår hardware og operativ system).

Når man oversætter et Java program sker der to ting: Programmets syntaks tjekkes, og der generes såkaldt mellemkode (byte-code). Mellemkoden er en mellemting mellem kildetekst og maskinekode (også kaldt assemblerkode). Når man oversætter et program i et "normalt" programmeringssprog, genereres der normalt maskinekode. Denne maskinkode kan direkte afvikles af den processor, man har valgt at oversætte til. Typisk den samme processor man afvikler oversætteren under, men ikke nødvendigvis. Dette gør at kode kun kan afvikles på den valgte processor og ikke andre. Man kan for eksempel ikke få glæde nye features i senere generationer af processoren.

Det den virtuelle maskine gør, er, at den fortolker den mellemkode, som er genereret af oversætteren. Ofte er der ikke særlig langt imellem mellemkode og den pågældende platforms arkitektur, dvs. for hver mellemkode instruktion er der meget få, eller bare en enkelt maskinekode instruktion.

Spørgsmål - "Hvorfor er der ikke nogen der har lavet en processor der kan forstå Java mellemkode direkte?". Der er der faktisk også nogen der har, men processoren har ikke kunnet hamle op med mere og mere avancerede virtuelle maskiner som bl.a. SUN og IBM har lavet.

Selvom der ikke er særlig langt mellem mellemkode og maskinekode, skal den virtuelle maskine gøre mere end bare at oversætte mellem de to typer af kode. Den virtuelle maskine er også ansvarlig for styring af hukommelse samt for grænsefladen mod det underliggende operativsystem.

For at få det hele til at køre lidt hurtigere har mange leverandører af virtuelle maskiner udviklet en teknologi kaldet JIT - Just In Time. Tanken bag denne teknologi er, at lige inden kode skal afvikles, bliver det oversat til maskinekode på den aktuelle platform, og gemt til næste gang den samme kode skal oversættes. Disse oversatte dele bliver ikke gemt efter at den virtuelle maskine er afsluttet.

Den seneste generation af virtuelle maskiner fra SUN har kodenavnet HotSpot. Dette navn er meget velvalgt, da det, denne type af virtuelle maskine koncentrerer sig om, er de dele af koden som bliver afvikles oftes - og derfor også bør være dem, der bliver oversat først og bedst. Den virtuelle maskine foretager en såkaldt profilering af kode, hvor den gemmer information om hvor ofte en given metode bliver kaldt, hvor lang tid den tager at afvikle mv. Efter at den virtuelle maskine har været startet i en periode har den indsamlet information nok til at begynde at genere maskinekode. I og med den sidder med alle kort på hånden mht. det øjeblik hvor kode afvikles, kan den udvælge den / de funktioner, der bruges oftes, og som derfor bør oversættes først. Er der tale om en simpel kort metode, kan den virtuelle maskine vælge at indlejre kode fra funktion de steder, hvor den kaldes. Regulære funktionskald med overførsel af parametre erstattes således af kopier af funktionens kode. Dette sparer en del kostbare processor trin. Ifølge SUN er ydelsen fra en HotSpot virtuel maskinen meget tæt på den ydelse, man kan få fra det tilsvarende C++ program, fordi den optimering, den virtuelle maskine kan lave, er væsentlig bedre end den, C++ oversætteren kan foretage på oversættelsestidspunktet. Hvis man endvidere implementere nogle af de avancerede ting fra Java (fx. garbage collector, sikre referencer mv), som påvirker ydelsen negativt, udkonkurerer Java med HotSpot teknologien C++ fuldstændigt (jf. SUN).

Man kan få virtuelle Javamaskiner til næsten alle platforme, lige fra de store IBM mainframe systemer (OS 390), og ned til et smartcard (fx. Danmønt kortet). Det, som er den store forskel mellem implementeringen af Java på de forskellige platforme, er de runtime-biblioteker, der medfølger - selve mellemkoden er nøjagtig den samme.

På nuværende tidspunkt findes der:

- Java 2 Enterprise Edition (applikation server)
- Java 2 Standard Edition (arbejdsstation)
- Java 2 Micro Edition (indlejret systemer)

1.5. Programmering

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

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

1.5.1. Styresystemet

Styresystemet er det program, som styrer computeren, og tillader brugeren at bruge andre programmer. Af styresystemer kan nævnes Linux, DOS, Windows, MacOS, OS/2, 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 web-browser, og afvikles der. Den type programmer kaldes appletter eller miniprogrammer.

1.5.2. Hvorfor lære at programmere?

Det er sjovt og spændende, og 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, kan kendskab til programmering være en stor 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.

1.5.3. 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 kogebogsopskrift, som computeren følger punkt for punkt ovenfra og ned. Hvert punkt (eller instruktion eller kommando) gøres færdigt, før der fortsættes til næste punkt.

1.5.4. 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, at den eneste betydende faktor i produktiviteten for programmører er: Erfaring. Han fandt i øvrigt forskelle i produktiviteten på en faktor 25.

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.

1.6. 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).

Det har på 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.

1.6.1. 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 fatale fejl, der eksisterer i andre programmeringssprog, er fjernet i Java.

1.6.2. Objektorienteret

Samtidig kommer det med over 1000 foruddefinerede objekt-typer, som kan udføre næsten enhver tænkelig opgave. Præcist hvad "objektorienteret" betyder handler denne bog om.

1.6.3. 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.

1.6.4. 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.

1.6.5. Fortolket

Kildeteksten oversættes til en standardiseret platformuafhængig kode (kaldet bytekode), som derefter udføres af en javafortolker på det enkelte styresystem. Derved opnås, at man kun behøver at oversætte sin kildetekst én gang. Javafortolkeren er en såkaldt virtuel maskine, der konverterer instruktionerne i bytekoden til instruktioner, som det aktuelle styresystem kan forstå.

1.6.6. Højtydende

De nuværende fortolkere tillader javaprogrammer at blive udført næsten lige så hurtigt, som hvis de var blevet oversat direkte til det pågældende styresystem.

1.6.7. Flertrådet

Java er designet til at udføre flere forskellige programdele samtidigt, og en programudførsel kan blive fordelt over flere CPU'er (se kapitlet om flertrådet programmering).

1.6.8. Robust

Java er også robust; under afviklingen af et program tjekkes det, at handlingerne er tilladelige, og opstår der en fejl, såsom en ønsket fil ikke kan findes, erklærer 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 kapitlet om undtagelser).

1.6.9. 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.

1.6.10. 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 eksempelvis kapitlet om RMI).

1.6.11. Stor opbakning

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

Softwaregiganten Microsoft er en undtagelse. Microsoft blev i efteråret 2000 kendt skyldig ved domstolen i USA for ulovligt at misbruge sin monopollignende magt på PC-markedet for at skade bl.a. Java. Sagen blev naturligvis anket, og det er i skrivende stund uvist hvad den ender med.

Microsoft er ikke interesseret i, at programmerne kan udføres under andre styresystemer end Windows. De har lavet deres egen udgave af Java, der kun virker under Windows, og har (uden større held) forsøgt at lokke programmører til at bruge den.

1.6.12. Svagheder ved Java

Java har også en del kritikere, især blandt de, hvis forretningsmodel eller område er truet af Java. Ikke desto mindre er der nogle gode pointer iblandt:

- Java kræver 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.
- Sun Microsystems ejer Java, og nogle frygter, at det vil udvikle sig til et monopol som med Microsoft. Indtil videre har de dog opført sig eksemplarisk og bl.a. frigivet hele kildeteksten til Java, og derudover findes der andre uafhængige udgaver af Java.
- Java satser på at være platformuafhængigt, men der er alligevel små forskelle på de forskellige platforme. Dette gælder specielt ved programmering af grafiske brugergrænseflader. Det kræver erfaring og test at sikre sig, at ens program virker tilfredsstillende på flere platforme. Dette er ikke kun et Java-relateret problem, designere af hjemmesider har tilsvarende problemer. Java gør det nemmere at skrive platformuafhængige programmer, men det løser ikke alle problemer for programmøren.

Kapitel 2. Udviklingsmiljø

2.1. Indledning

Et professionelt udviklingsmiljø består af værktøjer, der helt eller delvist kan løse nedenstående opaver.

- Redigering af kildetekst
- Oversættelse af kildetekst
- Opsætningsstyring
- Debugging
- Logning
- Profilering
- Dokumentation
- Versionsstyring
- Test

De næste mange sektioner vil gå i dybden med flere af disse opgaver og give bud på værktøjer, der er nyttige i den sammenhæng. Allerførst vil vi dog se på en række populære integrerede udviklingsmiljøer.

2.2. Integrerede udviklingsmiljøer

Denne sektion giver et hurtigt overblik over populære integrerede udviklingsmiljøer.

2.2.1. 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 eller emacs under Linux.

Til oversættelse og kørsel kan man installere et Java-udviklingskit udgivet af Sun, f.eks. JDK1.3 (Java Development Kit version 1.3). Det kan hentes gratis på http://www.javasoft.com til et væld af styresystemer.

JDK'en 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.

2.2.2. Borland JBuilder

Der findes en række mere avancerede programmeringsværktøjer, hvor JBuilder fra Borland er en af de mest populære. JBuilder er skrevet i Java og kører på både Macintosh, Windows, Linux og Sun Solaris. Det anbefales at have 128 MB RAM.

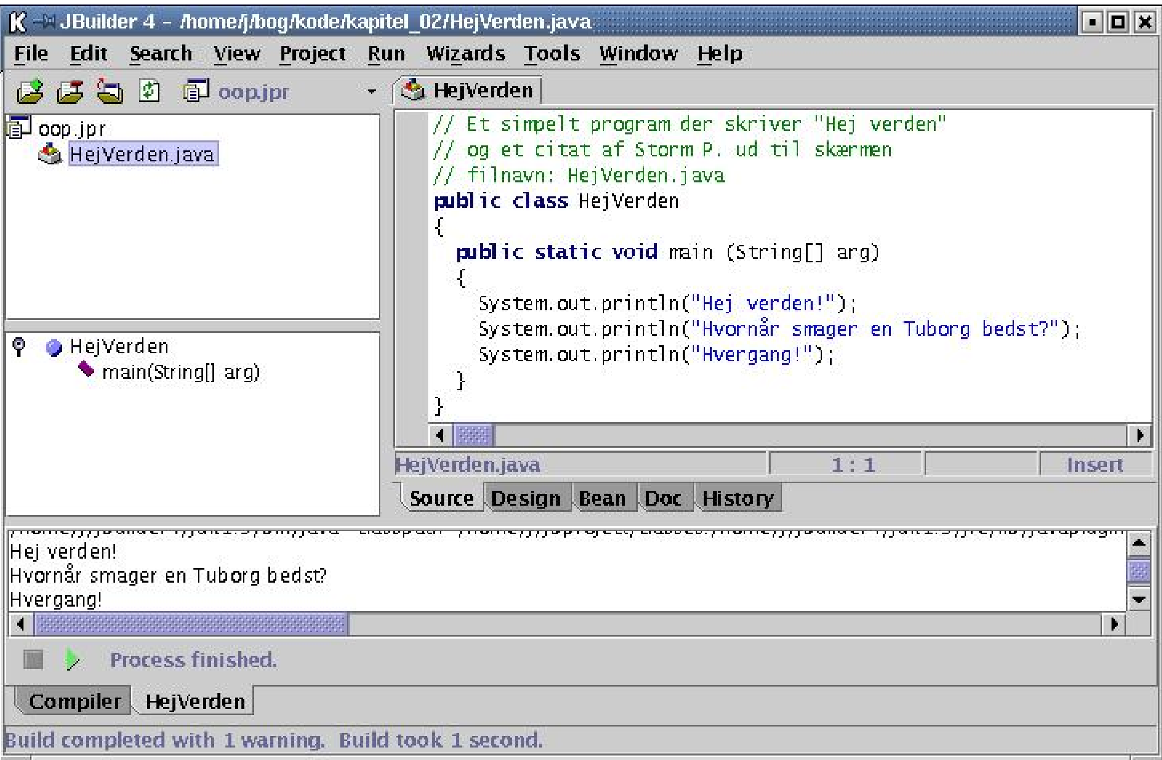
En basisversion af JBuilder kan hentes gratis fra http://www.borland.com/jbuilder/. Ønsker man adgang til de mere avancerede funktioner skal programmet købes.

JBuilder er opbygget 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 starter derefter fortolkeren. Uddata kan ses i den nederste ramme. "Debug" (der findes under "Run") bruges til fejlfinding af programmer og giver mulighed for at udføre programkoden trinvist.

Når man udvikler i JBuilder, er det baseret på projekter. Projekterne indeholder en liste over kildetekst-filerne samt, hvis det ønskes, projektinformation (lagret som HTML-dokument).

Projektet (her oop.jpr) kan ses i den venstre ramme. Den højre ramme indeholder kildeteksten på et faneblad. På de andre faneblade er bl.a. designværktøj til grafiske brugergrænseflader, dokumentationsvisning og versionskontrol.

Figur 2-1. Borland JBuilder

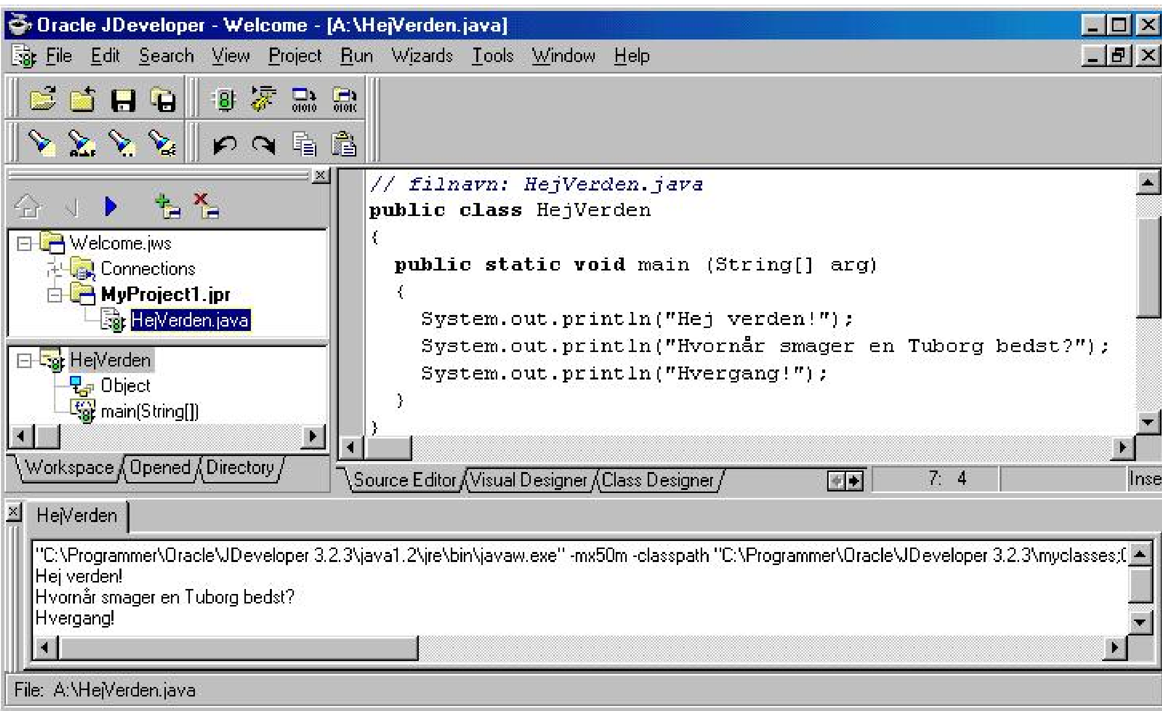


2.2.3. Oracle JDeveloper

Oracle udgiver JDeveloper, som er en udgave af JBuilder, hvor Oracle har integreret sit eget database-produkt. JDeveloper er lidt mere indviklet at bruge, idet projekter er samlet i arbejdsområder, en facilitet, man sjældent har brug for som almindelig udvikler.

JDeveloper kræver 128 MB RAM og kører under Windows og Linux. Det kan købes af Oracle, men bruger man i forvejen Oracles produkter, er der sandsynlighed for, at man har fået leveret JDeveloper med i pakken.

Figur 2-2. Oracle JDeveloper

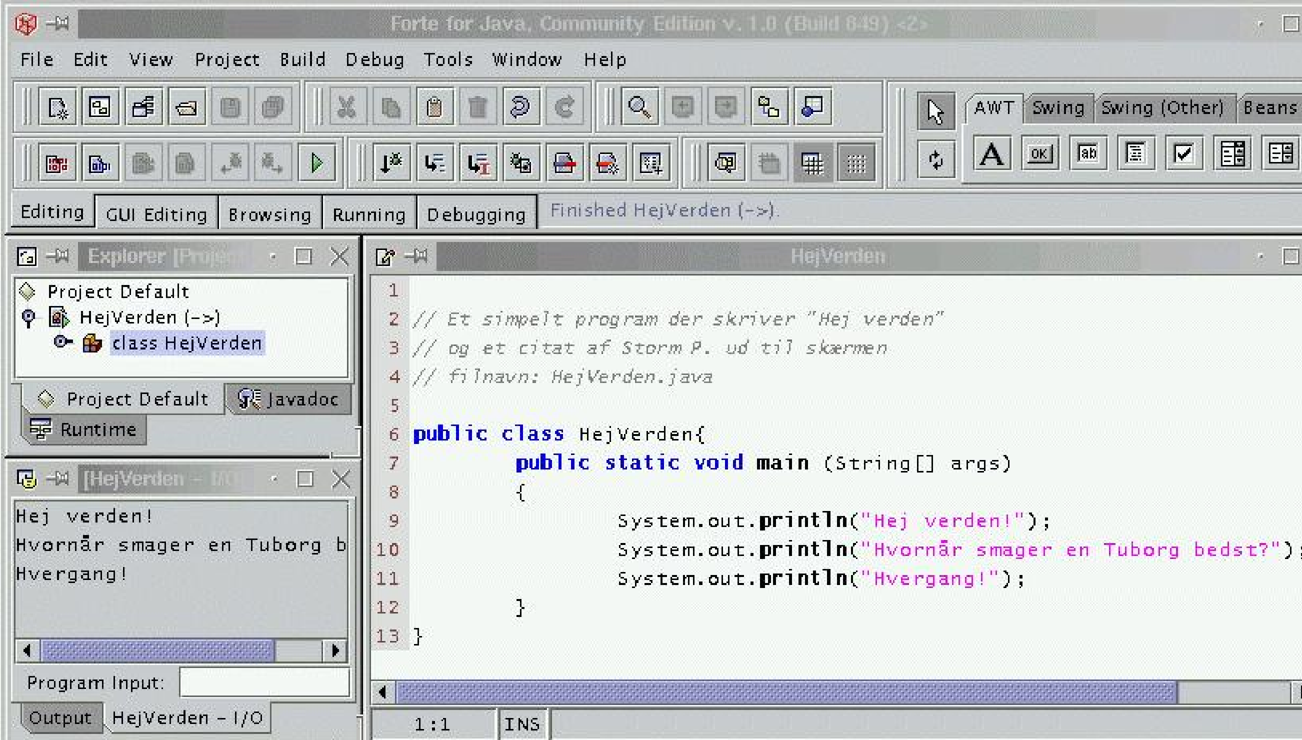


2.2.4. Sun Forte for Java

Sun udgiver sit eget udviklingsmiljø, også skrevet i Java, til Windows, Linux og Sun Solaris. Gratisversionen, der kan hentes på http://www.sun.com/forte/, har flere funktioner til at udvikle grafiske brugergrænseflader end JBuilders basisversion.

Hvilken en, der er bedst, afhænger nok af, hvem man spørger, men Forte virker efter forfatterens mening mindre gennemført end JBuilder. JBuilder fylder mere på disken, men har betydeligt mere hjælp, både til begynderen og den erfarne.

Figur 2-3. Sun Forte for Java

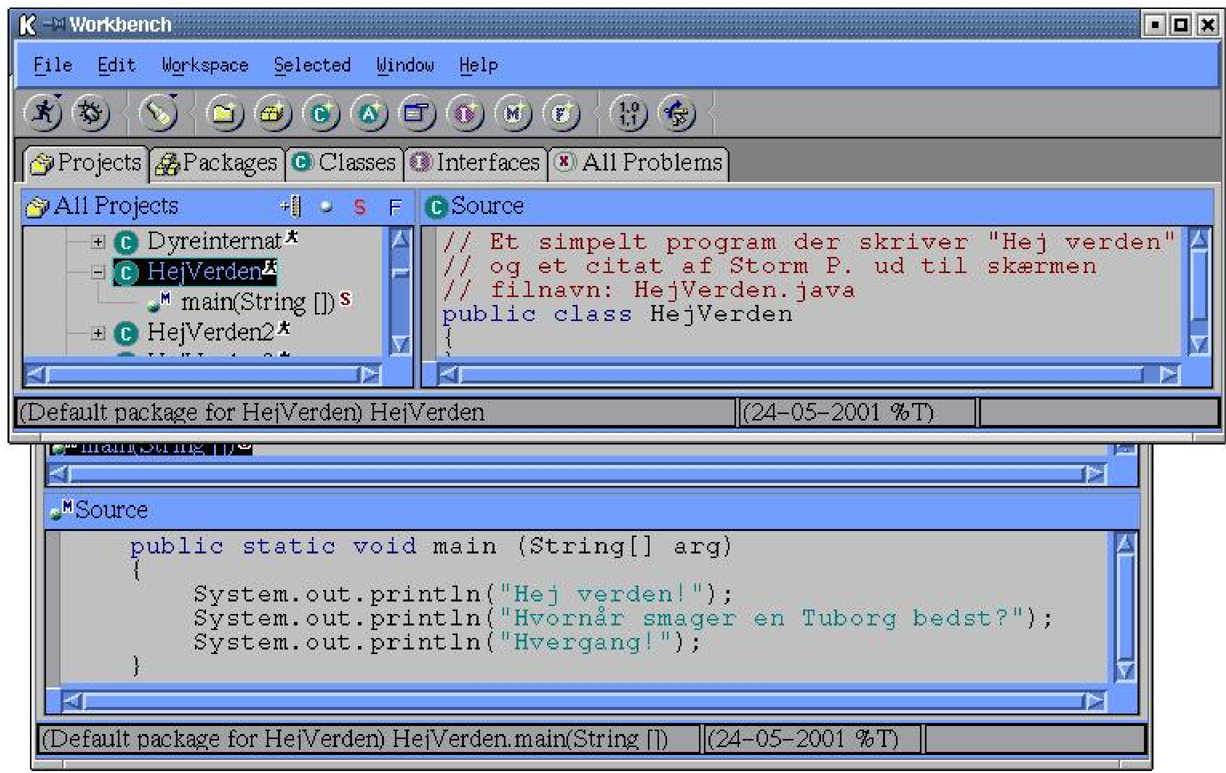


2.2.5. IBM VisualAge for Java

> **Bemærk:** VisualAge er blevet erstattet med WebSphere.

Ligesom Sun har IBM sit eget udviklingsmiljø til bl.a. Java, og en gammel udgave af miljøet kan hentes gratis på IBMs hjemmeside til bl.a. Linux (skærmbilledet viser VisualAge for Java under Linux). Gratisversionen kræver kun 32 MB RAM. De nyere udgaver er mere krævende (128 MB RAM).

Figur 2-4. IBM VisualAge for Java



2.2.6. Andre

Der findes mange andre udviklingsmiljøer til Java, bl.a. Visual Café, Simplicity, CodeGuide, AnyJ og Microsofts Visual J++. De fleste findes i en prøveudgave, der kan hentes gratis fra internettet, og som har alle nødvendige faciliteter til at lave mindre programmer.

2.3. Opsætningsstyring

Når du skal til at lave større og mere komplekse softwaresystemer, så vil du opdage at de består af mange forskellige delkomponenter. Delkomponenter vil i de fleste tilfælde være softwaremoduler med bestemte versionsnumre, men det kan også være brugervejledninger eller datafiler, som er knyttet til det pågældende softwaresystem.

Den proces at sammensætte komponenter så de udgør et softwaresystem kaldes for *konfigurering* og kan være en omfattende og ret så kedelig arbejdsopgave at udføre manuelt når man skal holde styr på 117 forskellige filer med bestemte versionsnumre og deres indbyrdes afhængigheder.

Heldigvis findes der værktøjer som er skræddersyet til konfigureringsstyring og vi vil i dette afsnit beskrive et værktøj, der er møntet specielt mod konfigurering af Java-softwaresystemer.

2.3.1. Ant

Ant er et Java-baseret værktøj. Ant kan hentes fra: http://jakarta.apache.org/ hvor man desunden kan finde yderligere dokumentation og information.

2.3.1.1. Typisk bibliotekslayout

TODO...skriv om motivationen for nedenstående layout /jk

```
projektnavn              Al projektrelateret data
projektnavn/build        Oversatte class-filer, ressourcefiler og billeder
projektnavn/dist         Binær distribution af softwaresystemet inkl. manualer osv.
projektnavn/docs         Al dokumentation
projektnavn/docs/api     JavaDoc-genereret HTML af din kode
projektnavn/lib          Eksterne jar-filer, som anvendes i projektet
projektnavn/log          Logfiler
projektnavn/src          Kildetekst, ressourcefiler og billeder
projektnavn/test/lib     Eksterne jar-filer, som anvendes til test
projektnavn/test/report  Testrapporter
projektnavn/test/src     Kildetekst relateret til test af softwaresystemet
```

2.3.1.2. Opsætningsfil

Nedenstående er vist et eksempel på en opsætningsfil til Ant som benytter ovenstående layout. For at anvende build.xml skal man kopiere build.xml til projekt katalog som man har valgt til dette project. Man kan benytte:

- **ant init** til at oprette ovenstående katalog struktur
- **ant compile** til at oversætte kildeteksten
- **ant dist** til at oprette en jar fil med klasse filerne
- **ant clean** sletter katalogerne dist og build
- **ant doc** til at generer javadoc. Du til skal dog erstatte *dk.cmm.** med dine egne pakker

Man kan utroligt mange forskellige ting med Ant. Lige fra at komplilere sin kildetekst, til at køre java-programmer, til test, til ftp med mere. I stedet for at lave sine egne scripts kan det anbefales at man ser på dokumentationen til Ant og finder den rette task således at man har samlet projektets opsætning på et sted.

```
<project name="antEksempel" default="compile" basedir=".">

<description>
  Eksempel paa Ant build.xml
</description>

<property name="name"       value="antEksempel"/>
<property name="packages"   value="dk.cmm.*"/>

<property name="build"      value="build"/>
<property name="dist"       value="dist"/>
<property name="docs"       value="docs"/>
<property name="api"        value="${docs}/api"/>
<property name="log"        value="log"/>
<property name="src"        value="src"/>
<property name="lib"        value="lib"/>
<property name="test"       value="test"/>
<property name="testlib"    value="${test}/lib"/>
<property name="testreport" value="${test}/report"/>
<property name="testsrc"    value="${test}/src"/>

<target name="init">
   <tstamp/>

   <mkdir dir="${build}"/>
   <mkdir dir="${dist}"/>
   <mkdir dir="${docs}"/>
   <mkdir dir="${api}"/>
   <mkdir dir="${log}"/>
   <mkdir dir="${lib}"/>
   <mkdir dir="${src}"/>
   <mkdir dir="${test}"/>
   <mkdir dir="${testlib}"/>
   <mkdir dir="${testreport}"/>
   <mkdir dir="${testsrc}"/>
</target>

<path id="classpath">
   <fileset dir="lib">
      <include name="**/*.jar"/>
   </fileset>
</path>

<target name="compile"
        depends="init"
 description="compile the source " >
    <javac srcdir="${src}"
           destdir="${build}"
```

```
    classpathref="classpath"/>
</target>

<target name="dist"
        depends="compile"
 description="Generates a jar file" >
    <jar jarfile="${dist}/${appName}-${DSTAMP}.jar" basedir="${build}"/>
</target>

<target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
</target>

<target name="doc"
 description="generates api documentation">
   <javadoc packagenames="${packages}"
            destdir="${api}"
            sourcepath="src"/>
</target>

</project>
```

2.4. Debugging

TODO

jdb

2.5. Logning

TODO

log4j, JDK log api (JSR47)

JSR47 vs. log4j (take two). http://jakarta.apache.org/log4j/docs/critique2.html

2.6. Profilering

TODO

java -Xprof JavaProgram

2.7. Dokumentation

Dokumentation er et vigtigt aspekt af ethvert softwareudviklingsprojekt. I dette afsnit fokuserer vi på et værktøj, der kan hjælpe dig med at udarbejde API-dokumentation.

API-dokumentation består af en API-specifikation samt en vejledning i at bruge API'en. API-specifikationen skal ses som en kontrakt mellem den som bruger API'en (klienten) og den som implementerer API'en (leverandøren). Hvis leverandøren f.eks. lover at metoden `double sqr(int number)` returnerer kvadratroden af `number` når tallet ikke er negativt så ved klienten at uanset, hvordan metoden er implementeret så vil den altid returnere kvadratroden med så stor præcision, som returtypen tillader. Hvis returværdien viser sig at være forkert i nogle tilfælde så har leverandøren brudt kontrakten og metoden er implementeret forkert. Hvis klienten kalder metoden med et negativt tal så er der også tale om kontraktbrud. Kontraktbrud kan enten resultere i en exception eller også er resultatet ikke veldefineret.

2.7.1. Javadoc

Javadoc er en dokumentationsværktøj, der genererer API-dokumentation ud fra

- Specielle *Javadoc-kommentarer* indeholdt kildeteksten. Disse udgør API-specifikationen, men kan også indeholde brugsvejledninger.
- *Pakkedokumentationsfiler*, der indeholder overordnet dokumentation for pakkerne.
- *Overbliksfiler*, der indeholder overordnet dokumentation om en mængde af pakker.

Denne sektion er ment som en hjælp til at få dig i gang med at bruge Javadoc så der er flere aspekter, som vi vil springe over. Når du har fået blod på tanden så kan du læse JDK's *Tool documentation* eller besøge nedenstående hjemmesider.

- Javadoc Tool Home Page, http://java.sun.com/j2se/javadoc/
- How to Write Doc Comments for Javadoc, http://java.sun.com/j2se/javadoc/writingdoccomments/

2.7.1.1. Javadoc-kommentarer

Javadoc-kommentarerne er på formen

```
/** et-eller-andet */
```

og kan knytte sig til klasser, interfaces, konstruktører, metoder og klassevariable. Alt afhængigt af hvad de knytter sig til så kan/skal man også angive en række beskrivende *Javadoc-tags* som er på formen.

```
@tagnavn et-eller-andet
```

Følgende eksempel illustrerer brugen af Javadoc-kommentarer og Javadoc-tags.

```
package dk.sslug;

/**
 * Denne klasse repræsenterer en simpel stak, der
 * lagrer heltal.
 *
 * @author Jonas Kongslund (jonas@kongslund.dk)
 * @version 1.1
 */
public class IntStack
{
  /** Antal elementer i stakken */
  protected int count;

  /**
   * Indeholder stakkens elementer. Elementerne er placeret
   * i <code>elements[0...count-1]</code>.
   * <p>
   * Toppen af stakken er <code>count-1</code>
   * når <code>count>0</code> og ellers udefineret.
   *
   * @see #pop()
   * @see #push(int)
   */
  protected int[] elements;

  /**
   * Standardkonstruktør for denne klasse.
   */
  public IntStack()
  {
    /* Øvelse: implementer metoden sådan at
       elements og count initialiseres til
       nogle fornuftige værdier */
  }

  /**
   * Fjerner og returnerer det øverste tal på stakken.
   *
   * @return int Det øverste tal på stakken
   * @exception java.util.EmptyStackException
   *            hvis stakken er tom
   */
```

```
  public int pop() throws java.util.EmptyStackException
  {
    /* Øvelse: implementer metoden */
    return -1;
  }

  /**
   * Placerer det angivne tal øverst på stakken.
   *
   * @param element Tallet der skal lægges på stakken
   */
  public void push(int element)
  {
    /* Øvelse: implementer metoden så stakken
       udvides såfremt den er fyldt */
  }

  /**
   * Placerer det angivne tal øverst på stakken.
   *
   * @param element Tallet der skal lægges på stakken
   * @deprecated Siden version 1.1; Metoden er
   *   erstattet af <code>push(int)</code>.
   * @see #push(int)
   */
  public void skub(int element)
  {
    push(element);
  }
}
```

Bemærk at vi i eksemplet bruger HTML-tags til at fremhæve blandt andet metodenavne. Dette er både tilladt og anbefalelsesværdigt pga. øget læsevenlighed i API-dokumentationen.

Det er desuden anbefalelsesværdigt at lade den første sætning i hver kommentar være beskrivende nok til at man hurtigt får en idé om, hvad eksempelvis en metode gør. Javadoc-værktøjet forventer faktisk dette da den bruger den første sætning til at generere oversigtslister med.

Prøv at kalde Javadoc-værktøjet med følgende argumenter

```
[jonas@zeta eksempler/dev-env]$  javadoc -author -version dk.sslug
```

Værktøjet genererer som standard HTML-filer. Åbn `index.html` med en browser for at se resultatet.

> **Bemærk:** Hvis du har brugt Javadoc på `eksempler/dev-env/`-kataloget så følger der også pakkedokumentation med i købet, selvom den ikke er medtaget i eksemplet foroven. Hvordan du laver pakkedokumentation beskrives senere i denne sektion.

I eksemplet er der knyttet Javadoc-kommentarer til metoder og klassevariable samt klassen selv. I de fleste af kommentarerne anvendes der Javadoc-tags.

- @*author*, der er obligatorisk, angiver ophavsmanden er for Javadoc-kommentarernes vedkommende beregnet til klasser og interfaces . Det er muligt at angive flere @author-tags på separate linjer såfremt der er flere ophavsmænd.
- @*version*, der er obligatorisk, angiver versionsnummeret og er for Javadoc-kommentarernes vedkommende beregnet til klasser og interfaces. Versionsnummeret har ikke nogen speciel betydning og kan derfor være hvad som helst.
- @*since*, angiver i hvilken version af API'en som tilføjelsen fandt sted og kan bruges overalt. Versionsnummeret har ikke nogen speciel betydning og kan derfor være hvad som helst.
- @*param* beskriver et argument og er beregnet til metoder og konstruktører. Først angives argumentets navn og dernæst beskrivelsen.
- @*return* beskriver returværdien og er beregnet til metoder. Først angives returtypen og dernæst beskrivelsen.
- @*exception* beskriver en exception og er beregnet til metoder og konstruktører. Der kan naturligvis være flere @exception-tags og hvis du synes @*throws* virker mere logisk så kan du også anvende denne.
- @*see* henviser til en pakke, klasse, klassevariabel, interface, konstruktør eller metode. En af de mulige henvisningsformer er `pakkenavn.klassenavn#medlem`, hvor medlem kan være et metodenavn på formen `flaf(type1,type2,...)` eller et variabelnavn. Pakkenavn og klassenavn kan udlades, hvis der henvises til en metode eller klassevariabel indenfor samme klasse. Dette Javadoc-tag kan anvendes i alle Javadoc-kommentarer.
- @*deprecated* kan bruges overalt og indikerer f.eks. at en metode ikke længere bør anvendes fordi den vil udgå i en senere version. Det er anbefalelsesværdigt at henvise til et alternativ ved hjælp af @see.

2.7.1.2. Pakkekommentarer

En pakkekommentar laves ved at oprette en HTML-fil kaldet `package.html`, der placeres i pakkens katalog. Hvis pakken f.eks. hedder `dk.sslug` så placeres filen i `dk/sslug`. Javadoc-værktøjet sørger automatisk for at medtage filen.

Forneden har vi et eksempel på en pakkekommentar.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>dk.sslug</title>
  </head>
  <body>
    Tilbyder containerklasser, som kan lagre forskellige typer
    data. Indtil videre er der kun én klasse, IntStack, men
    dette bliver der ændret på i næste version.

    @see dk.sslug.IntStack
```

```
    @since 1.0
  </body>
</html>
```

Javadoc-værktøjet bruger kun det der står mellem <body> og </body>, så titlen kan være hvad som helst.

Ligesom med Javadoc-kommentarer er det muligt at anvende Javadoc-tags i pakkekommentarer. Blandt dem der er nævnt tidligere kan du anvende

- @author (ikke obligatorisk som hos Javadoc-kommentarer)
- @version (ikke obligatorisk som hos Javadoc-kommentarer)
- @since
- @deprecated
- @see

Bemærk at pakkekommentarer må ikke indeholde `/** ... */` og en linje må ikke starte med `*`.

2.7.1.3. Linke til andre API-dokumenter

I eksemplet foroven kan `pop()`-metoden smide en `java.util.EmptyStackException`, men denne har vi ikke selv lavet og derfor linker API-dokumentationen ikke til den. Dette kunne dog godt være ønskværdigt såfremt man ikke lige kan huske hvad EmptyStackException dækker over (hvilket i dette tilfælde dog nok er ret usandsynligt).

Javadoc-værktøjet er så smart at det kan linke din API-dokumentation til klasser og interfaces, der er indeholdt i andre API-dokumenter. Det eneste man skal gøre er at fortælle Javadoc, hvor den eksterne dokumentation befinder sig.

Her er et eksempel, der forudsætter at JDK's API-dokumentation er indeholdt i `$JAVA_HOME/docs/api`.

```
[jonas@zeta eksempler/dev-env]$  javadoc -link $JAVA_HOME/docs/api dk.sslug
```

Du kan også angive en URL.

```
[jonas@zeta eksempler/dev-env]$  javadoc -link \
http://java.sun.com/products/jdk/1.3/docs/api/ dk.sslug
```

I begge tilfælde bliver der kun lavet links til dokumentationen for en ekstern klasse eller interface såfremt følgende er opfyldt

- Den er eksplict angivet i en import-sætning, f.eks. `import java.util.EmptyStackException;`. Det er ikke nok at der står `import java.util.*;`.

- Den er angivet som returtype eller argumenttype i en metode eller konstruktør.
- Den anvendes i en implements-, extends- eller throws-sætning, f.eks. `public int pop() throws java.util.EmptyStackException`

I eksemplet er det tilfælde tre som gør sig gældende.

2.8. Versionsstyring

TODO

CVS. www.linuxbog.dk/program/bog/vaerktoej.html

2.9. Test

Kunsten at lave fejlfri programmer er svær at mestre. Kunsten at lave robuste programmer er svær at mestre. Kunsten at lave programmer der svarer til kundernes forventning er svær at mestre. Netop derfor er det nødvendigt at ethvert softwareprojekt i et eller andet omfang gennemtestes for fejl og mangler inden det sendes ud på markedet.

En *modultest* er en test, der afprøver en afgrænset delmængde kode i isolation fra det samlede softwaresystem. Målet er at finde fejl og sandsynliggøre at modulet fungerer efter hensigten. Når vi taler Java så er et modul som oftest enten en klasse eller en samling af klasser hørende under en fælle pakke.

I denne sektion ser vi på et yderst værdifuldt værktøj til automatiseret modultest, *JUnit*.

2.9.1. JUnit

JUnit er et testframework, der har til formål at lette programmørens arbejde når der skal udarbejdes testmetoder. Frameworket er centreret omkring begrebet *Test Case* som repræsenterer en samling af testmetoder for et enkelt modul. På dansk kaldes en test case for en *testsamling*.

Du kan ganske gratis hente JUnit fra http://www.junit.org (http://www.junit.org/). På selvsamme adresse kan du også finde mange gode artikler og eksempler. Gennemgangen i denne sektion er kun ment som en hurtig introduktion så det kan varmt anbefales selv at grave efter mere information.

2.9.1.1. Test af IntStack

I Afsnit 2.7.1 blev der defineret en stak, `IntStack`, der kan lagre heltal. Nedenstående kode viser et eksempel på, hvordan JUnit kan anvendes til at teste denne stak.

For at kunne oversætte og udføre eksemplet så skal jar-filen `junit.jar` være med i CLASSPATH.

```
package dk.sslug;

import junit.framework.TestCase;
import dk.sslug.IntStack;

public class TestIntStack extends TestCase
{
  public TestIntStack(String name)
  {
    super(name);
  }

  public void testPushPop()
  {
    IntStack stack = new IntStack();
    stack.push(10);
    stack.push(20);
    stack.push(30);
    assertTrue( stack.pop() == 30 );
    assertTrue( stack.pop() == 20 );
    assertTrue( stack.pop() == 10 );
  }

  public void testEmptyStackException()
  {
    IntStack stack = new IntStack();
    try {
      stack.pop();
      fail("Burde have smidt en EmptyStackException");
    } catch (java.util.EmptyStackException e) {
    }
  }
}
```

Der er flere ting du skal bide mærke i. For det første skal en testklassen nedarve fra `TestCase`-klassen. For det andet skal testmetoderne starte med navnet *test* for at testframeworket kan udføre dem. For det tredje skal testmetoderne være deklareret public og ikke tage imod nogen argumenter.

Metoden `assertTrue(boolean)` bruges til at fortælle testframeworket, hvorvidt en test skal fejle eller ej. Hvis det boolske udtryk evaluerer til falsk så fejler testen.

Der findes flere forskellige `assertX()`-metoder (læs dokumentationen!). De mest brugte udover `assertTrue(boolean)` er nok

- `assertEquals(java.lang.Object expected, java.lang.Object actual)`
- `assertNotNull(java.lang.Object object)`
- `assertNull(java.lang.Object object)`

`fail()`-metoden får altid en test til at fejle.

Hvis ovenstående eksempel oversættes og køres med

```
[jonas@zeta eksempler/dev-env]$  java junit.textui.TestRunner \
dk.sslug.TestIntStack
```

så fås

```
.F.F
Time: 0.029
There were 2 failures:
1) testPushPop(dk.sslug.TestIntStack)junit.framework.AssertionFailedError: expected:<-1> bu
        at dk.sslug.TestIntStack.testPushPop(TestIntStack.java:27)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:42)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
2) testEmptyStackException(dk.sslug.TestIntStack)junit.framework.AssertionFailedError: Burd
        at dk.sslug.TestIntStack.testEmptyStackException(TestIntStack.java:37)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:42)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav

FAILURES!!!
Tests run: 2,  Failures: 2,  Errors: 0
```

Hvis der smides en runtime exception i testmetoden som ikke fanges så fejler testen også. I JUnit-terminologi er fejl forårsaget af `assertX()`- og `fail()`-metoder *failures* men runtime exceptions der ikke bliver fanget er *errors*.

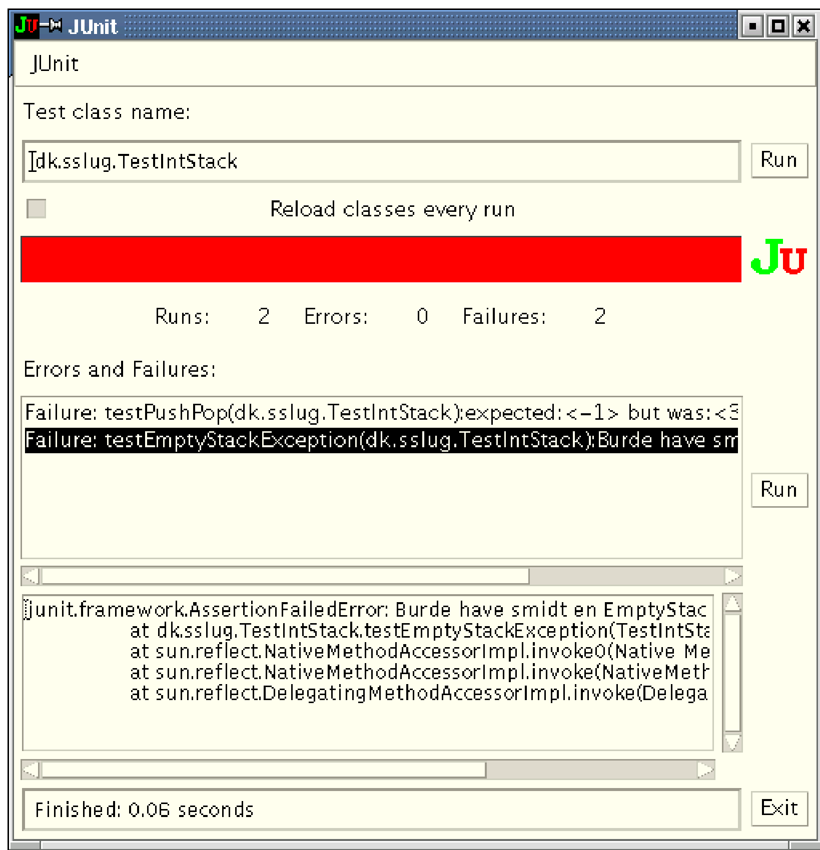
Du kan lege med at implementere `IntStack` i henhold til API-specifikationen og se om din implementation indeholder fejl.

Du kan også bruge en grafisk TestRunner. Hvis du kører nedenstående

```
[jonas@zeta eksempler/dev-env]$  java junit.awtui.TestRunner \
dk.sslug.TestIntStack
```

så fås

Figur 2-5. Grafisk JUnit TestRunner



2.9.1.2. Fælles initialisering for metoder

Ofte er man ude for at have to eller flere testmetoder, som skal operere på den samme kendte mængde af objekter. I JUnit-terminologi kaldes denne mængde af objekter for et *test fixture* og skal betragtes som en slags fast inventar som alle testmetoder kan benytte sig af. Ligeledes er det ikke sjældent at to eller flere testmetoder skal gøre brug af en fælles ressource, for eksempel en netværksforbindelse eller en databaseforbindelse.

Nedenstående eksempel viser hvordan dette kan gøres ved at overskrive metoderne `setUp()` og `tearDown()` og benytte sig af klassevariable.

```
package dk.sslug;

import junit.framework.TestCase;
import dk.sslug.IntStack;
```

```
public class TestIntStack extends TestCase
{
  IntStack nonempty_stack;
  IntStack empty_stack;

  public TestIntStack(String name)
  {
    super(name);
  }

  protected void setUp() throws Exception
  {
    nonempty_stack = new IntStack();
    nonempty_stack.push(10);

    empty_stack = new IntStack();
  }

  protected void tearDown() throws Exception
  {
    // intet behov for oprydning
  }

  ...
}
```

Hver gang en testmetode skal udføres så tages `setUp()` og `tearDown()` i brug. `setUp()` bliver kaldt umiddelbart før den enkelte testmetode, mens `tearDown()` bliver kaldt umiddelbart efter.

Bemærk at begge metoder er deklareret som protected og sat til at kunne kaste en exception.

Kapitel 3. Basal programmering

Kapitlet forudsættes i resten af bogen.

3.1. Det første javaprogram

Lad os se på et simpelt javaprogram, der skriver "Hej verden" og et citat af Storm P. ud til skærmen.

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

Inden vi oversætter og kører programmet

3.1.1. Kommentarer

Kommentarer er dokumentation beregnet på at gøre programmets kildetekst lettere at forstå. De påvirker ikke programudførelsen, da oversætteren ignorerer dem.

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
```

Kommentarer bør skrives, så de giver forståelse for, hvordan programmet virker - uden at være flertydige 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
*/
```

og

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

Der findes også en tredje form kaldet for *Javadoc-kommentarer*, som starter med /** og slutter med */. Javadoc-kommentarer gennemgåes i Afsnit 2.7.1.

3.1.2. 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!");
```

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

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

Programudførelsen starter i metoden: public static void main (String[] arg)

3.1.2.2. 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 (data), som man overfører til metoden. I dette tilfælde hvilken tekst der skal skrives ud til skærmen.

Vores main-metode kalder altså andre metoder.

3.1.3. 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 (kildeteksten), som centralenheden i computeren arbejder med, er i en binær kode (ene ettaller og nuller) Nævnte binære kode kaldes maskinkode eller bytecode). En sådan kode er umulig at læse for almindelige mennesker. Kildeteksten skal derfor oversættes (eng.: compile; mange siger også kompilere på dansk) til binær kode, som så kan udføres af computeren.

I Java kalder man den binære kode for bytekode. Bytekode er platform-uafhængigt, 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" - HejVerden er filnavnet, mens java er filtypen (efternavnet). Filnavnet skal være det samme som klassenavnet - her er det ikke nok at de samme tegn optræder i filnavnet, sammensætningen af store og små bogstaver skal identisk med klassenavnet. Filtypen skal være java. Man kan godt havde flere klasser i samme fil, men kun én kan have offentlig tilgang (public) filnavnet skal opkaldes efter denne - læs mere om dette senere.

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

3.1.3.1. Oversættelse og kørsel uden et udviklingsværktøj

Hvis man bruger det kommandolinje-orienterede JDK, skal man (fra samme katalog) skrive:

```
[jonas@zeta basal-prg]$ javac HejVerden.java
```

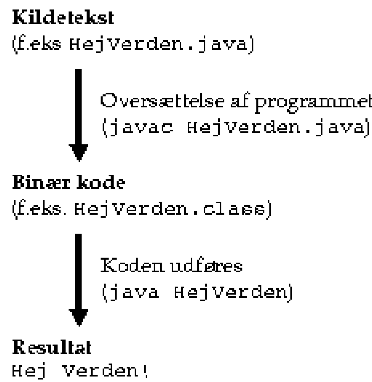
Såfremt der ikke er nogen fejl i kildeteksten så vil oversætteren forholde sig tavs. Det eneste tegn på at der er sket noget er at filen `HejVerden.class` er dukket op i samme katalog. Nu kan det køres med kommandoen

```
[jonas@zeta basal-prg]$ java HejVerden
```

Resultatet bliver:

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

Figur 3-1. Oversættelse og udførelse af programmet HejVerden



3.1.3.2. 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 det katalog, 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*.

3.2. Variable

Du kan vælge at give et område i computeren et navn - sætte en etiket på området. Området kan bruges til opbevaring af data. Da alle data ikke fylder lige meget i lageret, er det nødvendigt at definere, hvilken type data du vil bruge lageret til samtidig med erklæringen eller navngivningen af området.

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

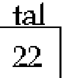
I det følgende gennemgår vi to af Javas variabeltyper: int (heltal) og double (kommatal).

3.2.1. Heltal

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

```
int tal;
```

Figur 3-2. Efter 1. tildeling



Nu er der reserveret plads i hukommelsen til et heltal. Man får fat i pladsen ved at bruge variabelnavnet 'tal'. Efter at variablen er erklæret kan den tildeles en værdi (der kan lagres en værdi på dens plads i computerens RAM), 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 om lidt).

Variabler kan, som navnet siger, ændre værdi. Det gør vi ved at tildele variablen en ny værdi (der kan gemmes en værdi på den plads i computerens lager, der er "markeret" med etiketten tal):

Figur 3-3. Efter 2. tildeling

```
tal = 42;
```

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

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

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

```
// Eksempel på brug af en variabel
// koden skal være i filen Variabler.java
public class Variabler
{
  public static void main (String[] args)
  {
    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
```

3.2.2. 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[] args)
  {
    System.out.println("Hej " + "Verden!");
```

```
  }
}
```

Resultatet bliver

```
Hej Verden!
```

Herunder skriver vi en streng og tallet 42 ud:

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

Resultatet bliver

```
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

3.2.3. 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 (fordi variabelnavnet ene og alene er en etiket sat på en del af computerens lager, er det naturligvis ikke muligt at have beregningsudtryk på venstre side):

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

Beregningsudtrykkene undersøges af Java ved at indsætte de værdier, der er gemt i variablerne. Her indsætter Java 5 + 8 og får 13, som lægges i antalDyr.

```
public class Dyreinternat
{
  public static void main(String[] args)
  {
    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);
  }
}
```

Resultatet bliver

```
Antal dyr: 13
Antal dyr nu: 13
```

Beregningen sker én gang *på det tidspunkt, hvor kommandoen udføres* Derfor er antalDyr's værdi ikke påvirket at af 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.

Et heltal divideret med et heltal giver et heltal 95 / 100 giver 0

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

3.2.4. Kommatal

Der findes mange andre variabeltyper end heltalstypen int. Hvis man vil regne med kommatal, 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.

Her er et eksempel på beregning af en cylinders rumfang:

```
//
// Beregning af rumfang for en cylinder
//
public class Cylinderberegning
{
  public static void main(String[] args)
  {
    double radius;
    radius = 5.0;
    double højde = 12.5;
    //beregn 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[] args)
  {
    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;
```

3.2.5. Matematiske funktioner

De matematiske funktioner som sinus, cosinus, kvadratrod osv. kaldes i Java med Math.sin(x), Math.cos(x), Math.sqrt(x) osv., hvor x er en variabel, et fast tal eller et beregningsudtryk.

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

```
public class Kvadratrod
{
  public static void main(String[] args)
  {
    System.out.println("kvadratroden af 0 er " + Math.sqrt(0));
```

```
    System.out.println("kvadratroden af 1 er " + Math.sqrt(1));
    System.out.println("kvadratroden af 2 er " + Math.sqrt(2));
    System.out.println("kvadratroden af 3 er " + Math.sqrt(3));
    System.out.println("kvadratroden af 4 er " + Math.sqrt(4));
    System.out.println("kvadratroden af 5 er " + Math.sqrt(5));
    System.out.println("kvadratroden af 6 er " + Math.sqrt(6));
    System.out.println("kvadratroden af 7 er " + Math.sqrt(7));
    System.out.println("kvadratroden af 8 er " + Math.sqrt(8));
    System.out.println("kvadratroden af 9 er " + Math.sqrt(9));
    System.out.println("kvadratroden af 10 er " + Math.sqrt(10));
  }
}
```

Resultatet bliver

```
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
```

Her er et program, der udregner længden af den skrå side (hypotenusen) af en retvinklet trekant ud fra længden af dens to lige sider (kateder): kvadratroden af a^2+b^2, hvor a og b er længderne af de to sider:

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

Resultatet bliver

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

Her er et tilsvarende program, der udregner hypotenusen ud fra længden af en af de andre sider og den modstående vinkel. Det gøres ud fra formlen a/sin(v), hvor a er længden af siden, og v er vinklen (i radianer):

```
public class Trekant2
{
  public static void main(String[] args)
  {
    double a, v, hypotenuse;
    a = 10;
    v = 0.3; // svarer til ca. 34 grader.
    hypotenuse = a/Math.sin(v);
    System.out.println("Hypotenusen har længden: "+hypotenuse);
  }
}
```

Resultatet bliver

```
Hypotenusen har længden: 33.838633618241225
```

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

3.2.6. Kald af metoder

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

En metode er en navngiven programstump (i nogle computersprog taler man på dette sted om funktioner), 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 kvadratrod. 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 = a/Math.sin(v);
```

Math.sin er navnet på metoden, og man kalder det v, 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 kald uden parametre skal man stadig have parenteserne med. Math.random() skal kaldes uden parametre. Her er et eksempel på et metodekald af Math.random():

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

Ved et metodekald kan man indsætte som parameter 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.sin(0.1);             // konstant som parameter
x = Math.sin(v);               // variabel som parameter
x = Math.sin(Math.sqrt(0.3));  // værdi af andet metodekald som parameter
```

Vi vil i Kapitel 5 se, hvad der sker, når computeren udfører et metodekald, samt lære, hvordan man kan lave sine egne metoder.

3.2.7. Logiske variable

En boolsk variabel, også kaldet en logisk variabel, kan kun indeholde værdierne sand eller falsk. Den bruges oftest til at huske, om noget er sandt eller ej, men kan også bruges til at 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=7, står der 7>8, det er ikke sandt, og

detErForSent får værdien false. Hvis klokken=10, står der 10>8, det er sandt, og detErForSent får værdien true.

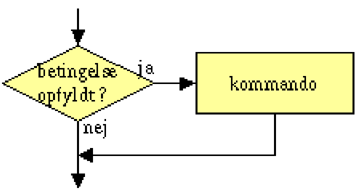
3.2.8. Opgaver

1. Skriv et program, som ud fra længde og bredde på et rektangel udskriver rektanglets areal.
2. Skriv et program, som for ligningen y=3*x*x+6*x+9 udskriver værdierne af y for x=0, x=1 og x=10.
3. Skriv et program, som omregner et beløb fra dollar til euro (f.eks. kurs 95).

3.3. 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.

Figur 3-4. Logikken i en if-sætning



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 Figur 3-4).

Alle er bekendte med 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

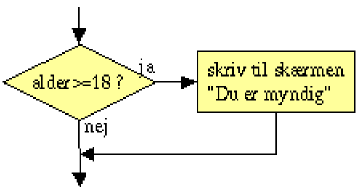
```
if (betingelse) kommando;
```

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 boolske udtryk). På dansk er sætningen "over 18" tvetydig: skal man være OVER 18, dvs. 19, for at være myndig? Java har derfor to forskellige sammenligningsoperatorer: a >= b undersøger, om a er større end eller lig med b, mens a > b undersøger om a er større end b. I appendiks Afsnit 3.9.4 findes en oversigt over sammenligningsoperatorerne.

Figur 3-5. Rutediagram for Alder



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 et javaprogram skriver man:

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

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

Resultatet bliver

```
Du er 15 år.
```

Kommandoen System.out.println("Du er myndig"), bliver kun udført, hvis udtrykket (alder > 18) er sandt. I dette tilfælde er alder = 15, og der bliver ikke skrevet noget ud. Hvis vi ændrer i programmet, så alder = 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 linjen

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

blive udført.

Bemærk, oversætteren er ligeglad med indrykning, linjeskift etc. Det er teksten, der tæller. Vi kunne lige så godt have skrevet

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

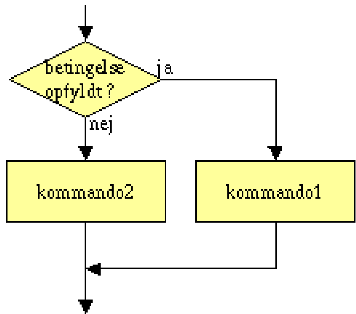
eller for den sags skyld

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

... men det sidste er en dårlig stil, for det gør det sværere at læse kildeteksten. Normalt indrykker man 2-4 tegn, når en kommando er betinget.

3.3.1. if-else

Figur 3-6. Logikken i en if-else-sætning



Hvis vi ønsker at gøre én ting, hvor 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[] args)
  {
    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 så alder = 18, er betingelsen (alder >= 18) sand, og vi får resultatet

```
Du er myndig.
Du er 18 år.
```

Det er selvfølgelig lidt kedeligt at kunne se direkte i vores program uden at køre det, om en betingelse er opfyldt. I virkeligheden er betingelser kun nyttige, når forgreningen afhænger af nogle ydre omstændigheder, f.eks. af brugerens indtastning eller af værdien af en variabel fra en anden del af programmet. Hvis du bare vil prøve, kan du lave et ikke-forudsigeligt program med Math.random().

3.3.2. Opgaver

1. Lav et veksleprogram fra dollar til euro. Det skal påregne en kommission på 2 %, dog mindst 0,5 euro. Afprøv programmet med forskellige beløb.

2. Skriv et program, der beregner porto for et brev. Inddata er brevets vægt (i gram). Uddata er prisen for at sende det som A-post i Danmark.

3.4. Blokke

En blok er en samling af kommandoer. Den starter med en blokstart-parentes { og slutter med en blokslut-parentes}.

En blok grupperer flere kommandoer, så de udføres samlet som én kommando

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

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

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

Resultatet bliver

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

3.4.1. 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.

Det er god skik at bruge indrykning i en blok

Indrykning gør programmet meget nemmere at overskue

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

```
public class Alder3UheldigIndrykning{
public static void main(String[] args)
{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");
}}}
```

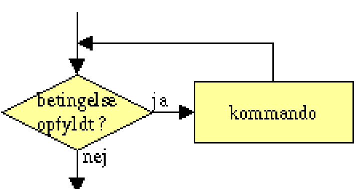
De fleste udviklingsværktøjer har funktioner til at rykke flere linjers kode ind og ud (i JBuilder gøres det ved at markere en tekst, og trykke Tab for at rykke ind og skift-Tab for at rykke ud).

3.5. Løkker

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

3.5.1. while-løkken

Figur 3-7. Logikken i en while-løkke



while-løkken har formen:

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

Kommandoen udføres igen og igen mens betingelsen er opfyldt. Dvs. før kommandoen udføres, undersøges betingelsen, og det kontrolleres, at den er opfyldt (se Figur 3-7).

Oftest grupperer man flere kommandoer i en blok.

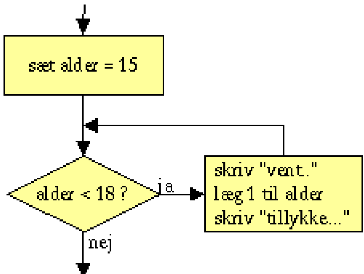
```
public class Alder4
{
  public static void main(String[] args)
  {
    int alder;
    alder = 15;

    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.");
  }
}
```

Resultatet bliver

```
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.
```

Figur 3-8. Rutediagram for noget af Alder4



Før løkken starter, har alder en startværdi på 15. Under hvert gennemløb tælles variablen 1 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 med Kvadratrod Afsnit 3.2.5).

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

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

Resultatet bliver

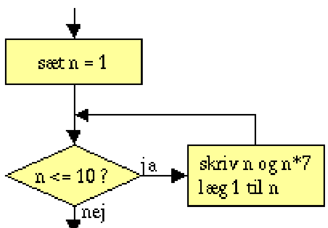
```
kvadratroden af 0 er 0.0
kvadratroden af 1 er 1.0
kvadratroden af 2 er 1.4142135623730951
```

```
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
```

En tællevariabel er en variabel, der tælles op i en løkke, indtil den når en bestemt øvre grænse. I eksemplerne ovenfor bruges alder og n som tællevariabler.

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

Figur 3-9. Rutediagram for Syvtabel




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

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

Resultatet bliver

```
1 : 7
2 : 14
3 : 21
```

```
4 : 28
5 : 35
6 : 42
7 : 49
8 : 56
9 : 63
10 : 70
```

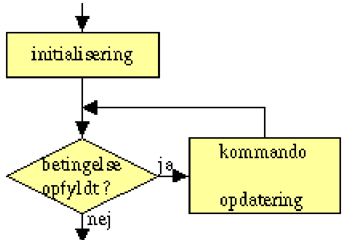
Tællevariabel-formen er den mest almindelige for løkker, men man kan sagtens komme ud for andre former for løkker. Der kan f.eks. godt indgå et regneudtryk i betingelsen.

3.5.2. for-løkken

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

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

Figur 3-10. Strukturen i en for-løkke

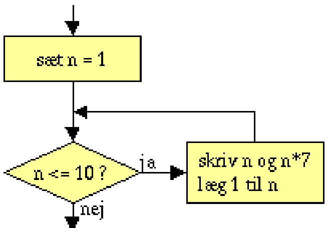


- *initialisering* er en (evt.: erklæring og) tildeling af en tællevariabel, f.eks. `alder = 15`
- *betingelse* er et logisk udtryk, der angiver betingelsen for, at løkken skal fortsætte med at blive udført, f.eks. `alder < 18`
- *opdatering* er ændringen i tællevariablen, f.eks. `alder = alder + 1`

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:

Figur 3-11. Rutediagram for Syvtabel2 (samme som for Syvtabel)




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

Programmører 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 en ned, f.eks. alder--.

3.5.3. 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 syvtabellen igen, men denne gang "brokker" programmet sig, når det når op på 6, og efter 8 skriver den "ved ikke" i stedet for at regne resultatet ud.

```
public class Syvtabel3
{
```

```
  public static void main(String[] args)
  {
    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)");
    }
  }
}
```

Resultatet bliver

```
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)
```

Vi kan også lave løkker i løkker. Herunder udregner vi n*7 ved at lægge 7 sammen n gange.

```
public class Syvtabel4
{
  public static void main(String[] args)
  {
    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);
    }
  }
}
```

Resultatet bliver

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

```
5 : 35
6 : 42
7 : 49
8 : 56
9 : 63
10 : 70
```

3.5.4. 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[] args)
  {
    for (int n=1; n<=10; n=n-1)
      System.out.println(n+" : "+ 7*n);
  }
}
```

Resultatet bliver

```
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 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.

3.5.5. Opgaver

1. Prøv at køre hvert eksempel, og forvis dig om, at du forstår det.Mange udviklingsværktøjer understøtter trinvis gennemgang til fejlfinding (eng.: debugging). Prøv trinvis gennemgang i dit værktøj, og hold øje med variablerne. (I JBuilder og JDeveloper gøres det med F8 "step over").
2. Omskriv Alder4-programmet til at bruge en for-løkke.
3. Lav et program, der tæller nedad fra 10 til 1.
4. Lav et program, der udregner værdien af 1+2+3+ ... +20.
5. Ret programmet til at udregne værdierne af 1+2+3+ ... +n, når n er 10, 11, .. 30 (vink: brug en indlejret løkke).
6. Lav et program, der udskriver 1-tabellen, 2-tabellen, .. op til 10-tabellen.
7. Skriv et program, som for ligningen y=3*x*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.

3.6. Værditypekonvertering

Java er det, man kalder et stærkt typet sprog. Det betyder, at alle variabler og værdier har en bestemt type gennem hele deres levetid, og at der er visse 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;  // Fejl.
```

Forsøger man på dette, vil man få oversætter-fejlen: *Possible loss of precision: double, required: int*.

Årsagen til, at vi i Java ikke kan gemme 2.7 i x, kan forstås på to måder:

1. x har kun plads i lageret til at gemme hele tal (fra -2 mia. til +2 mia).
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.

Begge måder at forstå årsagen på er rigtige og gyldige.

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. Dette er ikke helt uden problemer. Der er åbenlyst et informationstab, da kommadelen af værdien må fjernes. Derudover kunne double-værdien være 5 mia. i stedet for 2.7, og det er der ikke plads til i en int. Et tredje problem er, at man skal vælge, hvordan man vil udføre konverteringen. Skal man afrunde korrekt til 3, eller nedrunde til 2? Det første tager lidt mere tid end det sidste.

Af disse årsager bliver man i nogle tilfælde nødt til eksplicit at fortælle oversætteren, at den skal foretage en værdi-typekonvertering.

3.6.1. Eksplicit værdi-typekonvertering

Man konverterer en værdi til en anden type ved at skrive det eksplicit (eng.: cast) 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 til nærmeste hele tal.

3.6.2. Implicit værdi-typekonvertering

Implicit 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 typekonvertering 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 appendikset sidst i dette kapitel findes en tabel over typerne.

3.6.3. Misforståelser af værdi-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
```

Resultatet bliver

```
2
2.7
```

Man kunne måske fristes til at tro, at i punkt A konverteres variablen 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.

3.7. Fejl

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

3.7.1. Indholdsmæssige (logiske) fejl

Da maskinen ikke forstår programmet, kan den heller ikke rette op på fejlene i programmørens 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, der var tilsigtet:

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

Resultatet bliver

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

Dette eksempel har en stavefejl og en forkert udregning. I Afsnit 3.5.4 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.

3.7.2. Sproglige fejl

Mens computeren ikke har mulighed for at finde indholdsmæssige fejl i programmerne, kan den godt finde sproglige og syntaksmæssige problemer, dvs. hvis kildeteksten gør brug af ukendte variable 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 kildeteksten, 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 p*r*intln (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

3.7.3. 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[] args)
  {
    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 kapitlet om undtagelser.

3.8. Opgaver

3.8.1. 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

3.8.2. Kurveprogram

1. Skriv et program, der tegner grafen over kvadratrod-funktionen (Math.sqrt()).Vink: Når du vil skrive en "*" uden linjeskift kan du bruge System.out.print("*") (dvs. uden 'ln'). 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 0.2*x*x +0.5*x +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 kun 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?

3.9. 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.

3.9.1. 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.

3.9.2. De simple typer

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

Tabel 3-1. 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

Type	Art	Antal bit	Mulige værdier	Standardværdi
long	heltal	64	-9223372036854775808 til 9223372036854775807	0
float	kommatal	32	±1.40239846E-45 til ±3.40282347E+38	0.0
double	kommatal	64	±4.94065645841246544E-324 til ±1.79769313486231570E+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.

3.9.3. Værditypekonvertering

Konvertering til en anden type sker automatisk i de 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 informationstabet (glemmes dette 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.

3.9.4. Aritmetiske operatorer

Tabel 3-2. Java

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.

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

3.9.5. Regning med logiske udtryk

u1 og u2 er to logiske udtryk eller logiske variabler

Tabel 3-3. Java

<table>
<tr><th>Operator</th><th>Brug</th><th>Forklaring</th></tr>
<tr><td>&&</td><td>u1 && u2</td><td>både u1 og u2 er sandt</td></tr>
<tr><td>||</td><td>u1 || u2</td><td>u1 eller u2 er sandt</td></tr>
<tr><td>!</td><td>! u1</td><td>negation af u1</td></tr>
</table>

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

Tabel 3-4. Java

1. udtryk	2. udtryk	1. udtryk && 2. udtryk
FALSK	FALSK	FALSK
FALSK	SAND	FALSK
SAND	FALSK	FALSK
SAND	SAND	SAND

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.

Tabel 3-5. Java

<table>
<tr><th>1. udtryk</th><th>2. udtryk</th><th>1. udtryk || 2. udtryk</th></tr>
<tr><td>FALSK</td><td>FALSK</td><td>FALSK</td></tr>
<tr><td>FALSK</td><td>SAND</td><td>SAND</td></tr>
<tr><td>SAND</td><td>FALSK</td><td>SAND</td></tr>
<tr><td>SAND</td><td>SAND</td><td>SAND</td></tr>
</table>

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 <= 5)).

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

3.9.6. Sammenligningsoperatorer

Tabel 3-6. Java

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

3.9.7. 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. På den måde er det ofte lettere at se problemet, hvis programmet ikke virker.
- Gør flittigt brug af System.out.println(...) til at kontrollere, om programmet gør som forventet.
- Lær en standardiseret indrykning fra starten og følg den stringent (Se afsnit Afsnit 3.4).

Kapitel 4. Objekter

Indhold:

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

Kapitlet forudsættes i resten af bogen.

Forudsætter Kapitel 3, Basal programmering.

4.1. 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: tekststrenge, datoer, lister, filer og kataloger, 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

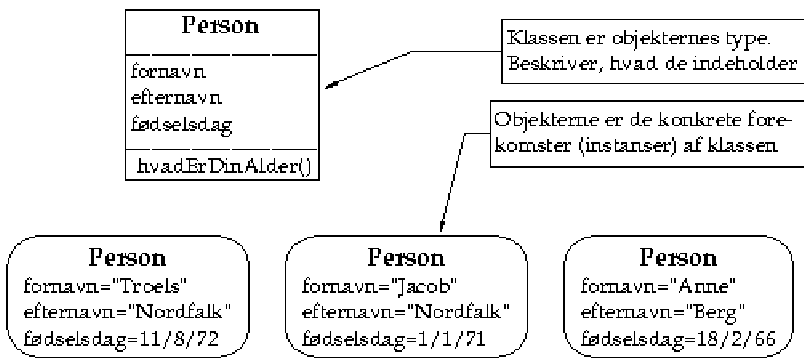
Ligesom med de simple typer afhænger det af objektets type, dets klasse, hvad man kan gøre med det. Ordet "klasse" skal forstås i betydningen "kategori, gruppe". Alle objekter kan klassificeres som værende af en bestemt type (klasse), f.eks. Streng, Dato, Fil, Knap.

Et objekts type kaldes dets klasse

Klassen bestemmer, hvilke metoder og data et objekt indeholder

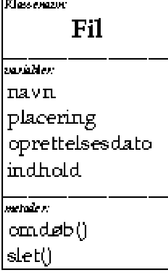
Objekter af samme klasse forstår de samme beskeder (kommandoer og spørgsmål) og indeholder samme slags data. Objekter af klassen Person indeholder f.eks. begge et navn, men navnene (data) i de to person-objekter kan være forskellige.

Figur 4-1. Java



Vi tegner klasser som vist her:

Figur 4-2. Java



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, StringBuffer og Vector.

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

4.2. Punkter (klassen Point)

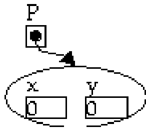
Det første objekt, vi vil arbejde med, er Javas Point-objekt, der repræsenterer et *punkt*. I Java indeholder et punkt to heltalsvariable, nemlig en x- og en y-koordinat. Vi vil senere bruge Point-klassen, når vi kommer til programmering af grafik.

4.2.1. 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;
```

Figur 4-3. Java ***p refererer til et punktobjekt.Objektet har x=0 og y=0***



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:

```
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 dets datas startværdi

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.

4.2.2. Objektvariable

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

```
int a;
```

... og gemmer p's x-koordinat i variablen:

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

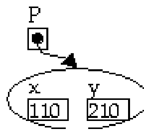
p's x-koordinat får man fat i ved at skrive p punktum x. Vi kan derefter udskrive a:

```
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);
```

Figur 4-4. Efter tildeling af p.x og 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:

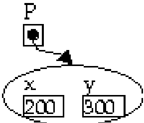
```
p.x = p.x + 5;
```

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

4.2.3. 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.

Figur 4-5. Efter kald af move(200,300)




```
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.

Et objekt kan indeholde metoder

Et metodekald på et objekt kan ændre objektets variabler

Her er Point-klassen illustreret i UML-notationen:

Figur 4-6. Java

Point

x

y

move(x,y)

translate(x,y)

Herunder er nogle af de metoder, som punktobjekter forstår (en oversigt over klassen kan findes i appendiks, Afsnit 4.7.1).

Nogle af Point-klassens metoder

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, hvordan metoden bruger parameteren.

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.

4.2.4. Eksempel

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

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

public class Punkt
{
  public static void main(String[] args)
  {
    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);
  }
}
```

Resultatet bliver:

```
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
```

4.2.5. 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 kapitlet om pakker allerede nu).

4.3. Rektangler (klassen Rectangle)

Vi vil nu gå videre til nogle lidt mere indviklede 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.

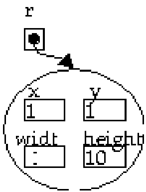
Et rektangel-objekt består af en x- og y-koordinat og en højde og bredde. Disse objektvariabler (data) hedder x,y,width og height.

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:

Figur 4-7. Java




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

Dette skaber et Rectangle-objekt med 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 vi skal bruge fire linjers programkode på at sætte et objekts værdier hver gang vi opretter det.

4.3.1. 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 skaber et nyt objekt og initialiserer objektets data

Nogle konstruktører tager parametre, der beskriver, hvordan objektet skal oprettes

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 4.7.1. Vi kan f.eks. bruge den, der tager to parametre:

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

4.3.2. 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 variablen '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*x+y*y er, og svaret, som metoden giver tilbage, gemmer vi i variablen hypotenuse.

4.3.3. Metoders returtype

Ligesom parametre skal være af den rette type, gælder det 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 r 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 4.7.2.

Nogle af Rectangle-klassens metoder.

boolean *contains* (Point p) *"indeholder du p?"*returnerer true hvis *p* er inden for rektanglet, ellers false.

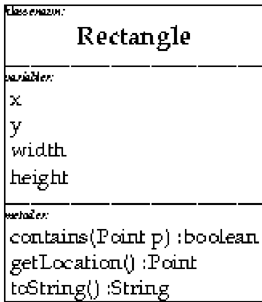
Point *getLocation*() *"hvad er din placering?"*

returnerer et Point-objekt, der har samme koordinater som rektanglets øverste venstre hjørne.

String *toString*() *"hvordan vil du beskrive dig selv?"*giver en beskrivelse af rektanglet med (x,y)-koordinater og mål som en streng.

Her er Rectangle illustreret i UML-notation.

Figur 4-8. Java



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 Rektangler
{
  public static void main (String[] args)
  {
    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
```

4.3.4. 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 mellemvariable, 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.

4.4. Tekststrenge (klassen String)

Figur 4-9. s refererer ikke til noget

Vi kommer nu til de mest brugte objekter, nemlig tekststrenge (af typen String). En variabel, der refererer til strenge 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 refererer endnu ikke til nogen konkret streng. Lad os tildele s en værdi:

Figur 4-10. s refererer nu til en streng

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

Nu er situationen som vist på figuren til højre. Vi kan bruge s i vores program, f.eks. til at skrive ud på skærmen:

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

Nu kan vi spørge streng-objektet om forskellige ting. For eksempel kan vi kalde metoden length(), der svarer til spørgsmålet "hvor lang er du?". Strengen vil svare med tallet 8:

```
...
int strengensLængde;
strengensLængde = s.length();
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 strenge. I kursiv til højre står spørgsmålene, som de svarer til.

Nogle af String-klassens metoder. En mere fuldstændig oversigt kan findes i Afsnit 4.7.3.

char *charAt* (int indeks) *"hvilket tegn er der på plads nummer x?"*Returnerer tegnet på det angivne *indeks*. Indeks tæller fra 0.

String *replace* (char gammeltTegn, char nytTegn) *"hvad hvis tegn x erstattes med y?"*Returnerer en ny streng, som er identisk med denne streng, bortset fra at alle forekomster af *gammeltTegn* er erstattet med *nytTegn*.

String *substring* (int startindeks) *"hvad er delstrengen fra x?"*Returnerer en ny streng, som er en del af denne streng. Delstrengen starter ved *startindeks* og går til slutningen.

String *substring* (int startindeks, int slutindeks) *"hvad er delstrengen fra x til y?"*Returnerer en ny streng, som er en del af denne streng. Delstrengen starter ved *startindeks* og slutter ved *slutindeks* (til og med *slutindeks*-1).

String *toLowerCase* () *"hvordan ser du ud med små bogstaver?"*Returnerer en ny streng, som er identisk med denne streng, bortset fra at alle store bogstaver er erstattet med små.

String *toUpperCase* () *"hvordan ser du ud med store bogstaver?"*Returnerer en ny streng, som er identisk med denne streng, bortset fra at alle små bogstaver er erstattet med store.

boolean *equals* (String str) *"er det samme indhold?"*Returnerer sand, hvis denne streng indeholder den samme tegnsekvens som *str*, ellers falsk.

int *length* () *"hvad er din længde?"*Returnerer længden af (antal tegn i) strengen.

int *indexOf* (String str) *"hvor er delstrengen x?"*Returnerer indekset på den første forekomst af *str* som delstreng. Hvis *str* ikke er en delstreng, returneres -1.

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

```
// Strengeleg.java
// Viser brugen af String-klassen og dens metoder.
public class Strengeleg
{
  public static void main(String[] args)
  {
    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("Tegnet på plads nummer 2 er: "+s.charAt(2));
    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
Tegnet på plads nummer 2 er: e
Det første g er på plads nummer: 4
```

4.4.1. Strenge 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. String-objekter er derimod indrettet sådan, at når de først er oprettet, så kan de ikke ændres (det giver Java mulighed for at spare hukommelse ved at slå streng-objekter med fælles indhold sammen til én streng). I stedet for at ændre indholdet af strengen returnerer String-objekters metoder altid en anden streng, som er resultatet af ændringen.

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
```

Kalder man derimod en metode på et String-objekt, bliver den ikke ændret:

```
  s.replace('d','f');      // s forandres ikke
```

replace()-metoden giver en ny streng tilbage til os, hvor alle 'd'-tegn er erstattet med 'f', men den bliver smidt væk med det samme, da vi ikke bruger returværdien. I stedet kunne vi skrive:

```
  String s2;
  s2 = s.replace('d','f');  // s forandres ikke, men s2 husker resultatet
```

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

Her ses samlet et eksempel på strenges uforanderlighed:

```
// Strengeleg2.java
public class Strengeleg2
{
  public static void main (String[] args)
  {
    String s1;
    String s2;
    String s3;
    String s4;

    s1 = "Ude godt, men hjemme bedst.";
    s2 = s1.toUpperCase();          // kald toUpperCase() på s1
    s3 = s2.replace('E', 'X');        // kald replace() på s2
    s4 = s3.substring(4, 16);        // kald substring() på s3

    System.out.println ("s1: " + s1);  // s1 er uændret af toUpperCase()-kald
```

```
    System.out.println ("s2: " + s2);  // s2 er uændret af replace()-kaldet
    System.out.println ("s3: " + s3);  // s3 er uændret af s3.substring(4, 20)
    System.out.println ("s4: " + s4);  // s4 er resultatet af substring()-kaldet
  }
}
```

```
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 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-variablerne 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.

4.4.2. 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, der altid vil være helt den samme som originalen?

4.4.3. 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 Strengeleg2.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.

4.4.4. 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 nogenlunde 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);
```

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".

4.4.5. 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 == *referencerne til* (adresserne på) objekterne, ikke *indholdet af* dem. Sammenligningen s1==s2 går godt fordi s1 og s2 refererer til samme objekt.

Derfor vil det gå galt hvis s1 og s2 refererer til to objekter forskellige steder i hukommelsen, selvom de har samme indhold:

```
s1 = "Hej verden";
s2 = "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 to objekter er ens, og den bør man bruge i stedet for ==.

Sammenligning af adresser på objekter sker med ==

Sammenligning af objekters indhold sker med equals()-metoden

4.4.6. 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 finder og fjerner alle forekomster af ordet "måske" fra en tekst.
4. 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.
5. Skriv et program, der tæller antallet af kommaer i en tekst.
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) eller med charAt(n), som er beskrevet i apendikset, og kræver at du også bruger variabler af typen char).
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".

4.5. Lister (klassen Vector)

En Vector er en liste af andre objekter, nummereret efter et indeks.

Figur 4-11. Java

Vector
addElement(objekt) insertElementAt(objekt,indeks) size() :int elementAt(indeks) :Object toString() :String

Konstruktører og metoder er beskrevet i appendiks, Afsnit 4.7.4.2.

En Vector er en liste af andre objekter

Man opretter en vektor med f.eks.:

```
Vector v;
v = new Vector();
```

Derefter kan man tilføje et objekt i enden af listen med *addElement*(objekt), f.eks.:

```
v.addElement("æh");
```

tilføjer strengen "æh" sidst i vektoren.

Man kan sætte ind midt i listen med v.*insertElementAt*(objekt, int indeks), f.eks.:

```
v.insertElementAt("øh",0);
```

indsætter "øh" på plads nummer 0, sådan at vektoren 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 henter elementerne ud igen med *elementAt* (indeks).

Med v.*size*() får man antallet af elementer i vektoren, i dette tilfælde 2.

Nogle af Vector-klassens metoder

Metoder

void *addElement*(objekt)Føjer *objekt* til vektoren. *objekt* kan være et vilkårligt objekt (men ikke en simpel type)

void *insertElementAt*(objekt, int indeks)Indsætter *objekt* i vektoren lige før plads nummer *indeks*

void *removeElementAt*(int indeks)Sletter objektet på plads nummer *indeks*

int *size*()Returnerer antallet af elementer

Object *elementAt* (int indeks)Returnerer en reference til objektet på plads nummer *indeks*. Husk at lave en typekonvertering af referencen til den rigtige klasse før resultatet lægges i en variabel (se Point-eksemplet nedenfor).

String *toString* ()Returnerer vektorens indhold som en streng. Dette sker ved at konvertere hver af elementerne til en streng.

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

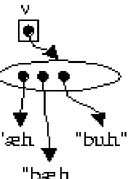
Her er et lille eksempel:

```
import java.util.*;

public class Vectortest
{
  public static void main(String args[])
  {
    Vector v;

    v = new Vector();

    v.addElement("æh");
    v.addElement("bæh");
    v.addElement("buh");

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

    v.insertElementAt("og",2);
    System.out.println("Nu har v elementerne "+v); //toString() kaldes implicit

    v.removeElementAt(0);
    System.out.println("Nu har v elementerne "+v+" og størrelsen "+v.size());

    System.out.println("På plads nummer 2 er: "+v.elementAt(2));
  }
}
```

```
v har elementerne [æh, bæh, buh]
Nu har v elementerne [æh, bæh, og, buh]
Nu har v elementerne [bæh, og, buh] og størrelsen 3
På plads nummer 2 er: buh
```

Vi indsætter først tre (referencer til) strenge i vektoren. I hukommelsen ser det sådan ud:

Figur 4-12. Java



Dernæst lægges "og" ind på plads nummer 2, dvs. efter "bæh" og før "buh". Til sidst fjernes "æh" på plads nummer 0.

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

4.5.1. Eksempel med Point

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

Læg mærke til, hvordan man kan gennemløbe en vektor:

```
for (int n=0;n<pv.size();n++)
{
  Point p;
  p=(Point) pv.elementAt(n); // punkt nr n hentes ud af vektoren
                             // (typekonvertering nødvendig)
  //...
}
```

Da vektorer kan indeholde objekter af alle mulige typer, er man nødt til at lave en typekonvertering af den objekt-reference, som elementAt() returnerer.

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

public class MindsteAfstand
{
  public static void main(String args[])
  {
    Vector pv; // punkt-vektor
    Point origo, p1, p2, p3;
    double mindist=10000;
```

```
    pv=new Vector();
    origo=new Point(0,0);
    p1=new Point(0,65);
    p2=new Point(50,50);
    p3=new Point(120,10);

    pv.addElement(p1);
    pv.addElement(p2);
    pv.addElement(p3);

    for (int n=0;n<pv.size();n++)
    {
      double dist;
      Point p;

      p=(Point) pv.elementAt(n); // punkt nr n hentes ud af vektoren
                                 // (typekonvertering nødvendig)

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

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

```
Den mindste afstand mellem punkterne [java.awt.Point[x=0,y=65], java.awt.Point[x=50,y=50],
```

Her er en lille stump programkode, der i stedet finder den mindste afstand mellem to punkter, der ligger ved siden af hinanden i vektoren:

```
    for (int n=0;n<pv.size()-1;n++)
    {
      double dist;
      Point p,q;

      p=(Point) pv.elementAt(n);
      q=(Point) pv.elementAt(n+1);

      dist = q.distance(p);
      if (dist<minDist)
      {
        mindist=dist;
      }
    }
```

Og herunder er vist, hvordan man kan finde den mindste afstand mellem to vilkårlige punkter i vektoren:

```
for (int n=0;n<pv.size();n++)
{
  for (int k=n;k<pv.size();k++)
  {
    double dist;
    Point p,q;
    p=(Point) pv.elementAt(n);
    q=(Point) pv.elementAt(k);

    dist = q.distance(p);
    if (dist<minDist)
    {
      mindist=dist;
    }
  }
}
```

4.6. Ekstra eksempler

Tillykke! Du har nu lært, hvordan man opretter og arbejder med objekter, og du er kommet igennem de vigtigste klasser af objekter. De eksempler og klasser, der følger nu er ikke fundamentalt forskellige fra det, du har set. Der bliver med andre ord ikke introduceret nogen nye begreber, men klasserne kan være nyttige, og eksemplerne kan måske hjælpe dig med at få repeteret ideerne om brug af objekter, metodekald og konstruktører.

4.6.1. Blanding af kort med Vector

I det følgende bruges nogle af de metoder der er nævnt i appendikset om Vector.

Dette program blander nogle spillekort (beskrevet som strenge) i en vektor. Det gøres ved 100 gange at tage et kort fra en tilfældig plads og flytte det til en anden tilfældig plads:

```
import java.util.*;

public class BlandKort
{
  public static void main(String args[])
  {
    Vector bunke;
    bunke = new Vector();

    // Opbyg bunken
    for (int n=2; n<9; n++)
    {
      bunke.addElement("ruder"+n);  // ruder
```

```
      bunke.addElement("klør"+n);    // klør
      bunke.addElement("spar"+n);    // spar
    }

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

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

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

      bunke.removeElementAt(nr);

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

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

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

I et rigtigt program ville de enkelte kort nok være repræsenteret med objekter fra en Kort-klasse med objektvariablerne farve og værdi.

4.6.2. Datoer (klassen Date)

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

Figur 4-13. Java

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

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

```
Date netopNu;
netopNu = new Date();
```

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 tager 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 appendikset 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[] args)
  {
    Date netopNu;
    Date jacob;
```

```
    netopNu = new Date();
    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: Sun Jul 15 14:33:59 CEST 2001
Jacob blev født Fri Jan 01 12:00:00 CET 1971
Jacob var halvt så gammel Wed Apr 09 01:46:59 CEST 1986
```

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

Man kan gøre meget mere med datoer end vist her.

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.

GregorianCalendar-klassen repræsenterer vort kalendersystem (det gregorianske / julianske) og har alle de kalenderfunktioner, man kunne ønske sig. Med den kan man arbejde med ugedage, måneder, år, tidszoner, sommertid etc.

Disse klasser vil ikke blive behandlet her, da de benytter begreber, som ikke er introduceret endnu (klassevariabler og klassemetoder).

4.6.3. 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 before-metoden til at tjekke, om du er nået frem til nu).

4.7. Appendiks

Her finder du en oversigt over de vigtigste klasser og deres vigtigste metoder. På siden http://java.sun.com/products/jdk/1.2/docs/api/index.html findes en komplet oversigt.

Nogle af metoderne er markeret med:

- JDK1.2: Denne metode er kun tilgængelig fra og med Java version 1.2.

- (frarådes): Denne metode er frarådet (eng.: deprecated), men findes endnu af hensyn til programmer skrevet til tidligere versioner af Java. Oversætteren advarer mod brugen med '... has been deprecated', men programmet kører i øvrigt fint.

4.7.1. 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.;*

Variabler

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) JDK1.2 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]

4.7.2. Rectangle

Rectangle repræsenterer et todimensionalt rektangel.

java.awt.Rectangle - todimensionalt rektangel - skal importeres med import java.awt.;*

Variabler

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.

4.7.3. String

Strenge er specielle ved, at de ikke kan ændres, når de først er oprettet.

java.lang.String - tekststrenge

char *charAt* (int indeks)returnerer tegnet på det angivne *indeks*. Indeks tæller fra 0.

String *replace* (char gammeltTegn, char nytTegn) returnerer en ny streng, som er identisk med denne streng, bortset fra at alle forekomster af *gammeltTegn* er erstattet med *nytTegn*.

String *substring* (int startindeks)Returnerer en ny streng, som er en del af denne streng. Delstrengen starter ved *startindeks* og går til slutningen.

String *substring* (int startindeks, int slutindeks)Returnerer en ny streng, som er en del af denne streng. Delstrengen starter ved *startindeks* og slutter ved *slutindeks* (til og med *slutindeks*-1).

String *toLowerCase* ()returnerer en ny streng, som er identisk med denne streng, bortset fra at alle store bogstaver er erstattet med små.

String *toUpperCase* ()returnerer en ny streng, som er identisk med denne streng, bortset fra at alle små bogstaver er erstattet med store.

String *trim* ()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.

int *length* ()returnerer længden af (antal tegn i) strengen.

int *indexOf* (String str)returnerer indekset på den første forekomst af *str* som delstreng. Hvis *str* ikke er en delstreng, returneres -1.

int *lastIndexOf* (String str)returnerer indekset på den sidste forekomst af *str* som delstreng. Hvis *str* ikke er en delstreng, returneres -1.

boolean *startsWith* (String str)returnerer sand, hvis denne streng starter med de samme tegn som *str*, ellers falsk.

boolean *endsWith* (String str)returnerer sand, hvis denne streng slutter med de samme tegn som *str*, ellers falsk.

boolean *equals* (String str)returnerer sand, hvis denne streng har samme indhold som *str*, ellers falsk.

boolean *equalsIgnoreCase* (String str)returnerer sand, hvis denne streng har samme indhold som *str*, 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).

4.7.4. Specialtegn i strenge

Visse tegn kan man ikke skrive direkte i tekststrenge i kildeteksten. De står opført herunder:

Tabel 4-1. Java

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)	\\
Unikode-tegn nummer XXXX	\uXXXX

Unikode-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". Unikode-tegntabellen kan findes på http://unicode.org (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
```

4.7.4.1. Date

Date repræsenterer en tid, dvs. en dato og et klokkeslæt.

java.util.Date - et tidspunkt - skal importeres med import java.util.;*

Konstruktører

Date ()opretter et Date-objekt, som repræsenterer tidspunktet, da det blev oprettet, målt i millisekunder.

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* på dette objekt. Sættes den til en dag uden for denne måned, ændres måneden tilsvarende. ... tilsvarende med *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 kalendre end det gregorianske kalendersystem der bruges i den vestlige verden. Det betyder dog ikke det store for europæiske programmer.

4.7.4.2. Vector

Vector er en liste af andre objekter.

java.util.Vector - en liste af objekter - skal importeres med import java.util.;*

Konstruktører

Vector ()opretter en tom Vector.

Metoder

void *addElement*(objekt)føjer *objekt* til vektoren. *objekt* kan være et vilkårligt objekt (men ikke en simpel type).

void *insertElementAt*(objekt, int indeks)indsætter *objekt* i vektoren lige før plads nummer *indeks.*

void *removeElementAt*(int indeks)sletter objektet på plads nummer *indeks.*

boolean *isEmpty*()returnerer sand, hvis vektoren er tom (indeholder 0 elementer).

int *size*()returnerer antallet af elementer.

Object *elementAt* (int indeks)returnerer en reference til objektet på plads nummer *indeks*. Husk at lave en typekonvertering af referencen til den rigtige klasse, før du bruger resultatet.

boolean *contains*(objekt)returnerer sand, hvis *objekt* findes i vektoren.

int *indexOf*(objekt)returnerer indekset på første forekomst af *objekt* i vektoren. Hvis den ikke findes, returneres -1.

Enumeration *elements*()giver en opremsning (eng.: enumeration) af elementerne.

String *toString* ()returnerer vektorens indhold som en streng. Dette sker ved at konvertere hver af elementerne til en streng.

Kapitel 5. Definition af klasser

Kapitlet forudsættes af resten af bogen.

Forudsætter Kapitel 4, 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

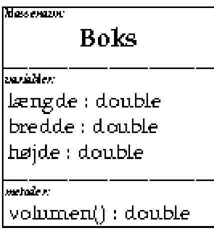
Et objekt er en konkret forekomst (instans) af klassen

Når man programmerer objektorienteret, samler man data i selvstændige objekter og definerer metoder, som arbejder på disse data i objekterne.

5.1. En Boks-klasse

Lad os tage et eksempel på en klassedefinition:

Figur 5-1. Java



Vi definerer klassen Boks, som indeholder tre variabler, nemlig bredde, højde og længde. Derudover definerer vi metoden volumen(), som arbejder på disse data. Metoden returnerer en double og tager ingen parametre.

```
public class Boks
```

```
{
  double længde;
  double bredde;
  double højde;

  double volumen()
  {
    double vol;
    vol = længde*bredde*højde;
    return vol;
  }
}
```

5.1.1. Variabler

Variablerne bredde, højde og længde kaldes også objektvariabler, fordi hvert Boks-objekt har en af hver.

Objektvariabler erklæres direkte i klassen uden for metoderne

Vi kan lave et Boks-objekt, boksobjekt med new:

```
    Boks boksobjekt;
    boksobjekt = new Boks();
```

Nu er der oprettet et Boks-objekt i lageret, der således har en højde-, en bredde- og en længde-variabel.

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 variabler eksisterer kun, så længe metoden, hvori de er erklæret, udføres

Modsat højde, længde og bredde begynder vol-variabler altså ikke at eksistere, bare fordi vi har skabt en Boks.

5.1.2. Brug af klassen

Objekter af klassen Boks kan f.eks. benyttes på følgende måde:

```
public class BenytBoks
{
  public static void main(String args[])
  {
    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.

I følgende sætninger (i klassen BenytBoks) sættes det nyoprettede Boks-objekts variabler:

```
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 variablerne, som er blevet tilført data i linjerne ovenfor. Metoden returnerer en double - denne lægges over i rumfang-variablen, som udskrives.

5.1.3. 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 den eventuelt tager:

```
double volumen()
```

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 resultatet af vol bliver givet tilbage (returneret) til der, hvor 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 metoden volumen() derefter kaldes på dette objekt, 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

5.1.4. 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 args[])
  {
    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.

5.2. Indkapsling

Indkapsling af data og metoder i objekter betyder, at man ikke lader andre bruge objekterne helt efter eget forgodtbefindende. 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) variabler?

Indkapsling i klasser er vigtig, når programmerne bliver store og komplekse. Hvis det er muligt at ændre data i en klasse, se eksemplet ovenfor, kan det føre til situationer, som kommer ud af kontrol i store komplekse systemer.

Ved at indkapsle data er den eneste måde at ændre data på brugen af metoder. I metoderne kan man sikre sig mod vanvittige overgreb på variabler ved at tilføre logik, der sikrer, at variablerne 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 10001 pakker på hver *minus* en kubikmeter og 10000 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 variabler har fornuftige værdier.

Man styrer "synligheden" af en variabel eller metode med nøgleordene public og private:

public betyder "synlig for alle"

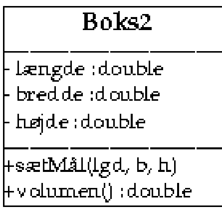
private betyder "kun synlig i klassen"

Herunder ses en modificeret version af eksemplet med Boks- og BenytBoks-klassen, men nu er variablerne 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)
    {
       System.out.println("Ugyldige mål. Bruger standardmål.");
      længde = 10.0;
      bredde = 10.0;
      højde  = 10.0;
    } else {
      længde = lgd;
      bredde = b;
      højde  = h;
    }
  }

  public double volumen()
  {
    double vol;
    vol = længde*bredde*højde;
    return vol;
  }
}
```

Klassen er illustreret med UML nedenfor. Bemærk, at variablerne er private, så de har et - foran, mens metoderne, som kan ses udefra (public), har et + foran:

Figur 5-2. Java



Nu da variablerne bredde, højde og længde er erklæret private, er det ulovligt at ændre dem "udefra", i vores BenytBoks-program.

Til gengæld har vi defineret metoden sætMål(), som man kan kalde for at sætte målene. Nu da den eneste måde at ændre data på er ved specifikt at kalde metoden sætMål(), kan vi indlægge en ønsket logik - for eksempel sikre os mod 0 (nul) eller negative værdier.

```
public class BenytBoks2
{
  public static void main(String args[])
  {
    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 repræsentationen. Man kunne f.eks. gemme volumen i Boks-klassen i stedet for højden og så lade højden være beregnet.

5.3. 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 med nogle konstruktører:

Figur 5-3. Java

Boks3

- længde : double
- bredde : double
- højde : double

+Boks3()
+Boks3(lgd, b, h)
+sætMål(lgd, b, h)
+volumen() : double

```
public class Boks3
{
  private double længde;
  private double bredde;
  private double højde;

  public Boks3()
  {
    System.out.println("Standardboks oprettes");
    sætMål(10, 10, 10);
  }

  // En anden konstruktør der tager 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);
  }

  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;
    }
  }

  public double volumen()
  {
    return længde*bredde*højde;
  }
```

```
}
```

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)
```

Vi prøver Boks3 med:

```
public class BenytBoks3
{
  public static void main(String args[])
  {
    Boks3 enBoks;
    // brug konstruktøren uden parametre
    enBoks = new Boks3();

    System.out.println("Volumen er: "+ enBoks.volumen());

    Boks3 enAndenBoks;
    // brug den anden konstruktør
    enAndenBoks = new Boks3(5,5,10);

    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
```

5.3.1. 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 et objekt med 'new'.

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:7: No constructor matching Boks3() found in class Boks3.
                enBoks = new Boks3();
```

5.3.2. Opgaver

1. Definér klassen Pyramide. Objekterne skal have variablerne *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.
2. Ret Boks3 til også at have variablen massefylde, og definér en ekstra konstruktør der også tager massefylden (den oprindelige konstruktør med lgd, b og h kan sætte massefylden til 1). Lav også metoder til at sætte massefylden, sætMassefylde(double m), og udregne vægten, vægt(). Test din klasse med en ændret udgave af BenytBoks3.

5.4. 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.

Figur 5-4. Java

Terning
værdi :int
+Terning() +kast() +toString() :String

```
// En klasse der beskriver 6-sidede terninger
public class Terning
```

```
{
  // den side der vender opad lige nu
  int værdi;

  // konstruktør
  public Terning()
  {
    kast(); // kald kast() der sætter værdi til noget fornuftigt
  }

  // metode til at kaste terningen
  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 args[])
  {
    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.
```

5.4.1. 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 Vector-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.

5.5. Relationer mellem objekter

Indtil nu har alle vore objekter haft simple typer som variabler. Nu vil vi se på objekter der har andre objekter som variabler (dvs. de har referencer til andre objekter).

5.5.1. 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, der 'har' terningerne, og som er bekvem at bruge fra hovedprogrammet. Med 'har' menes, at referencerne til Terning-objekterne kendes af raflebægeret, men 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
{
  public Vector terninger;  // Raflebaeger har en vektor af terninger

  public Raflebaeger(int antalTerninger)
  {
    terninger = new Vector();
    for (int i=0;i<antalTerninger;i++)
    {
      Terning t;
      t = new Terning();
      tilføj(t);
    }
  }

  public void tilføj(Terning t)       // Læg en terning i bægeret
  {
    terninger.addElement(t);
  }

  public void ryst()                  // Kast alle terningerne
  {
    for (int i=0;i<terninger.size();i++)
    {
      Terning t;
      t = (Terning) terninger.elementAt(i);
      t.kast();
    }
  }

  public int sum()                    // Summen af alle terningers værdier
  {
    int resultat;
    resultat=0;
    for (int i=0;i<terninger.size();i++)
    {
      Terning t;
      t = (Terning) terninger.elementAt(i);
      resultat = resultat + t.værdi;
    }
    return resultat;
  }

  public int antalDerViser(int værdi) // Antal terninger med en bestemt værdi
  {
    int resultat;
    resultat = 0;
    for (int i=0;i<terninger.size();i++)
    {
      Terning t;
      t = (Terning) terninger.elementAt(i);
      if (t.værdi==værdi)
```

```
      {
        resultat = resultat + 1;
      }
    }
    return resultat;
  }

  public String toString ()           // Beskriv bægerets indhold
  {// (vektorens 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 ToSeksere
{
  public static void main(String[] args)
  {
    Raflebaeger bæger;
    boolean toSeksere;
    int antalForsøg;

    bæger = new Raflebaeger(3);   // opret et bæger med 3 terninger
    toseksere=false;
    antalForsøg = 0;
    while (toseksere==false)
    {
      bæger.ryst();              // kast alle terningerne
      System.out.print("Bæger: " + bæger + " sum: " + bæger.sum());
      System.out.println(" Antal 6'ere: "+bæger.antalDerViser(6)
                       + " antal 5'ere: "+bæger.antalDerViser(5));
      if (bæger.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:

```
bæger = new Raflebaeger(3);
```

opretter et raflebæger med tre terninger i.

5.5.2. 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, ved at kalde en fireEns()-metode:

```
public boolean fireEns()
{
  ...
}
```

Lav tilsvarende de andre metoder.

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
...
```

5.6. Nøgleordet this

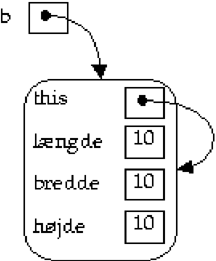
Nogle gange kan et objekt have brug for at referere til sig selv. Det gøres med *this*-nøgleordet, der ligner (og bruges som) en variabel.

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 føjTilVektor(Vector v)
  {
    v.addElement(this);
  }
}
```

Figur 5-5. *this* virker som en variabel der refererer til objektet selv



I sætMål() er der nu to sæt variabler med samme navn. Da vælger Java 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 vektor med:

```
Vector v = new Vector()
Boks2medThis b = new Boks2medThis();
v.addElement(b);
```

Med metoden føjTilVektor() kan vi i stedet for bede b om at tilføje sig selv:

```
b.føjTilVektor(v);
```

Vi vil senere (i Spiller-klassen i matadorspillet i kapitlet om nedarvning) se et eksempel på dette, hvor det er en fordel i praksis.

5.7. Ekstra eksempler

Dette afsnit giver nogle ekstra eksempler, der repeterer stoffet i kapitlet.

5.7.1. 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.

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.

Figur 5-6. Java

NSidetTerning
- sider :int - værdi :int
+NSidetTerning() +NSidetTerning(antalSider :int) +kast() +hentVærdi() :int +sætVærdi(:int) +toString() :String

```
// En n-sidet terning.
public class NSidetTerning
{
  // hvor mange sider har terningen (normalt 6)
  private int sider;

  // den side der vender opad lige nu
  private int værdi;

  // konstruktør opretter normal terning med 6 sider
  public NSidetTerning ()
  {
    sider = 6;
    kast(); // sæt værdi til noget
  }

  // opretter terning med et vist antal sider
  public NSidetTerning (int antalSider)
  {
    if (antalSider >= 3) sider = antalSider;
    else sider = 6;
    kast();
  }

  // metode der kaster terningen
  public void kast ()
  {
    // find en tilfældig side
    double tilfældigtTal = Math.random();
    værdi = (int) (tilfældigtTal * sider + 1);
  }

  // giver antallet af øjne på siden der vender opad
  public int hentVærdi ()
  {
    return værdi;
  }
```

```
  // sætter antallet af øjne der vender opad
  public void sætVærdi (int nyVærdi)
  {
    if (nyVærdi > 0 && nyVærdi <= sider) værdi = nyVærdi;
    else System.out.println("Ugyldig værdi");
  }

  // giver en strengrepræsentation af terningen
  // 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;
  }
}
```

Her er et program til at afprøve klassen med:

```
public class BenytNSidetTerning
{
  public static void main(String args[])
  {
    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)
```

5.7.2. 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!")

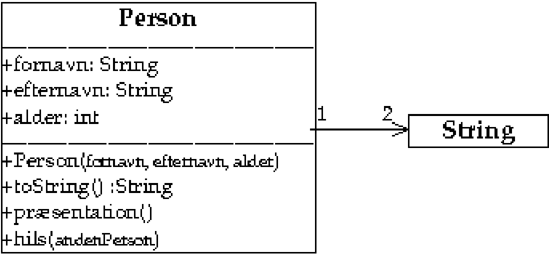
Til sidst kunne man forestille sig, at en person kan hilse på en anden person (metoden hils()). Det afhænger af alderen hvordan man hilser. En person på over 60 år vil hilse på Jacob med "Goddag hr Nordfalk", mens en yngre bare vil sige "Hej Jacob".

```
import java.util.*;
public class Person
{
  public String fornavn;
  public String efternavn;
  public int alder;
  public Vector konti; // bruges senere

  public Person(String fornavnP, String efternavnP, int alderP)
  {
    fornavn = fornavnP;
    efternavn = efternavnP;
    alder = alderP;
```

```
    konti = new Vector(); // 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.efternavn+". ");

    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-en*-relation mellem disse to klasser.

Figur 5-7. 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 args[])
  {
    Person x, y, z;
```

```
    x = new Person("Jacob", "Nordfalk", 30);
    y = new Person("Kai", "Lund", 86);
    z = new Person("Peter", "Holm", 2);

    System.out.println("Vi har oprettet "+x+", "+y+" og "+z);
    x.præsentation();
    y.præsentation();
    z.præsentation();
    x.hils(y);
    y.hils(x);
    z.hils(x);
  }
}
```

```
Vi har oprettet Jacob Nordfalk (30 år), Kai Lund (86 år) og Peter 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.

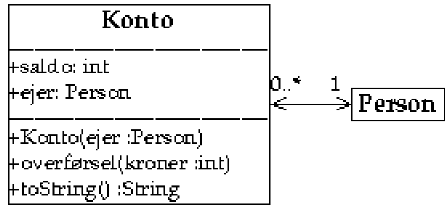
5.7.3. Bankkonti

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-en*-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)
  {
    saldo = 0;
    ejer = ejeren;                // Sæt kontoen til at referere til personen
    ejer.konti.addElement(this); // ..og personen til at referere til kontoen
  }
```

```
  public void overførsel(int kroner)
  {
    saldo = saldo + kroner;
  }

  public String toString()
  {
    return ejer+" har "+saldo+" kroner";
  }
}
```

Figur 5-8. Java



En Konto har altid en Person tilknyttet

Dette er et eksempel på en *har-en*-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.addElement(this);
```

5.8. Opgaver

Husk at lave små main()-programmer, der afprøver de ting, du programmerer.

1. Udbyg Person-klassen med metoden formue(), der skal returnere summen af saldiene på personens konti. Lav en BenytKonto med flere personer, der har flere konti.
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, og metoden udskrivPosteringer(), der skal udskrive posteringerne og løbende saldo på skærmen.
3. Lav en klasse, der repræsenterer en bil. En bil har en farve, et antal kørte kilometer og en (vurderet) pris.

4. Udbyg Bil-klassen med en vektor, 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 en person sætter sig ind i bilen.public String hvemSidderIBilen() // giver en streng, der beskriver personerne i bilen.public void alleStigerUd() // kaldes, når alle stiger ud af bilen.
5. Udbyg Person-klassen, 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).

5.8.1. 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;
  {
}
```

1. Der er 3 fejl i koden nedenfor. Find dem, og ret dem.

```
public class Fejlfinding2
{
  private String b;
  b = "Hej";

  public Fejlfinding2() {
    return b;
  }

  public Fejlfinding2(String c) {
    b = c
  }
 }
}
```

1. Der er 9 fejl i koden nedenfor. Find dem, og ret dem.

```
import java.util.*;

public class Fejlfinding3
{
  public String v = 'hej';
  public string etLangtVariabelnavn;
  public vector v2;
```

```
  public INT v3;
  public flot v4;
  public v5;
  public int vi = 5.3;
  public Vector vi3 = "xxx";
  public String vi5 = new String(Hej);
}
```

1. Der er 8 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(int y)
  {
    b = y;
  }

  public void x4(int y)
  {
    return 5;
  }
```

1. Der er 7 fejl i koden nedenfor. Find dem, og ret dem.

```
public class Fejlfinding5
{
  private int a = 5
  public void x1(int y)   {
    a = y;                           }
}
```

```
  public 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;
  }
}
```

1. 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();
  }
}
```

Kapitel 6. Nedarvning

Kapitlet forudsættes af resten af bogen.

Forudsætter Kapitel 5, 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).

6.1. 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 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.

6.1.1. 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. Sådan 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 til en ny klasse, FalskTerning, ved at starte erklæringen med:

```
public class FalskTerning extends Terning
```

Vi har automatisk overtaget (arvet) alle metoder og variabler fra Terning-klassen fordi vi skriver "extends Terning". 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.print("[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 klassediagrammet er underklassen vist med en hul pil fra FalskTerning1 til Terning.

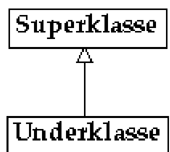
Dette kaldes også en *er-en*-relation; FalskTerning1 *er en* Terning, da den jo har alle de egenskaber en terning har.

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 og underklasse. Her er et udpluk:

Figur 6-1. Java



Superklasse kaldes også: Baseklasse, basisklasse, forældreklasse, stamklasse.

Underklasse kaldes også: Afledt klasse, nedarvet klasse, barn, subklasse.

I vores eksempel er superklassen Terning og underklassen FalskTerning1.

I det følgende program kastes med to terninger, en rigtig og en falsk:

```
public class Snydespil1
{
  public static void main(String[] args)
  {
    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!");
    }
  }
}
```

Resultatet bliver:

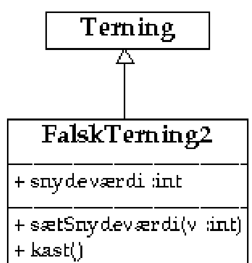
```
[kast() på FalskTerning1] t1: 1
t2: 3
[kast() på FalskTerning1] t1=1  t2=5
[kast() på FalskTerning1] t1=1  t2=3
[kast() på FalskTerning1] t1=4  t2=3
[kast() på FalskTerning1] t1=6  t2=6
To ens!
[kast() på FalskTerning1] t1=2  t2=6
```

Vi kan altså bruge FalskTerning1-objekter på præcis samme måde som Terning-objekter.

6.1.2. At udbygge med flere metoder og variabler

Lad os nu se på et eksempel, hvor vi definerer nogle variabler og metoder i nedarvingen.

Figur 6-2. Java




```
public class FalskTerning2 extends Terning
{
  public int snydeværdi;

  public void sætSnydeværdi(int nySnydeværdi)
  {
    snydeværdi = nySnydeværdi;
  }

  public void kast()
  {
    System.out.print("[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(), der sætter snydeværdi til noget andet.

```
public class Snydespil2
{
  public static void main(String[] args)
  {
    FalskTerning2 t1 = new FalskTerning2();
    t1.sætSnydeværdi(4);

    for (int i=0; i<5; i++)
    {
      t1.kast();
      System.out.println("t1=" + t1);
    }
  }
}
```

Resultatet bliver:

```
[kast() på FalskTerning2] [kast() på FalskTerning2] t1=4
[kast() på FalskTerning2] t1=4
[kast() på FalskTerning2] t1=6
[kast() på FalskTerning2] t1=6
[kast() på FalskTerning2] t1=4
```

6.1.3. Nøgleordet super

Nogen gange ønsker man i en nedarvet klasse at få adgang til superklassens metoder, selvom 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.

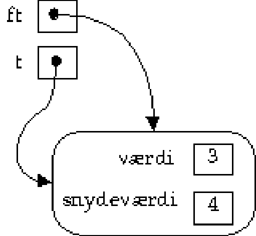
Med *super* refererer man til de metoder, der er kendt for superklassen. Dermed kan vi skrive en smartere udgave af FalskTerning:

```
public class FalskTerning3 extends Terning
{
  // tilsidesæt kast med en "bedre" udgave
  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.

6.2. Polymorfe variabler

Figur 6-3. Snydespil2MedPolymorfiefter punkt A



Se på følgende eksempel:

```
public class Snydespil2medPolymorfi
{
  public static void main(String[] args)
  {
    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);
    }
  }
}
```

Resultatet bliver:

```
[kast() på FalskTerning2] [kast() på FalskTerning2] t=4
[kast() på FalskTerning2] t=6
[kast() på FalskTerning2] 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) .

Jamen, hvad så med typesikkerheden ?

6.2.1. 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, hvor 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 variablens type

t-variablen har ikke ændret type (det kan variabler ikke), men den peger nu på et objekt af typen FalskTerning2. Men 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. Variablen bruger den bare ikke.

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 t1 refererer til et FalskTerning2-objekt, kan man kun bruge t-variablen til at kalde metoder eller anvende 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
```

6.2.2. Polymorfi

En anden meget væsentlig detalje omkring denne dispensation er, at det er *objektets* type, ikke variablens, 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 variablen t's type (så ville den jo udføre Ternings kast()).

Variablens type bestemmer, hvilke metoder man kan kalde på objektet, og hvilke objektvariabler man kan læse og ændre

Objektets type bestemmer, hvilken metode-definition der bliver udført

Af samme grund kaldes det at definere en metode, som allerede findes, fordi den er arvet, for *tilsidesættelse* (eng.: override) af metoden. Man *tilsidesætter* metodens opførsel med en anden opførsel.

For at tilsidesætte en metode skal man i underklassen lave en eksakt kopi af metode-hovedet fra superklassen

6.2.3. Et eksempel på polymorfi: Brug af Raflebaeger

```
public class SnydeMedBaeger
{
  public static void main(String[] args)
  {
    Raflebaeger bæger = new Raflebaeger(0);

    FalskTerning2 ft = new FalskTerning2();
    ft.sætSnydeværdi(6);

    bæger.tilføj(ft);   // tilføj() tager et objekt af typen Terning,
                        // og dermed også af typen FalskTerning2.

    Terning t = new Terning();
    bæger.tilføj(t);

    ft = new FalskTerning2();
    ft.snydeværdi=6;
    t=ft;               // t bruges som mellemvariabel for sjov.
    bæger.tilføj(t);

    for (int i=1; i<10; i++)
    {
      bæger.ryst();
    }
  }
}
```

I SnydeMedBaeger kaldes Raflebaeger's ryst()-metode. Hvis du nu kigger i definitionen af dennes ryst()-metode (se afsnit Afsnit 5.5), kan du se, at den kalder kast()-metoden på de enkelte objekter i "terninger"-vektoren:

```
  public void ryst()
  {
    int i;
    for (i=0;i<terninger.size();i++)
    {
      Terning t;
      t=(Terning) terninger.elementAt(i);
      t.kast();
    }
  }
```

Da to af objekterne, vi har lagt ind i bægeret, 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 at den kan bruges sammen med hans egen nye slags terning.

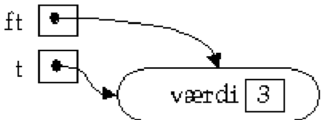
6.2.4. Hvilken vej er en variabel polymorf ?

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?

Figur 6-4. Efter punkt A(programmet vil ikke oversætte)



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:

```
t = new Terning();
ft = t;                 // sprogfejl
                        // punkt A
ft.snydeværdi = 2;
```

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.

Bemærk, at her, som i andre sammenhænge, kigger Java kun på en linje af gangen. F.eks. giver nedenstående stadig en sprogfejl, selvom det i princippet kunne lade sig gøre:

```
t = new FalskTerning2();
ft = t;                  // sprogfejl
ft.snydeværdi = 2;
```

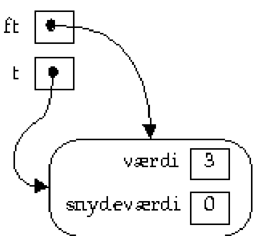
Her refererer ft i sidste linje til et rigtigt FalskTerning2-objekt, og den sidste linje ville derfor give mening, men programmet kan ikke oversættes, fordi typesikkerhedsreglen med dispensation ikke er opfyldt i linje 2.

6.2.5. 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 typeregel forhindrer oversættelse. Her kan vi bruge explicit reference-typekonvertering:

Figur 6-5. Efter punkt A




```
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.

Når vi læser objekter i en vektor, er det faktisk det, der sker.

I Raflebaeger's ryst()-metode skrev vi:

```
Terning t;
t = (Terning) terninger.elementAt(i);
t.kast();
```

I en vektor kan gemmes alle typer objekter. For at kunne lægge noget fra en vektor ned i en Terning-variabel er det derfor nødvendigt at lave en reference-typekonvertering til Terning. Dette går fint, så længe man har stoppet Terning- eller FalskTerning2-objekter i vektoren, men man kan jo putte hvad som helst i en vektor...

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.

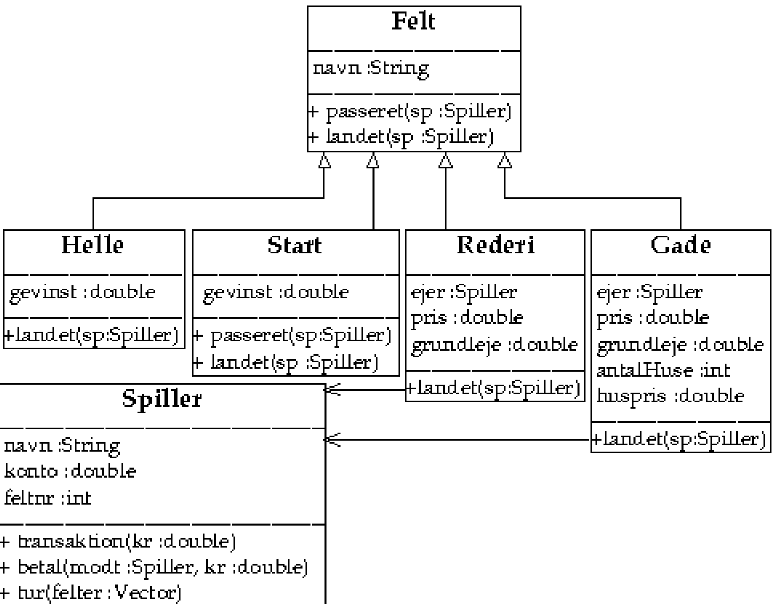
```
Terning t;
t = new Terning();
Point p;
p= (Point) t;         // Sprogfejl: Point og Terning er urelaterede
```

6.3. 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.

Figur 6-6. Java



Øverst har vi klassen Felt, som indeholder fællestrækkene for alle matadorspillets felter. F.eks. skal alle felter 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. Alle felter har også et navn, f.eks "Hvidovrevej".

```
// Superklassen for alle matadorspillets felter

public class Felt
{
  String navn;

  public void passeret(Spiller sp) // kaldes når en spiller passerer dette felt
  {
    System.out.println(sp.navn+" passerer "+navn);
  }

  public void landet(Spiller sp)   // kaldes når en spiller lander på dette felt
  {
  }
}
```

Læg mærke til, at der er forskel mellem sp.navn (spillerens navn) og navn (Felt-objektets navn).

Under Felt har vi klasserne Helle, Start, Rederi og Gade, der indeholder data og programkode, der 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.

Klassen Helle er simpel; den skal lægge 15000 kr. til spillerens kassebeholdning, hvis spilleren lander på feltet. Dette gøres ved at tilsidesætte den nedarvede passeret()-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
  {
    System.out.println(sp.navn+" lander på helle, og får overført "+gevinst);
    sp.transaktion(gevinst);        // opdater spillers konto
    System.out.println(sp.navn+"s konto lyder nu på "+sp.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

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
  {
    System.out.println(sp.navn+" passerer start og modtager "+gevinst);
    sp.transaktion(gevinst);                      // kredit/debit af konto
```

```
    System.out.println(sp.navn+"s konto lyder nu på "+sp.konto);
  }

  public void landet(Spiller sp)                   // tilsidesæt metode i Felt
  {
    System.out.println(sp.navn+" lander på start og modtager "+gevinst);
    sp.transaktion(gevinst);
    System.out.println(sp.navn+"s konto lyder nu på "+sp.konto);
  }
}
```

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-en*-relation til klassen Spiller), en pris og en leje for at lande på grunden.

```
// Rederier

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)
  {
    System.out.println(sp.navn+" er landet på "+navn);
    if (sp==ejer)
    {                                       // spiller ejer selv grunden
      System.out.println("Dette er "+sp.navn+"s egen grund");
    }
    else if (ejer==null)
    {                                       // ingen ejer grunden, så køb den
      if (sp.konto > pris)
      {
        System.out.println(sp.navn+" køber "+navn+" for "+pris);
        ejer=sp;
        sp.transaktion( -pris );
      }
      else System.out.println(sp.navn+" har ikke penge nok til at købe "+navn);
    }
    else
    {                                       // feltet ejes af anden spiller
      System.out.println("Husleje: "+grundleje);
      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 bebygges

public class Gade extends Felt
{
  Spiller ejer;
  double pris;
  double grundleje;
  int antalHuse;
  double huspris;

  public Gade(String navn, double pris, double leje, double huspris)
  {
    this.navn=navn;
    this.pris=pris;
    this.grundleje=leje;
    this.huspris=huspris;
    antalHuse = 0;
  }

  public void landet(Spiller sp)
  {
    System.out.println(sp.navn+" er landet på "+navn);

    if (sp==ejer)
    {                                       // eget felt
      System.out.println("Dette er "+sp.navn+"s egen grund");
      if (antalHuse<5 && sp.konto>huspris)
      {                                     // byg et hus
        System.out.println(ejer.navn+" bygger et hus på "+navn+" for "+huspris);
        ejer.transaktion( -huspris );
        antalHuse = antalHuse + 1;
      }
    }
    else if (ejer==null)
    {                                       // ingen ejer grunden, så køb den
```

```
        if (sp.konto > pris)
        {
          System.out.println(sp.navn+" køber "+navn+" for "+pris);
          ejer=sp;
          sp.transaktion( -pris );
        }
        else System.out.println(sp.navn+" har ikke penge nok til at købe "+navn);
      }
      else
      {                                       // felt ejes af anden spiller
        double leje = grundleje + antalHuse * huspris;
        System.out.println("Husleje: "+leje);
        sp.betal(ejer, leje);                 // spiller betaler til ejeren
      }
    }
  }
```

Et spil kunne opbygges ved at lægge forskellige felter ind i en vektor for at få et bræt:

```
// Matadorspil for to spillere
import java.util.*;

public class SpilMatador
{
  public static void main(String[] args)
  {
    Spiller sp1=new Spiller("Søren",50000);   // opret spiller 1
    Spiller sp2=new Spiller("Gitte",50000);   // opret spiller 2

    Vector felter=new Vector();               // indeholder alle felter
    felter.addElement(new Start(5000));
    felter.addElement(new Gade("Gade 1",10000, 400,1000));
    felter.addElement(new Gade("Gade 2",10000, 400,1000));
    felter.addElement(new Gade("Gade 3",12000, 500,1200));
    felter.addElement(new Rederi("Maersk",17000,4200));
    felter.addElement(new Gade("Gade 5",15000, 700,1500));
    felter.addElement(new Helle(15000));
    felter.addElement(new Gade("Gade 7",20000,1100,2000));
    felter.addElement(new Gade("Gade 8",20000,1100,2000));
    felter.addElement(new Gade("Gade 9",30000,1500,2200));

    // løb igennem 20 runder
    for (int runde = 0; runde<20; runde=runde+1)
    {
      sp1.tur(felter);
      sp2.tur(felter);
    }
  }
}
```

Man kan så lave en simpel tur()-metode, der rykker en spiller rundt på felterne ved at hente objekterne i vektoren, reference-typekonvertere dem til Felt og kalde objekternes passeret()-metode og landet()-metoden på det sidste objekt.

Denne tur()-metode placerer vi i klassen Spiller sammen med oplysningerne om spilleren.

```
// Definition af en spiller

import java.util.*;

public class Spiller
{
  String navn;
  double konto;
  int feltnr;

  public Spiller(String navn, double konto)
  {
    this.navn=navn;
    this.konto=konto;
    feltnr = 0;
  }

  public void transaktion(double kr)
  {
    konto = konto + kr;
  }

  public void betal(Spiller modtager, double kr)
  {
    System.out.println(navn+" betaler "+modtager.navn+": "+kr+" kr.");
    modtager.transaktion(kr);
    transaktion(-kr);
  }

  public void tur(Vector felter)
  {
    int slag=(int)(Math.random()*6)+1;                // terningkast
    System.out.println("***** "+navn+" på felt "+feltnr+" slår "+slag+" *****");

    // nu rykkes der
    for (int i=1;i<=slag;i=i+1)
    {
      // gå til næste felt: tæl op, hvis vi når over antal felter så tæl fra 0
      feltnr = (feltnr + 1) % felter.size();
      Felt felt;
      felt = (Felt) felter.elementAt(feltnr);
      if (i<slag) felt.passeret(this);  // kald passer() på felter vi passerer
      else felt.landet(this);           // kald land() på sidste felt
    }
    try {Thread.sleep(3000);} catch (Exception e) {}  // vent 3 sek.
  }
```

```
}
```

Fidusen er, at denne tur()-metode kan skrives uafhængigt af, hvilke felt-typer der findes: tur()-metoden kalder automatisk de rigtige landet()- og passeret()-metoder, selvom den kun kender Felt-klassen.

Bemærk i øvrigt, hvordan vi med *this* overfører en reference til spilleren selv når vi kalder passeret() og landet() på Felt-objekterne.

Linjen

```
    try {Thread.sleep(3000);} catch (Exception e) {}
```

får programmet til at holde en pause i tre sekunder inden det går videre (try og catch vil blive forklaret i kapitlet om undtagelser).

Bemærk også hvordan vi sørger for, at variablen feltnr forbliver at have en værdi mellem 0 og antallet af felter med operatoren %, der giver resten af en division (se Kapitel 3).

Her ses uddata af en kørsel af programmet:

```
***** Søren på felt 0 slår 3 *****
Søren passerer Gade 1
Søren passerer Gade 2
Søren er landet på Gade 3
Søren køber Gade 3 for 12000.0
***** Gitte på felt 0 slår 5 *****
Gitte passerer Gade 1
Gitte passerer Gade 2
Gitte passerer Gade 3
Gitte passerer Maersk
Gitte er landet på Gade 5
Gitte køber Gade 5 for 15000.0
***** Søren på felt 3 slår 2 *****
Søren passerer Maersk
Søren er landet på Gade 5
Husleje: 700.0
Søren betaler Gitte: 700.0 kr.
***** Gitte på felt 5 slår 4 *****
Gitte passerer Helle
Gitte passerer Gade 7
Gitte passerer Gade 8
Gitte er landet på Gade 9
Gitte køber Gade 9 for 30000.0
***** Søren på felt 5 slår 1 *****
Søren er landet på helle, og får overført 15000.0
Sørens konto lyder nu på 52300.0
***** Gitte på felt 9 slår 5 *****
Gitte har passeret start og modtager 5000.0
```

```
Gittes konto lyder nu på 10700.0
Gitte passerer Gade 1
Gitte passerer Gade 2
Gitte passerer Gade 3
Gitte er landet på Maersk
Gitte har ikke penge nok til at købe Maersk
***** Søren på felt 6 slår 1 *****
Søren er landet på Gade 7
Søren køber Gade 7 for 20000.0
***** Gitte på felt 4 slår 1 *****
Gitte bygger et hus på Gade 5 for 1500.0
Gitte er landet på Gade 5
Dette er Gittes egen grund
***** Søren på felt 7 slår 1 *****
Søren er landet på Gade 8
Søren køber Gade 8 for 20000.0
***** Gitte på felt 5 slår 4 *****
Gitte passerer Helle
Gitte passerer Gade 7
Gitte passerer Gade 8
Gitte bygger et hus på Gade 9 for 2200.0
Gitte er landet på Gade 9
Dette er Gittes egen grund
***** Søren på felt 8 slår 2 *****
Søren passerer Gade 9
Søren er landet på start og modtager 5000.0
Sørens konto lyder nu på 17300.0
***** Gitte på felt 9 slår 1 *****
Gitte er landet på start og modtager 5000.0
Gittes konto lyder nu på 12000.0
***** Søren på felt 0 slår 3 *****
Søren passerer Gade 1
Søren passerer Gade 2
Søren bygger et hus på Gade 3 for 1200.0
Søren er landet på Gade 3
Dette er Sørens egen grund
***** Gitte på felt 0 slår 5 *****
Gitte passerer Gade 1
Gitte passerer Gade 2
Gitte passerer Gade 3
Gitte passerer Maersk
Gitte bygger et hus på Gade 5 for 1500.0
Gitte er landet på Gade 5
Dette er Gittes egen grund
***** Søren på felt 3 slår 5 *****
Søren passerer Maersk
Søren passerer Gade 5
Søren passerer Helle
Søren passerer Gade 7
Søren bygger et hus på Gade 8 for 2000.0
Søren er landet på Gade 8
Dette er Sørens egen grund
***** Gitte på felt 5 slår 1 *****
```

```
Gitte er landet på helle, og får overført 15000.0
Gittes konto lyder nu på 25500.0
***** Søren på felt 8 slår 3 *****
Søren passerer Gade 9
Søren har passeret start og modtager 5000.0
Sørens konto lyder nu på 19100.0
Søren er landet på Gade 1
Søren køber Gade 1 for 10000.0
```

... (og så videre)

6.3.1. 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.

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.

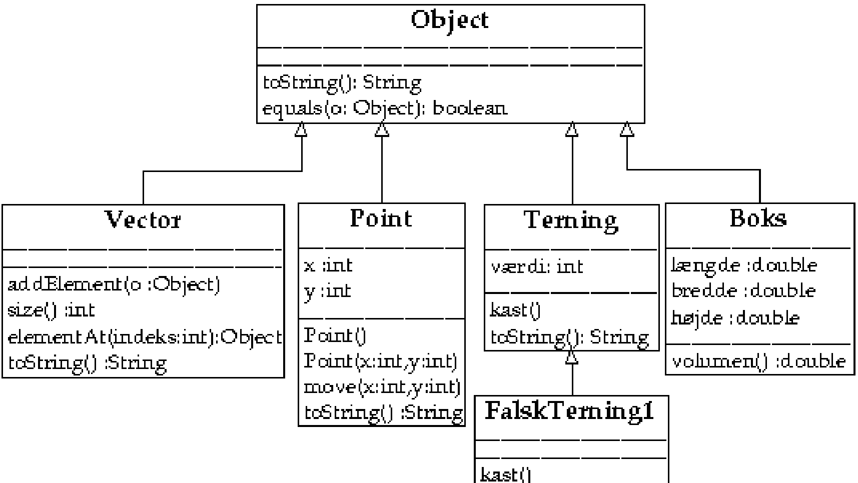
6.4. Stamklassen Object

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.

Figur 6-7. Java



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
...
```

Resultatet bliver:

```
  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, 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 vektorer indeholder de samme elementer.

6.4.1. 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 o;
o=new Point();
o="hej";
o=new FalskTerning2();
```

Omvendt kan man ikke rigtig bruge variablen til noget, før man har lavet en eksplicit referencetypekonvertering:

```
Terning t;
t=(Terning) o;
t.kast();
```

Her var jeg heldig, at o faktisk refererede til en (underklasse) af Terning. Jeg får først at vide, om min typekonvertering er gået godt på kørselstidspunktet.

Tilsvarende kan man bruge Object som parameter / returtype og få et fleksibelt, men ikke særlig sikkert program. Klassen Vector benytter sig af dette: Vektorer arbejder med lister af typen Object, det er derfor, man kan gemme alle slags objekter i dem.

```
vector v = new Vector();
Point p = new Point();
v.addElement(p);
```

Metoden addElement() tager et objekt 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):

```
vector v = new Vector();
Point p = new Point();
Object o;                   // overflødig mellemvariabel
o = p;                      // implicit reference-typekonvertering
v.addElement(o);
```

Når man kalder elementAt() 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) v.elementAt(0);
```

Igen vises det samme blot med mellemvariabel, så man kan se, hvilken typekonvertering der finder sted:

```
o = v.elementAt(0);         // ingen konvertering
p= (point) o;               // eksplicit reference-typekonvertering
```

6.5. Konstruktører i underklasser

Vi minder om, at:

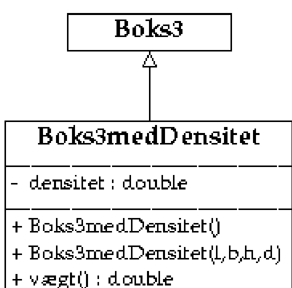
- Konstruktører definerer, hvordan objekter oprettes, og med hvilke parametre de må oprettes. Der kan kun oprettes objekter på en måde, der passer med en konstruktør.
- Hvis en klasse ikke har defineret nogen konstruktører, så defineres automatisk en standardkonstruktør (uden parametre og med tom metodekrop). F.eks. har Boks automatisk fået en tom konstruktør, så den kunne oprettes med new Boks().

En underklasse skal selv definere, hvordan dets objekter skal kunne oprettes, så den skal selv definere sine konstruktører. Underklassen kan også 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.

Figur 6-8. Boks3medDensitet tillader oprettelse på to måder



Herunder definerer vi Boks3medDensitet, der arver fra Boks3. Den nye egenskab er massefylden og metoden vægt(). Den skal kunne oprettes med: new Boks3medDensitet(), som opretter boksen med nogle standardværdier eller med: new Boks3medDensitet(lgd,b,h,d), hvor d er densiteten (massefylden).

```
public class Boks3medDensitet extends Boks3
{
  private double densitet;

  public Boks3medDensitet()
  {
```

```
    // super(); overflødig, den kaldes implicit
    densitet = 10.0;
  }

  public Boks3medDensitet(double lgd, double b,
          double h, double densitet)
  {
    // kald superklassens konstruktør
    super(lgd,b,h);
    this.densitet = densitet;
  }

  public double vægt()
  {
    return volumen() * densitet;    // superklassen udregner volumen for os
  }
}
```

Konstruktører skal defineres på ny i en underklasse

En konstruktør i en underklasse kalder først en af superklassens konstruktører

Superklassens konstruktør kan kaldes med: super(*parametre*)

Hvis programmøren ikke kalder en af superklassens konstruktører, indsætter Java automatisk et kald af superklassens konstruktør uden parametre

Disse 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)
  {
  }
}
```

og

```
public class B extends A
{

}
```

Dette vil ikke oversætte, fordi B af Java vil blive lavet om til:

```
public class B extends A
{
  public B() // 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*".

Der er derimod ingen problemer med:

```
public class A
{
}
```

og

```
public class B extends A
{
}
```

Java laver det om til:

```
public class A
{
  public A() // indsættes automatisk af Java
  {
  }
}
```

og

```
public class B extends A
{
  public B() // indsættes automatisk af Java
  {
    super();
  }
}
```

6.6. 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)
  {
    System.out.println(sp.navn+" er landet på "+navn);
    if (sp==ejer)
    {                                       // spiller ejer feltet
      System.out.println("Dette er "+sp.navn+"s egen grund");
    }
    else if (ejer==null)
    {                                       // ingen ejer grunden, så køb den
      if (sp.konto > pris)
      {
        System.out.println(sp.navn+" køber "+navn+" for "+pris);
        ejer=sp;
        sp.transaktion( -pris );
      }
      else System.out.println(sp.navn+" har ikke penge nok til at købe "+navn);
    }
    else
    {                                       // felt ejes af anden spiller
      double leje = beregnLeje();
```

```
      System.out.println("Husleje: "+leje);
      sp.betal(ejer, leje);                 // spiller betaler til ejeren
    }
  }
}
```

Nu er Rederi ret nem. Den skal nemlig (i denne simple udgave) opføre sig præcis som Grund. Vi skal blot definere konstruktøren, som skal kalde den tilsvarende konstruktør i Grund:

```
// Rederier

public class Rederi2 extends Grund2
{
  public Rederi2(String navn, double pris, double leje)
  {
    super(navn, pris, leje);    // kald 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, hvis spilleren, der er landet på gaden, ikke er ejeren. Hvis det *er* ejeren, prøver vi at bygge et hus (udskilt i metoden bygHus()).

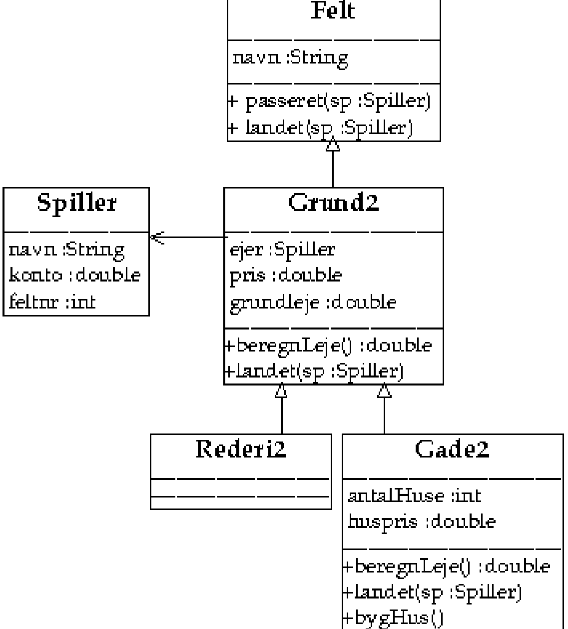
```
// En gade der kan bebygges

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);
    this.huspris=huspris;
    antalHuse = 0;
  }

  public double beregnLeje()                         // tilsidesæt Grund2's
  {
    return grundleje + antalHuse * huspris;
  }

  public void landet(Spiller sp)
  {
    if (sp==ejer)
    {                                                // eget felt; byg hus
      System.out.println(sp.navn+" er landet på "+navn);
      System.out.println("Dette er "+sp.navn+"s egen grund");
      if (antalHuse<5 && sp.konto>huspris) bygHus();  // byg hus hvis vi kan
```

```
    }
    else super.landet(sp);                           // brug gamle landet()
  }

  public void bygHus()
  {
    System.out.println(ejer.navn+" bygger et hus på "+navn+" for "+huspris);
    ejer.transaktion( -huspris );
    antalHuse = antalHuse + 1;
  }
}
```

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 (ejeren), er der en pil fra Grund2 til Spiller, en *har-en*-relation. Resten af pilene symboliserer *er-en*-relationer, f.eks. Gade2 *er en* Grund2, Grund2 *er et* Felt.

Figur 6-9. Java



6.7. Opgaver

1. Lav en LudoTerning, der arver fra Terning. Tilsidesæt toString() med en, der giver "*" på en 3er og "globus" på en 4er. Afprøv klassen.
2. Tilføj en Bryggeri-klasse til matadorspillet (version 1) og prøv om det virker.På bryggerier afhænger lejen af, hvor stort et slag spilleren slog, da han landede på feltet, men start med at lade lejen være tilfældig. Du kan evt. kopiere Gade.java i stedet for at skrive koden forfra. Husk at indsætte et bryggeri i felter-vektoren i main().
3. Ret SpilMatador 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.
4. 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. Ret på Spiller, så slag er en objektvariabel i stedet for en lokal variabel. Nu kan man udefra aflæse, hvad spilleren slog sidst. Brug værdien af slag i landet()-metoden i Bryggeri til at lade lejen afhænge af, hvad spilleren slog.

Kapitel 7. 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 5, 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 i funktion.

En pakke er en 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.

7.1. 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 Vector-klassen, kunne vi skrive:

```
java.util.Vector v;
v = new java.util.Vector();
```

Det er jo lidt besværligt, og derfor kan vi vælge øverst i kildetekstfilen at skrive:

```
import java.util.Vector;
```

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:

```
Vector v;
```

```
    v = new Vector();
```

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

7.2. 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, Vector og meget andet
- java.awt Abstract Window Toolkit. Basal vinduesbaseret programmering og grafik
- java.applet klasser til understøttelse af appletter
- java.io IO-funktioner, filhåndtering og datastrømme
- java.net netværksfaciliteter
- java.rmi Remote Method Invocation - til distribuerede systemer
- java.sql databaseadgang (også kaldet JDBC)
- java.text håndtering af tekst uafhængigt af sprog
- javax.swing avanceret vinduesbaseret programmering

Hvorfor hedder den sidste javax? javax betyder, at sproget er udvidet med nogle ting, som ikke på nuværende tidspunkt er en del af det egentlige standardbibliotek, og som måske er bestemt til aldrig at blive det. Et andet eksempel på javax er javax.comm, som er en kommunikationspakke, der håndterer seriel og parallel transmission af data.

7.2.1. 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()).

7.3. 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 underkataloger på et filsystem.

En klasse svarer til en fil på filsystemet

En pakke svarer til et underkatalog på filsystemet

For eksempel findes klassen java.util.Vector som filen Vector.class i et katalog, der hedder util, som er et underkatalog til et katalog, der hedder java: java/util/Vector.class (i DOS og Windows som: java\util\Vector.class).

Ofte er klasserne og katalogerne pakket sammen i et såkaldt Java-arkiv (.jar-fil). jar-filer minder meget om zip-filer.

Oversætteren skal kende pakkens fysiske placering i filsystemet:

1. Som et underkatalog med samme navn som pakken.
2. I et underkatalog 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 klassedefinitioner.

7.4. 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 datobearbejdningspakker.

7.4.1. Eksempel på brugen af egne pakker

I følgende eksempel findes to klasser, nemlig Klasse1 og Klasse2 i en pakke (der hedder minPakke). De bruges af den kørbare klasse BrugPakker:

```
import minPakke.*;

public class BrugPakker
{
  public static void main(String args[] )
  {
    Klasse1 a = new Klasse1();
    Klasse2 b = new Klasse2();
    a.snak();
    b.snak();
  }
}
```

Klasse1 og Klasse2 skal ligge i et underkatalog, der hedder minPakke:

```
// Filnavn: minPakke/Klasse1.java
package minPakke;

public class Klasse1
{
  public void snak()
  {
    System.out.println("Dette er Klasse1, der taler!");
  }
}
```

og

```
// Filnavn: minPakke/Klasse2.java
package minPakke;

public class Klasse2
{
  public void snak()
  {
    System.out.println("Dette er Klasse2, der taler!");
  }
}
```

7.4.2. Navngive pakker

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.

7.4.3. Pakke klasser i JAR-filer (Java-arkiver)

Laver man sine egne pakker, ønsker man ofte at kunne distribuere dem til andre. De skabes med et ZIP-værktøj som WinZip eller GnoZip til Linux eller fra kommandolinjen med jar, der følger med, når man installerer Java. Kommandoen jar minder meget om UNIX' tar-kommando. Man opretter et arkiv ved at skrive f.eks.:

```
jar cf minPakke.jar minPakke
```

Dette vil oprette JAR-filen minPakke.jar med klasserne minPakke/Klasse1.class og minPakke/Klasse2.class.

7.5. Opgaver

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 ligger filerne organiseret?

Prøv, om du kan finde filen, der indeholder Vector-klassen.

Kapitel 8. Lokale, objekt- og klassevariable

Indhold:

- Klassevariable
- Repetition af objektvariabler og lokale variabler
- Rekursion

Forudsættes ikke i resten af bogen.

Forudsætter Kapitel 5, Definition af klasser.

De variabler, vi er stødt på indtil nu, har enten været lokale variabler eller objektvariabler.

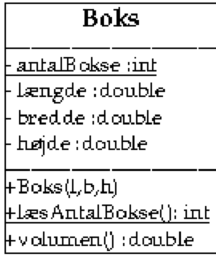
Objektvariabler hedder sådan, fordi de bliver oprettet for hvert objekt.

Der findes også variabler, der eksisterer "i klassen", uafhængigt af om der bliver oprettet objekter. Disse kaldes klassevariable og erklæres med nøgleordet *static*, og de kaldes også statiske variabler.

Herunder ses et eksempel på en klassevariabel og en klassemetode (antalBokse).

Klassevariabler og -metoder vises med understregning i UML-notationen (diagrammet til højre).

Figur 8-1. Java




```
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;
  }
}
```

Variablen antalBokse er en klassevariabel, fordi den er erklæret med static-nøgleordet. Dette betyder, at variablen 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.

Variablerne bredde, højde og længde er objektvariabler, fordi hvert Boks-objekt har tilknyttet en af hver.

Og for fuldstændighedens skyld: 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.

En klassemetode er en metode, der er erklæret static. Den arbejder på klasseniveau (uafhængigt af om der er skabt nogen objekter) og kan derfor ikke anvende objektvariabler eller -metoder.

Vi kan afprøve Boks4 med:

```
public class BenytBoks4
{
  public static void main(String args[])
  {
    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());
  }
}
```

Resultatet bliver:

```
Antal bokse: 0
Antal bokse: 1
Antal bokse: 3
```

Inde fra objektet bruges statiske variabler og metoder ligesom de almindelige variabler og metoder. Det ses f.eks. i Boks' konstruktør:

```
    antalBokse = antalBokse + 1;
```

8.1. Eksempler i standardpakkerne

Du har allerede benyttet dig af et par statiske metoder og variabler.

8.1.1. Klassevariable

Der er mange klassevariabler i standardpakkerne. Af de oftest brugte kan nævnes

- Math.PI - værdien af pi er en klassevariabel i Math-klassen (pakken java.lang).
- System.out - systemoutputtet er et PrintStream-objekt, der bl.a. har metoderne print() og println(). Objektet er en klassevariabel i System-klassen (pakken java.lang).
- Color.black - et Color-objekt, der repræsenterer sort. Objektet ligger som en klassevariabel i (selvsamme) Color-klasse (pakken java.awt).

Som det ses, er klassevariabler nyttige til variabler, der er tilgængelige overalt. Det er det nærmeste man kommer globale variabler i Java, som det kendes fra andre programmeringssprog.

8.1.2. Klassemetoder

Af nyttige klassemetoder 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), Math.pow(double x, double y), Math.max(double x, double y), Math.min(double x, double y), ...
- 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.
- Integer.parseInt(String s) returnerer værdien af *s* som et heltal. F.eks. giver Integer.parseInt("13") tallet 13.
- String.valueOf(double d) gør det modsatte af Double.parseDouble, den returnerer nemlig en streng, som repræsenterer et flydende kommatal. 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 tegn, f.eks. mellemrum, linjeskift, tabulator.
- System.exit() - stopper programudførelsen og afslutter Java.
- 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 main er da også en klassemetode. Der oprettes aldrig nogen BenytBoks-objekter!

8.2. 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 variablens 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,
```

```
                                 // fordi 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.
```

8.2.1. Parametervariable

Parametervariablerne 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
  }
  System.out.println();
}

  ....
  int stj=10;
  udskrivStjerner(stj); // kald af udskrivStjerner
  // stj er stadig 10.
  ...
```

Dette mærker kalderen intet til, netop fordi kalderens værdi blev kopieret. Her skal man være opmærksom på forskellen mellem variabler af primitiv type og variabler af objekt-type. Fordi det sidste er referencer, peger parametervariablen på samme objekt som kalderen, når den bliver kopieret. Ved at ændre i objektet, som parametervariablen refererer til, kan man derfor ændre på kalderens objekt.

Derfor kan metoden herunder godt ændre på kalderens punkt-objekt:

```
public void flyt(Point p, int dx, int dy)
{
  p.x=p.x+dx;  // OK, vi ændrer på kalderens objekt
  p.y=p.y+dy;
}

  ...
  Point p1=new Point();
```

```
    p1.x=10;p1.y=10;
    flyt(p1,10,10);
    // nu er p1 (20,20)
    ...
```

Men man kan stadig ikke ændre på kalderens reference. Dvs. p1's værdi:

```
  public void flyt(Point p, int dx, int dy)
  {
    // hmm... vi glemmer kalderens objekt, men det opdager han ikke
    p=new Point(p.x+dx,p.y+dy);
  }

    ...
    Point p1=new Point();
    p1.x=10;p1.y=10;
    flyt(p1,10,10);
    // nu er p1 stadig (10,10)
    ...
```

En lokal variabel oprettes, når man går ind i blokken, hvor den er defineret, og nedlægges igen, når man går ud af blokken. Der bliver oprettet en ny variabel, hver gang programudførelsen går ind i blokken.

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 ikke bare kan ændre ens lokale variabler.

8.2.2. Rekursion

Rekursion er en teknik, der netop udnytter, at der bliver oprettet en ny omgivelse med nye lokale variabler, hver gang en metode kaldes. En rekursiv metode er en metode, der kalder sig selv. F.eks.:

```
  void tælNed(int tæller)
  {
    System.out.print((+tæller+ );
    if (tæller>=0) tælNed(tæller-1); // tælNed kalder sig selv !!
    System.out.print( +tæller+));
  }
```

Hvis man kalder tælNed(4), får man udskrevet følgende:

```
(4(3 (2 (1 (0  0) 1) 2) 3) 4)
```

Fidusen er, at parameteren tæller eksisterer én gang for hver gang, tælNed() kalder sig selv. Så når tælNed() vender tilbage til kalderen, som også er tælNed(), er tællers værdi bevaret som før kaldet.

Visse problemstillinger kan løses meget elegant med rekursion, men vi vil ikke her komme yderligere ind på emnet.

Kapitel 9. Arrays

Kapitlet forudsættes ikke i resten af bogen, men er nyttigt i praktisk programmering.

Forudsætter Kapitel 4, Objekter (og et enkelt sted Kapitel 6, Nedarvning).

Ofte har man behov for at håndtere et større antal objekter eller simple typer på en ensartet måde. Indtil nu har vi gjort det med lister af typen Vector, 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 Vector.

Ligesom med Vector skal man skelne mellem array-variablen og array-objektet. Array-variablen refererer til array-objektet, som indeholder variablerne.

9.1. Erklæring og brug

Man erklærer en array-variabel med den type data, man ønsker at lave et array af, umiddelbart efterfulgt af "[]", 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 har som udgangspunkt værdien 0. Havde arrayet indeholdt referencer til objekter, var de blevet sat til null.

Arrayets værdier kan sættes og aflæses ved at angive indeks i firkantede []-parenteser efter variabelnavnet:

```
public class ArrayEksempel1
{
```

```
  public static void main(String[] args)
  {
    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();
  }
}
```

Resultatet bliver:

```
28 0 13 41 0 0
```

Indekseringen starter altid fra 0 af, og sidste lovlige indeks er dermed lig med arrayets længde-1. Indekserer man uden for arrayets grænser, kastes undtagelsen 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 (og det gamle arrays og dets værdier er glemt).

9.1.1. Eksempel: Statistik

Arrays er gode til at lave statistik med. Her laver vi statistik på slag med to terninger:

```
import java.util.*;

public class TerningStatistik
{
  public static void main(String[] args)
  {
    int[] antal = 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);

      antal[sum] = antal[sum]+1;  // optæl statistikken for summen af øjne
    }

    for (int n=2; n<=12; n=n+1) System.out.println( n + ": " + antal[n]);
  }
}
```

Resultatet bliver:

```
2: 2
3: 9
4: 7
5: 17
6: 14
7: 15
8: 8
9: 9
10: 8
11: 7
12: 4
```

9.2. Main-metoden

Main-metoden, som vi har defineret utallige gange, tager en parameter, som er et array af strenge. Dette array indeholder kommandolinje-parametrene ved kørsel af programmet.

```
public class Kommandolinje
{
  public static void main(String[] args)
  {
    System.out.println("Antallet af parametre er: " + args.length);

    for (int i=0; i< args.length; i=i+1)
      System.out.println("Parameter "+i+" er: " + args[i]);
  }
}
```

Resultatet bliver:

```
Antallet af parametre er: 3
Parameter 0 er: x
Parameter 1 er: y
Parameter 2 er: z
```

Programmet herover er kørt fra prompten med "java Kommandolinje x y z".

9.3. Arrays med startværdier

Arrays kan initialiseres med startværdier i { } og er adskilt med komma.

```
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[] args)
  {
    int[] måneder = {31,28,31,30,31,30,31,31,30,31,30,31};

    System.out.println("Længden af januar er: " + måneder[0]);
    System.out.println("Længden af april er: " + måneder[3]);

    for (int i=0; i <måneder.length; i++)
      System.out.print(måneder[i] + ", ");

    System.out.println();
  }
}
```

Resultatet bliver:

```
Længden af januar er: 31
Længden af april er: 30
31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
```

9.4. Gennemløb og manipulering

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 da også konstant. Den eneste måde at få et array af en anden størrelse er at oprette et andet array og så kopiere det gamle indhold over i det nye array.

Herunder ses, hvordan man kan fjerne et element fra et array.

```
public class FjernEtElement
{
```

```
  public static void main(String args[])
  {
    // 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();
  }
}
```

Resultatet bliver:

```
a før: 0 10 20 30 40 50 60 70 80 90
a efter: 0 10 20 30 40 60 70 80 90
```

9.5. 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 objekterne. Herover oprettes altså ingen punkter! Dvs. pkt[0], pkt[1],...,pkt[9] er alle null.

Arrays kan bruges til at gå mellem værdier fra et domæne til værdier i et andet domæne. For eksempel konvertering af måneders numre (1-12) til deres navne:

```
public class MaanedersNavne
```

```
{
  public static void main(String[] args)
  {
    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 9. måned er " + måneder[9] );
  }
}
```

Resultatet bliver:

```
Den 1. måned er januar
Den 6. måned er juni
Den 9. måned er september
```

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) };
```

9.5.1. Polymorfi

Ligesom med almindelige variabler kan elementerne i et array godt referere til nedarvinger.

```
// Bruger Terning.java og FalskTerning2.java fra tidligere kapitler
public class Terninger
{
  public static void main(String[] args)
  {
    Terning[] t={new Terning(), new FalskTerning2(), new FalskTerning2()};

    for (int i=0; i<t.length; i++) t[i].kast();
  }
}
```

9.6. Arrays versus vektorer

Som det ses, er det besværligt at ændre størrelsen på et array, f.eks. når der skal indsættes eller slettes elementer. Til gengæld kan arrays indeholde simple typer, det er nemmere at få adgang til elementerne, og man kan initialisere et array på én linje. Faktisk er Vector-klassen et array i en indpakning, der gør det nemmere at bruge. Hvad du vælger er op til dig selv.

En vektor 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

Der er desuden den fordel ved arrays, at de er typesikre. Vi kan ikke komme til at lægge værdier af forkerte typer ind i et array - så stopper oversætteren os.

9.7. Opgaver

1. Lav et program, der simulerer kast med 2 terninger. Der udføres f.eks. 100 kast. Optæl i et array hyppigheden af summen af øjenantallene.
2. Lav programmet om til at lave hyppighedsstatistik på med 6 terninger. Udvid det derpå til at kunne lave statistik på kast med et vilkårligt antal terninger.

Kapitel 10. Appletter og grafik

Indhold:

- At lægge Java i en hjemmeside
- At tegne simpel grafik
- Metoder i en applet

Kapitlet forudsættes af Kapitel 11, Grafiske brugergrænseflader og Kapitel 12, Interfaces.

Forudsætter Kapitel 4, Objekter (Kapitel 5, Definition af klasser og Kapitel 6, Nedarvning er en fordel).

En applet er et javaprogram i en hjemmeside. Når siden vises, vil browseren (fremviseren af HTML-dokumentet) 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. En hjemmeside med en applet vil have en HTML-kode, der henviser til, hvor browseren skal finde programkoden. Det ser sådan her ud:

```
<applet code="MinApplet.class" width=400 height=300> </applet>
```

Her blev angivet, at appletten hedder MinApplet, og den skal være 400 punkter bred og 300 høj. MinApplet.class, den binære kode fra MinApplet.java, skal ligge i samme katalog som hjemmesiden.

HTML-koder er skrevet mellem < og >. Et helt 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
          CODEBASE = "."
          CODE     = "MinApplet.class"
          WIDTH    = 400
          HEIGHT   = 300>
        </APPLET>
```

```
        Slut herfra!
</BODY>
</HTML>
```

For mere viden om HTML henvises til de mange introduktioner til, hvordan man laver hjemmesider.

10.2. Javakoden

Selve javaprogrammet er en klasse, der arver fra Applet. Her skal paint()-metoden, som kaldes når appletten skal tegnes på skærmen, defineres. Til dette formål får paint() et Graphics-objekt (beskrevet i Afsnit 10.4.3) overført, som vi kan tegne med.

I eksemplet nedenfor tegner vi en linje, en fyldt oval og noget tekst med grøn skrift.

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

public class MinApplet extends Applet
{
  public void paint(Graphics g)
  {
    // Herunder referer g til et Graphics-objekt man kan tegne med.
    g.drawLine(10,10,50,70);

    g.fillOval(5,5,300,50);

    g.setColor(Color.green);

    g.drawString("Hej grafiske verden!",100,30);
  }
}
```

Her ses, hvordan HTML-koden med appletten ser ud i en browser (Netscape under Linux).

Figur 10-1. Java

Man ser, at først kommer HTML-teksten "Velkommen til ...", derunder appletten og nederst igen noget tekst fra HTML-koden.

10.2.1. Metoder i appletter, som du kan kalde

Den vigtigste metode er repaint(), som forårsager, at appletten bliver gentegnet (ved, at systemet kalder paint()).

Andre rare metoder er getImage(), der giver dig mulighed for at indlæse grafik, og getSize(), der giver applettens størrelse. De er beskrevet i appendiks senere i kapitlet (Afsnit 10.4.1).

10.2.2. Metoder i appletter, som systemet kalder

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 en række metoder, som kan tilsidesættes efter behov.

Den vigtigste er paint(), som systemet kalder, hver gang der er behov for at tegne appletten, f.eks. hvis den har været dækket af et andet vindue.

En anden er init(), der kaldes, når appletten indlæses som en del af HTML-dokumentet. Metoden bliver kun kaldt én gang, så det er en god idé at placere programkode, der opretter objekter og initialiserer programmet, i init(). Da appletten er et objekt, kan man selvfølgelig også gøre det i konstruktøren, men da skal man være opmærksom på, at de metoder du kan kalde (f.eks. repaint() og getSize()), ikke har nogen virkning, da applettens omgivelser ikke er blevet initialiseret endnu.

Det er en god idé at initialisere variabler og oprette objekter i init()-metoden, og kun have selve gentegningen i paint().

10.2.3. Eksempel

Her er et program, der tegner en kurve over sinus-funktionen.

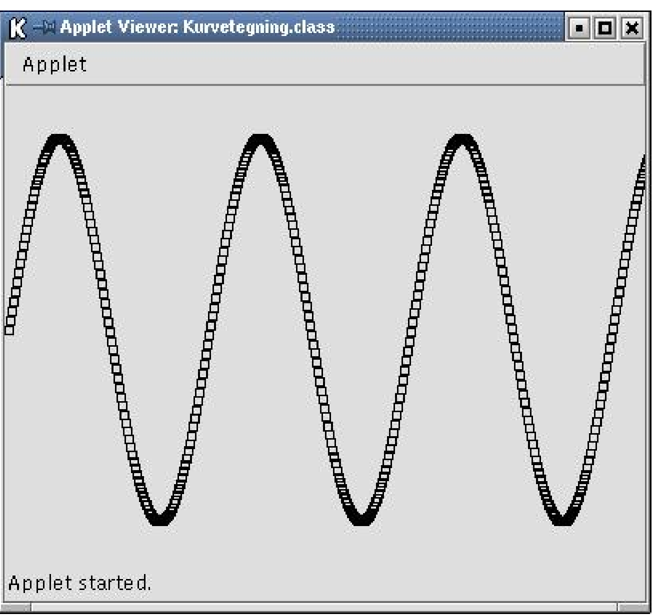
Det definerer init()-metoden, hvor det udregner koordinater for alle punkterne, der skal tegnes (her bruger vi getSize() for at vide, hvor stor appletten er).

Vi tegner punkterne i paint(), der er gjort så lille og hurtig som muligt (den kaldes jo hver gang appletten bliver gentegnet).

Punkterne huskes i en vektor, der er defineret som objektvariabel, sådan at den er kendt, så længe Kurvetegning-objektet findes. På den måde får vi data fra init() over til paint().

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

public class Kurvetegning extends Applet
{
  Vector punkter; // objektvariabel kendt i både init() og paint()

  public void init()  // Forbered punkterne
  {
    punkter = new Vector();
    int br = getSize().width;  // applettens bredde
    int hø = getSize().height; // applettens højde

    for (int i=0; i<br; i++)
    {
      double y = 0.5*hø - 0.4*hø*Math.sin((double) i*20 / br);
      punkter.addElement(new Point(i, (int) y ));
    }
  }

  public void paint(Graphics g) // tegn punkterne
  {
    for (int i=0; i<punkter.size(); i=i+1)
    {
      Point p = (Point) punkter.elementAt(i);
      g.drawRect(p.x, p.y, 5, 5);
    }
  }
}
```

Figur 10-2. Java



10.3. Opgaver

1. Lav en applet, der viser et digitalur som tekst (vink: Brug et Date-objekt).
2. Lav en applet, der viser et analogt ur.

10.4. Appendiks

Appletter har nogle metoder, som det kan være nyttigt at kende. De er delt i to grupper, nemlig dem *du kan kalde*, og dem *systemet kalder*, og som du kan omdefinere for at få udført noget af din kode, når de kaldes.

10.4.1. Metoder i appletter, som du kan kalde

Disse metoder står til din rådighed, når du programmerer appletter. Det er kun de vigtigste af metoderne, der er gengivet (se Javadokumentationen for Applet og Component for en komplet liste).

Nogle af Applet-klassens metoder

repaint(int millisekunder)forårsager at systemet kalder paint() på appletten lidt senere.

Dimension *getSize*()returnerer applettens højde og bredde i et Dimension-objekt (der har variablerne width og height).

URL *getCodeBase*()giver URL'en til CODEBASE, dvs. hvor .class-filen er.

URL *getDocumentBase*()giver URL'en til der, hvor HTML-dokumentet ligger.

AudioClip *getAudioClip*(URL url, String filnavn)returnerer et lydklip-objekt, typisk fra en .wav-fil.

Image *getImage*(URL url, String filnavn)returnerer et billede-objekt, typisk fra en .png eller .png-fil.

String *getParameter*(String parameternavn)returnerer den pågældende parameterværdi, hvis den er defineret i HTML-koden, ellers null. En parameter sættes med <PARAM name="navn" value="værdi"> før </APPLET>.

10.4.2. Metoder, som systemet kalder

Appletter har en række metoder, som du selv kan definere, og som systemet vil kalde.

```
public void paint(Graphics g)
```

Her programmerer du, hvordan appletten skal se ud på skærmen ved at kalde metoder på Graphics-objektet g (dets metoder er forklaret i næste afsnit).

Metoden kaldes af systemet, 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 init()
```

Kaldes, når fremviseren indlæser HTML-dokumentet og appletten. Her kan du lægge kode, der initialiserer programmet. init() bliver kun kaldt én gang.

```
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 stop()
```

Kaldes, når appletten bliver skjult. Det kan være, fordi vinduet bliver minimeret, eller fordi brugeren går til et andet dokument. Ligesom start() kan stop() godt blive kaldt flere gange.

```
public void destroy()
```

Kaldes, når appletten smides væk af fremviseren, 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().

10.4.3. 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", når styresystemet afgører, at vinduet skal tegnes op. Herunder gengives kun nogle af metoderne - se Javadokumentationen for en komplet liste.

java.awt.Graphics - todimensional grafiktegning

Metoder

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 *drawString*(String tekst, int x, int y)

tegner tekst med øverste venstre hjørne i (x,y).

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 *fillRect*(int x, int y, int bredde, int højde)

tegner et udfyldt rektangel.

void *fillOval*(int x, int y, int bredde, int højde)

tegner en udfyldt oval med øverste venstre hjørne i (x,y). Er bredde==højde, tegnes en cirkel.

void *fillArc*(int x, int y, int bredde, int højde, int startvinkel, int slutvinkel)

tegner en udfyldt del af en oval, men kun fra *startvinkel* til *slutvinkel* (mellem 0 og 2 pi).

Rectangle *getClipBounds*()

giver grafik-objektets klipnings-omrids. Kun punkter inden for dette omrids bliver faktisk tegnet, ting uden for omridset bliver beskåret til den del, der er inden for omridset.

void *setColor*(Color nyFarve)

sætter tegnefarven 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.

Kapitel 11. Grafiske brugergrænseflader

Indhold:

- Design af en grafisk brugergrænseflade med et værktøj
- De vigtigste grafiske komponenter og deres egenskaber
- Containere og layout-managers
- Større opgave: Matador-spillet som en applet

Forudsættes af Kapitel 13, Hændelser.

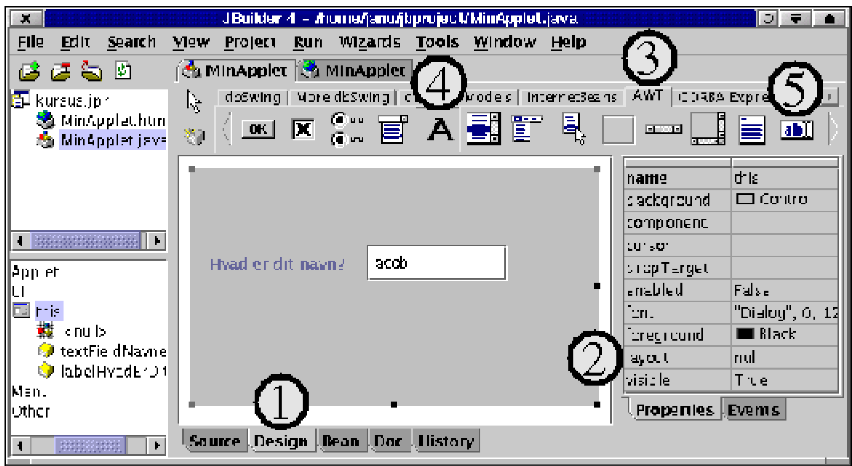
Kapitlet forudsætter Kapitel 10, Appletter og grafik, og at du har adgang til et værktøj, der kan udvikle grafiske brugergrænseflader (f.eks. Borland JBuilder, Oracle JDeveloper, Sun Forte eller WebSphere). Den større opgave forudsætter Kapitel 6, Nedarvning.

Når man skal lave en grafisk brugergrænseflade, gøres det ofte ved at anvende standardkomponenter. Vi vil starte med at se på, hvordan det gøres i praksis med et værktøj.

11.1. Generering med et værktøj

Med moderne udviklingsværktøjer kan man udarbejde en grafisk brugergrænseflade ud fra standardkomponenter på ret kort tid. Herunder er beskrevet, hvordan man gør i Borland JBuilder. JDeveloper har helt de samme muligheder og udseende. Hvis du bruger et andet værktøj, må du prøve dig lidt frem. Ideerne er de samme, og koden, der genereres, ligner også nogenlunde, men menuerne og knapperne varierer selvfølgelig noget.

Figur 11-1. Java



Tag en eksisterende applet, f.eks. MinApplet fra kapitlet om appletter, og føj den til et projekt. Hvis du i stedet vil oprette en ny, så vælg "New.." og Applet. Fjern pakkenavnet, skriv et navn på din klasse, vælg superklasse ("base class") Applet, og klik "Finish". Definér evt. en paint()-metode, der tegner noget (hvis du bruger et andet værktøj end JBuilder, så find menuerne til at oprette en ny applet, og gør det).

1. Gå over på Design-fanen (ved punkt 1 nederst). Den er delt op i en del, hvor du designer din brugergrænseflade til venstre, og en tabel af egenskaber til højre (punkt 2).
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).
3. Nu kan du gå i gang med at lægge komponenter ind på grænsefladen. Vælg i første omgang at arbejde med AWT-komponenter (punkt 3).
4. Vælg først en Label (det store A ved punkt 4), og klik på grænsefladen. Der dukker en mærkat 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 mærkaten. Ret den til f.eks. "Hvad er dit navn?".
5. Indsæt derefter et TextField (et indtastningsfelt -punkt 5). Ret variabelnavnet til textFieldNavn og teksten til f.eks. "Jacob".

Gå tilbage til Source-fanen. Nu ser kildeteksten således ud:

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

public class MinApplet extends Applet
{
  Label labelHvadErDitNavn = new Label();
  TextField textFieldNavn = new TextField();

  public void paint(Graphics g)
  {
    // Herunder referer g til et Graphics-objekt man kan tegne med.
    g.drawLine(10,10,50,70);

    g.fillOval(5,5,300,50);

    g.setColor(Color.green);
    g.drawString("Hej grafiske verden!",100,30);
  }

  public MinApplet() {
    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, null);
    this.add(labelHvadErDitNavn, null);
  }
}
```

De to objekter, vi satte på i designeren, er erklæret og oprettet øverst uden for metoderne:

```
  Label labelHvadErDitNavn = new Label();
  TextField textFieldNavn = new TextField();
```

Nedenunder står vores gamle paint() uændret. Herunder er der oprettet en konstruktør, der kalder metoden jbInit(). Den andet kode, 'try{ ... } catch (Exception e) {...}', er beregnet til at håndtere undtagelser, og vil blive forklaret senere i Kapitel 14, Undtagelser.

I metoden jbInit() nedenunder lægger JBuilder (og JDeveloper) al sin programkode. Man ser her, hvordan både Label og TextField har metoden setText(), og 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 på appletten.

"Design"- og "Source"-fanen i JBuilder (og JDeveloper) er to måder at se programmet på, og man kan frit skifte mellem dem. Laver man en designændring, vil det blive afspejlet i koden og omvendt. Prøv selv.

11.1.1. Interaktive programmer

Lad os nu tilføje en knap og et indtastningsfelt på flere linjer (TextArea). 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:

```
  void buttonOpdater_actionPerformed(ActionEvent e) {

  }
```

Hvis du kigger i jbInit(), kan du se, at JBuilder har indsat følgende kode:

```
    buttonOpdater.addActionListener(new java.awt.event.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 idé til, hvordan det skal gøres:

```
    String navn = textFieldNavn.getText();     // aflæs navnet

    textAreaHilsen.setText("Hej kære "+navn);  // sæt navnet
```

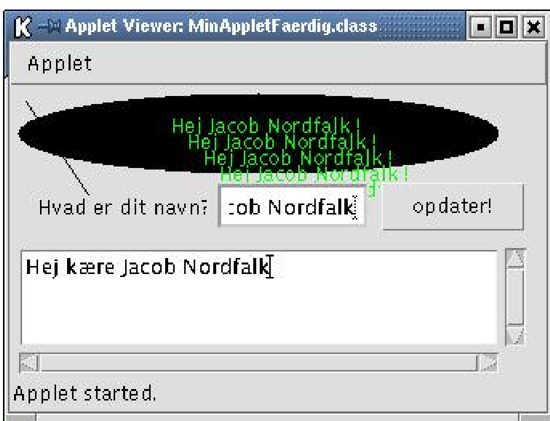
Her kommer det fulde eksempel. paint() er ændret til også at tegne navnet 5 gange.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class MinAppletFaerdig extends Applet
{
  Label labelHvadErDitNavn = new Label();
  TextField textFieldNavn = new TextField();
  Button buttonOpdater = new Button();
  TextArea textAreaHilsen = new TextArea();

  public MinAppletFaerdig()
  {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  public void paint(Graphics g)
  {
    g.drawLine(10,10,50,70);
```

```
    g.fillOval(5,5,300,50);

    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);
  }

  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.setLabel("opdater!");
    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, null);
    this.add(textAreaHilsen, null);
    this.add(buttonOpdater, null);
    this.add(textFieldNavn, null);
  }

  void buttonOpdater_actionPerformed(ActionEvent e) {
    String navn = textFieldNavn.getText();
    System.out.println("Opdater! navn="+navn);
    textAreaHilsen.setText("Hej kære "+navn);
    repaint(); // gentegn appletten
  }
}
```

Her har vi tastet "Jacob Nordfalk" ind og trykket på "opdater!"-knappen:

Figur 11-2. Java



Der er altså to måder at arbejde med grafik på:

- I paint() tegner man "i hånden" ved at give kommandoer til et Graphics-objekt.
- Ved at bruge grafiske standardkomponenter.

11.2. Komponenter

Komponenter er objekter, der bruges som en synlig del af en grafisk brugergrænseflade, f.eks. knapper, valglister, indtastningsfelter, mærkater.

Alle komponenter arver fra Component-klassen og har derfor dennes træk til fælles:

Metoderne setForeground(Color c) og setBackground(Color c) sætter farven hhv. 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.

Tabel 11-1. Java

Egenskab	Type	Sættes med	Læses med

Egenskab	Type	Sættes med	Læses med
foreground	Color	setForeground(Color c)	getForeground()
background	Color	setBackground(Color c)	getBackground()
font	Font	setFont(Font f)	getFont()
visible	boolean	setVisible(boolean synlig)	isVisible()

Nedenfor vil de mest almindelige komponenter blive beskrevet. Kun de nye egenskaber er beskrevet.

11.2.1. Button

Figur 11-3. En trykknap. Egenskaben *label* angiver, hvad der står på knappen.

Tabel 11-2. Java

Egenskab	Type	Sættes med	Læses med
label	String	setLabel(String t)	getLabel()

11.2.2. Checkbox

Figur 11-4. Et afkrydsningsfelt. Kan både bruges til flueben og til radioknapper. Hvis man skal have radioknapper (der gensidigt udelukker hinanden), skal objekterne knyttes sammen af et CheckboxGroup-objekt.

label angiver, hvad der står ved feltet. *state* angiver, om feltet er afkrydset.

Tabel 11-3. Java

Egenskab	Type	Sættes med	Læses med

Egenskab	Type	Sættes med	Læses med
label	String	setLabel(String t)	getLabel()
state	boolean	setState (boolean afkrydset)	getState()

11.2.3. Choice

Figur 11-5. En valgliste. Brug metoden add(String elementnavn) til at tilføje indgange.

Med getSelectedItem() undersøger man, hvad brugeren har valgt.

11.2.4. TextField

Figur 11-6. Et indtastningsfelt på en linje. Egenskaben *text* angiver, hvad der står i feltet.

Mindre brugt er: *columns* angiver, hvor bredt feltet skal være.

editable angiver, om brugeren kan redigere teksten i indtastningsfeltet.

echoChar bruges til felter, der skal skjule det indtastede, typisk adgangskoder.

Sæt f.eks. *echoChar* til '*' for at få vist stjerner i stedet for det indtastede.

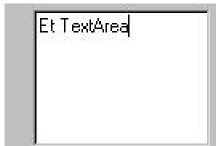
Tabel 11-4. Java

Egenskab	Type	Sættes med	Læses med
text	String	setText(String t)	getText()
editable	boolean	setEditable(boolean rediger)	isEditable()

Egenskab	Type	Sættes med	Læses med
columns	int	setColumns(int bredde)	getColumns()
echoChar	char	setEchoChar(char tegn)	getEchoChar()

11.2.5. TextArea

Figur 11-7. Et indtastningsfelt på flere linjer.



Egenskaberne *text, rows* og *columns* angiver, hvad der står i feltet, hhv. bredde og højde.

Tabel 11-5. Java

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()

TextField og TextArea har en del egenskaber til fælles, og disse fællestræk ligger i superklassen TextComponent (se klassediagrammet).

11.2.6. Label

Figur 11-8. En mærkat der viser en tekst (som brugeren ikke kan redigere i).

Egenskaben *text* angiver, hvad der står i feltet.

Tabel 11-6. Java

Egenskab	Type	Sættes med	Læses med
text	String	setText(String t)	getText()

11.2.7. List

Figur 11-9. En menu, hvor flere af indgangene kan ses samtidigt, og hvor man kan vælge en eller flere elementer. Brug metoden add(String elementnavn) til at tilføje indgange.

Med isIndexSelected(int index) undersøger man, om en indgang er valgt.

Egenskaberne *rows* og *multipleMode* angiver, hvad højden er, og om man kan vælge flere indgange samtidigt.

Tabel 11-7. Java

Egenskab	Type	Sættes med	Læses med
rows	int	setRows(int højde)	getRows()
multipleMode	boolean	setMultiple-Mode(boolean m)	getMultipleMode()

11.2.8. Canvas

Et tegne-område. Canvas er lidt besværlig, idet man skal nedarve fra klassen og implementere paint(Graphics g) for at kunne tegne noget.

En lettere (men ikke nødvendigvis altid smartere) måde er som sagt at definere paint()-metoden direkte i appletten/vinduet som vi har gjort tidligere.

11.3. Eksempel

Herunder et eksempel (genereret med JBuilder), der viser komponenterne omtalt i forrige afsnit. På billedet ses det resulterende skærmbillede under Windows (sidst i kapitlet ses, hvordan det ser ud under Linux).

Figur 11-10. Java

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

public class GrafiskeKomponenter extends Applet
{
  // opret alle komponenterne og husk dem i nogle objektvariabler
  Button button1 = new Button();
  Checkbox checkbox1 = new Checkbox();
  Checkbox checkbox2 = new Checkbox();
  Checkbox checkbox3 = new Checkbox();
  Checkbox checkbox4 = new Checkbox();
  Checkbox checkbox5 = new Checkbox();
  CheckboxGroup checkboxGroup1 = new CheckboxGroup();
  Choice choice1 = new Choice();
  TextField textField1 = new TextField();
  TextArea textArea1 = new TextArea();
  List list1 = new List();
  Label label1 = new Label();

  // JBuilder og JDeveloper definerer metoden jbInit() hvor de
  // initialiserer komponenterne. Det kunne dog lige så godt
  // ligge direkte i init()
  public void init() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  // initialisering af komponenterne med deres startværdier
  private void jbInit() throws Exception {
```

```
    button1.setLabel("OK");

    checkbox1.setLabel("En");     // Sæt afkrydsningsfelternes navne
    checkbox2.setLabel("To");
    checkbox3.setLabel("Tre");

    checkbox4.setLabel("Radio1"); // Sæt radioknappernes navne og
    checkbox5.setLabel("Radio2");
    checkbox4.setCheckboxGroup(checkboxGroup1); // gruppen de tilhører
    checkbox5.setCheckboxGroup(checkboxGroup1);
    checkboxGroup1.setSelectedCheckbox(checkbox4);

    choice1.add("Choice Rød");
    choice1.add("Choice Grøn");
    choice1.add("Choice Blå");

    textField1.setColumns(10);
    textField1.setText("Et TextField");

    textArea1.setColumns(15);
    textArea1.setRows(5);
    textArea1.setText("Et TextArea");

    label1.setText("En Label");

    list1.add("List Rød");
    list1.add("List Grøn");
    list1.add("List Blå");

    this.add(button1, null);    // til sidst skal komponenterne føjes
    this.add(checkbox1, null);  // til containeren (se senere)
    this.add(checkbox2, null);
    this.add(checkbox3, null);
    this.add(checkbox4, null);
    this.add(checkbox5, null);
    this.add(choice1, null);
    this.add(textArea1, null);
    this.add(textField1, null);
    this.add(label1, null);
    this.add(list1, null);
  }
}
```

11.4. Containere

En *container* er beregnet til at indeholde komponenter. De arver alle fra Container-klassen og har alle en såkaldt *layout manager* tilknyttet, der bestemmer, hvor og med hvilken størrelse komponenterne skal vises i containeren.

Før en komponent bliver vist, skal den tilføjes en container. I eksemplet ovenfor er appletten den container, komponenterne bliver tilføjet, og derfor står der sidst i initialiseringen:

```
this.add(button1, null);
```

11.4.1. Panel

Et panel er den simpleste og oftest brugte container. Den indeholder simpelt hen komponenterne (i henhold til layoutmanageren).

11.4.2. Applet

En applet er et udvidet panel, der er beregnet til at vise i en browser. Læs kapitlet om appletter for mere information.

11.4.3. Window

Window repræsenterer et vindue på skærmen. Det bruges meget sjældent direkte, man bruger i stedet arvningerne Frame og Dialog.

11.4.4. Frame

En Frame er den simpleste og oftest brugte måde at definere et "normalt" vindue med en titel.

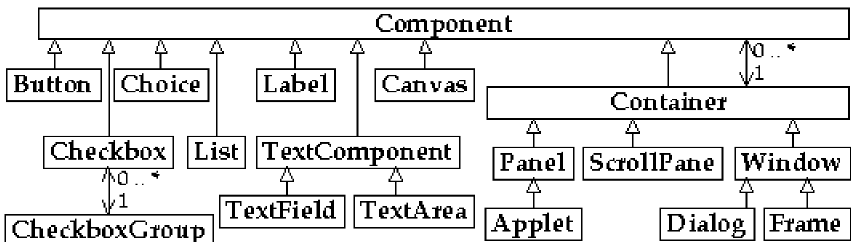
11.4.5. Dialog

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().

11.5. Relationer mellem klasserne

Herunder ses klassediagrammet for nogle komponenter og containere.

Figur 11-11. Java



De hule pile repræsenterer *er-en*-relationer (dvs. nedarvning).

De andre pile repræsenterer *har*-relationer (dvs. at et objekt har en reference til et andet objekt, evt. 'ejer' objektet).

Bemærk, at Container selv arver fra Component, så en Container kan i sig selv bruges som en komponent. Det er relevant for Panel og ScrollPane, der er beregnet til at blive placeret inden i andre containere.

11.6. Layout-managere

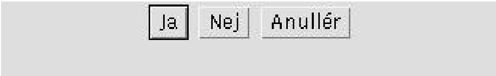
En *layout manager* styrer layoutet af komponenterne på et Panel eller en anden container. Alle containere har egenskaben *layout*, der kan sættes med metoden setLayout(Layout l).

Bruges et grafisk udviklingsværktøj, er det mest bekvemt at sætte layoutmanageren til null, der 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. Layoutet tager slet ikke højde for vinduets størrelse, så hvis vinduet bliver for lille, vil nogle af komponenterne ikke blive vist.

11.6.1. FlowLayout

FlowLayout placerer komponenterne ligesom bogstaver: Øverst fra venstre mod højre og på en ny linje nedenunder, når der ikke er mere plads.

Figur 11-12. Java

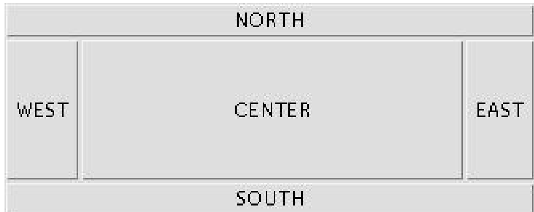


Angiver man ikke nogen anden layout-manager, vil FlowLayout blive brugt.

11.6.2. 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.

Figur 11-13. Java



Den mest almindelige måde at lave det grafiske layout af et skærmbillede er med BorderLayout. I de områder, hvor man ønsker at placere flere komponenter, sætter man først et Panel, og komponenterne tilføjes så panelet.

11.6.3. GridBagLayout

GridBagLayout lægger komponenterne efter et usynligt gitter.

Figur 11-14. Java

11.7. Opgave: Matadorspillet grafisk

Denne opgave kræver, at du har læst Kapitel 6, Nedarvning og Kapitel 10, Appletter og grafik.

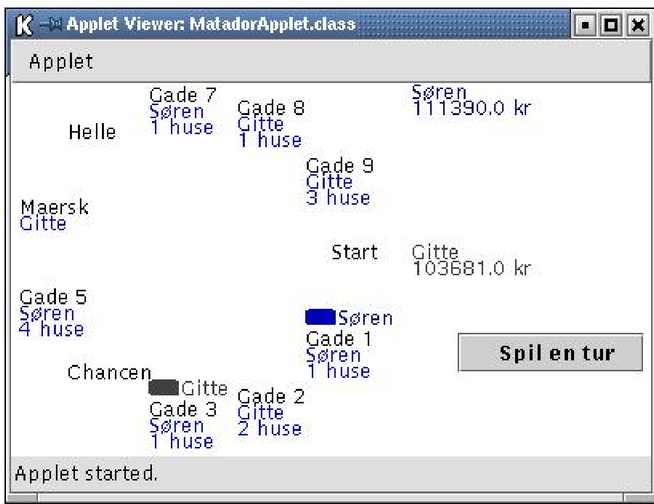
Lav matadorspillet om til at kunne vises grafisk i en applet. Der skal som minimum være en knap, som spiller en omgang.

11.7.1. Vink

Når du skal programmere, så vær systematisk, og del opgaven op i små delopgaver. Løs en delopgave ad gangen, og afprøv, at det fungerer, før du går videre.

1. Hent kildeteksten til matadorspillet (version 2: Felt.java, Gade2.java, Grund2.java, Helle.java, Rederi2.java, Start.java, Spiller.java og SpilMatador.java ændret til at bruge Gade2 og Rederi2), og prøv det.
2. Genbrug MinApplet ovenfor. Husk at initialisering skal ske i init()-metoden eller i konstruktøren. De variabler der skal deles mellem flere metoder, skal være objektvariabler (lokale eksisterer jo kun i den metode de er defineret i).
3. Lav en tur-knap, som spiller en runde.
4. Føj en metode til Felt, der tegner feltet. Hvert felt skal også have en position (den er en del af initialiseringen, så sæt den fra init()-metoden).
5. Løb igennem alle felter, og tegn dem i paint().
6. Udbyg derefter spillet efter egen smag.

Figur 11-15. Et grafisk matadorspil



11.7.2. Flere vink

Det er bedst at du bruger hovedet og kun ser på dem hvis du er gået i stå.

1. Prøve programmetHar du ikke allerede i sidste lektion prøvet matadorspillet, så prøv at køre programmet. Derefter 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 i en applet.
2. Struktur i en appletOpret en applet, eller genbrug MinApplet ovenfor. Flyt initialiseringen fra SpilMatador.java til init()-metoden eller konstruktøren. Husk at importere java.util.* øverst for at få adgang til Vector-klassen. Variablerne felter, sp1, sp2 skal nu være objektvariabler (før var de lokale variabler), for at de kan ses i resten af appletten:

```
import java.awt.*;
import java.applet.*;
import java.util.*;
public class MatadorApplet extends Applet
{
  // objektvariabler:
  Spiller sp1=new Spiller("Søren",50000,Color.green);    // opret spiller 1
  Spiller sp2=new Spiller("Gitte",50000,Color.yellow);   // opret spiller 2
  Vector felter=new Vector();
  public MatadorApplet()                                 // eller "public void init()"
  {
    felter.addElement(new Start(5000));
    felter.addElement(new Gade2("Gade 1",10000, 400,1000));
```

```
    felter.addElement(new Gade2("Gade 2",10000, 400,1000));
    //... osv.
```

Husk, at appletten først tegner noget, når initialiseringen er færdig, så hvis du f.eks. kører 20 runder i initialiseringen, tager det lang tid, førend systemet når til at kalde paint()!

1. Definér en tur-knapFor at få spillet til at køre kan du lave en knap. Når brugeren trykker på knappen, så kald spillernes tur()-metode. (Alternativ: kald spillernes tur()-metode inde i paint() og afslut paint() med: repaint(1000); dette får systemet til at kalde paint() igen efter et sekund).
2. Hvert felt skal have en position. Føj en position (af typen Point) til Felt-klassen:

```
import java.awt.*;
public class Felt
{
  String navn;
  Point position = new Point();
```

og definér metoden tegn(Graphics g) på Felt, der tegner feltets navn på positionen:

```
  public void tegn(Graphics g)
  {
    g.setColor(Color.black);
    g.drawString(navn,position.x,position.y);
  }
```

Husk at importere java.awt.* øverst for at få adgang til Point- og Graphics-klassen.

Løb alle felterne igennem i init(), og sæt koordinaterne på felterne:

```
    felter.addElement(new Gade2("Gade 8",20000,1100,2000));
    felter.addElement(new Gade2("Gade 9",30000,1500,2200));
    for (int i=0; i<felter.size(); i++)
    {
      double v = Math.PI*2*i/felter.size(); // vinkel i radianer
      Felt f = (Felt) felter.elementAt(i);
      f.position = new Point(
        100 + (int) (100*Math.cos(v)),
        110 + (int) (100*Math.sin(v))
      );
    }
```

1. Definér applettens paint()-metode til at kalde felternes tegn() for at tegne brættet:

```
  public void paint(Graphics g)
  {
    for (int i=0; i<felter.size(); i++)
    {
      Felt f = (Felt) felter.elementAt(i);
```

```
    f.tegn(g);
  }
```

En grund skal også have tegnet ejeren nedenunder, så den skal have en anderledes tegn(). Definér tegn() i Grund2. En gade skal også vise antallet af huse. Definér også tegn() i Gade2.

1. Find selv på flere ting:

- Lav tekstfelter, der beskriver hver spillers beholdning.
- Tegn spillernes biler på skærmen .
- Automatisk spil (vink: kald spillernes tur()-metode inde i paint(). Start paint() med: repaint(1000); dette får systemet til at kalde paint() igen efter et sekund).

Kapitel 12. Interfaces - grænseflader til objekter

Forudsættes af Kapitel 13, Hændelser, Kapitel 17, Flertrådet programmering, Kapitel 18, Serialisering, Kapitel 19, RMI og Kapitel 22, Indre klasser.

Forudsætter Kapitel 6, Nedarvning (og Kapitel 10, Appletter og grafik for at forstå et eksempel).

I generel sprogbrug er et interface (da.: grænseflade, snitflade) en form for grænseflade, som man gør noget gennem. F.eks. er en grafisk brugergrænseflade (eng.: Graphical User Interface - GUI) 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 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 interfacets metoder sammen med programkoden, der beskriver, hvad der skal ske, når metoderne kaldes.

12.1. Definere et interface

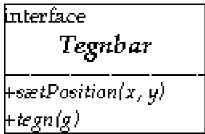
Lad os definere et interface kaldet Tegnbar, der beskriver nogle metoder på objekter, der kan tegnes.

```
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 (dvs. definerer kroppene).

Figur 12-1. Java



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 interfacet og give dem en metodekrop

Vi skal altså definere alle interfacets metoder sammen med programkoden, der beskriver hvad der skal ske, når metoderne kaldes.

```
import java.awt.*;

public class Stjerne implements Tegnbar
{
  private int posX, posY;

  public void sætPosition(int x, int y)  // kræves af Tegnbar
  {
    posX = x;
    posY = y;
  }

  public void tegn(Graphics g)          // kræves af 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. 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 Tegnbare objekter.

Man kan ikke oprette objekter ud fra et interface (der bare er en "skal" og intet siger om, hvordan metoderne er implementerede så hvordan skulle objektet reagere, hvis metoderne blev kaldt?).

```
    t = new Tegnbar();       // FEJL! Tegnbar er ikke en klasse
```

12.3. Eksempler med interfacet Tegnbar

Lad os udvide (arve fra) Terning til at implementere Tegnbar-interfacet. For at gøre koden kort har metoden tegn() en hjælpemetode ci(), der tegner en cirkel for et øje.

```
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+1+10*i,y+1+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,1,1); ci(g,2,0); ci(g,2,2); }
    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); }
  }
}
```

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. Her er et tegnbart rektangel:

```
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);
  }
}
```

12.3.1. En applet af Tegnbare objekter

Lad os nu lave en applet, der viser nogle tegnbare objekter:

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

public class TegnbareObjekter extends Applet
{
  Vector tegnbare = new Vector();
  GrafiskRaflebaeger bæger = new GrafiskRaflebaeger();

  public void paint(Graphics g)
  {
    super.paint(g);
    for (int n=0; n<tegnbare.size(); n++) {
      Tegnbar t = (Tegnbar) tegnbare.elementAt(n);
      t.tegn(g);
    }
  }

  public void sætPositioner()
  {
    for (int n=0; n<tegnbare.size(); n++) {
      Tegnbar t = (Tegnbar) tegnbare.elementAt(n);
      int x = (int) (Math.random()*200);
      int y = (int) (Math.random()*200);
      t.sætPosition(x,y);
    }
  }
```

```
  public void init() {
    Stjerne s = new Stjerne();
    tegnbare.addElement(s);

    tegnbare.addElement( new Rektangel(10,10,30,30) );

    tegnbare.addElement( new Rektangel(15,15,20,20) );

    GrafiskTerning t;

    t = new GrafiskTerning();
    bæger.tilføj(t);
    tegnbare.addElement(t);

    t = new GrafiskTerning();
    bæger.tilføj(t);
    tegnbare.addElement(t);

    tegnbare.addElement(bæger);

    sætPositioner();

    // mere kode her
    // ...
  }

  // flere metoder her
  // ...
}
```

Programmet holder styr på objekterne i tegnbare-vektoren. Da stjerner, rektangler, terningerne og raflebægeret alle er Tegnbare kan de behandles ens hvad angår tegning og positionering.

12.4. 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 vektor igennem og arbejde på objekterne i den, selvom de er af vidt forskellig type.

Dette så vi i TegnbareObjekter-appletten:

```
for (int n=0; n<tegnbare.size(); n++)
{
  Tegnbar t = (Tegnbar) tegnbare.elementAt(n);
  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.

12.5. 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 systemet på forskellig måde.

12.5.1. Sortering med Comparable

Hvis et objekt implementerer Comparable-interfacet skal det definere metoden:

```
  public int compareTo(Object obj)
```

For eksempel:

```
public class Element implements Comparable
{
  int x;

  public Element(int x1)
  {
    x = x1;
  }

  public String toString()
  {
    return "element"+x;
  }

  public int compareTo(Object obj)      // kræves af Comparable
  {
    Element andetElement = (Element) obj;// typekonverter først til Element

    if (x == andetElement.x) return 0;  // dette elem. og obj har samme plads
    if (x  > andetElement.x) return 1;  // dette element kommer efter obj
    else return -1;                // dette element kommer før obj
  }
```

```
}
```

Interfacet giver standardbibliotekerne mulighed for at sammenligne objekter og sortere dem i forhold til hinanden.

Sortering kan bl.a. ske ved at kalde metoden Collections.sort() med en vektor af objekter, der implementerer Comparable.

```
import java.util.*;
public class BrugElementer
{
  public static void main(String args[])
  {
    Vector liste = new Vector();
    liste.addElement( new Element(5));
    liste.addElement( new Element(3));
    liste.addElement( new Element(13));
    liste.addElement( new Element(1));

    System.out.println("før: "+liste);
    Collections.sort(v);
    System.out.println("efter: "+liste);
  }
}
```

Resultatet bliver:

```
før: [element5, element3, element13, element1]
efter: [element1, element3, element5, element13]
```

sort() vil kalde compareTo() på vores objekter for at ordne dem i rækkefølge. Havde vores objekter ikke implementeret Comparable, ville der opstå en køretidsfejl, da systemet så ikke havde nogen grænseflade, hvorigennem det kunne undersøge, hvordan elementerne skal ordnes.

12.5.2. 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 trådobjekt 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.

12.5.3. 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.

Lytteren skal registreres i en grafisk komponent, f.eks. en knap eller en applet. 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 11 om grafiske brugergrænseflader og hændelser er disse ting beskrevet nærmere.

12.6. Opgaver

1. Lav klassen Hus, der skal implementere Tegnbar, føj den til TegnbareObjekter og prøv om det virker.

Kapitel 13. Hændelser i grafiske brugergrænseflader

Indhold:

- Forstå hændelser og lyttere
- Abonnere på hændelser

Forudsættes af Kapitel 22, Indre klasser.

Forudsætter Kapitel 11, Grafiske brugergrænseflader og Kapitel 12, Interfaces.

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 hændelsestypen på den pågældende grafiske komponent.

En lytter er et objekt, der kan "abonnere" på en bestemt type hændelse. Når en lytter er registreret hos en grafisk komponent, bliver der kaldt en metode på lytter-objektet, når hændelsen 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.

F.eks.:

Appletter kan udsende hændelser af typen MouseEvent. Appletter har derfor metoden addMouseListener(MouseListener lytter), der kan bruges til at registrere lytter-objekter hos appletten. 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 klikker med musen i appletten, udsender appletten en MouseEvent-hændelse til alle lytter-objekter, der er blevet registreret vha. addMouseListener(). Det gør appletten ved at kalde metoden mouseClicked(MouseEvent hændelse) på lytter-objekterne.

13.1. Eksempel - LytTilMusen

Herunder definerer vi klassen Muselytter, der implementerer MouseListener. Hver gang der sker noget med musen, skrives det ud til skærmen.

MouseListener-interfacet har bl.a. metoden mousePressed, der kaldes, når musen trykkes ned. Parameteren er et MouseEvent-objekt, der bl.a. kan fortælle, hvor musen er, og hvilken knap der blev trykket på.

```
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 en lille applet, der:

1. Opretter et muselytter-objekt.
2. Registrerer lytter-objektet, så det får kaldt sine metoder, når der sker noget med musen.

```
import java.applet.*;
public class LytTilMusen extends Applet
{
  public void init()
  {
    Muselytter lytter = new Muselytter();
    this.addMouseListener(lytter);  // this er objektet selv
```

```
  }
}
```

Uddata fra appletten kan ses i konsolvinduet (i Netscape: Communicator/Tools/Java Console):

```
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 appletten besked om, at der er sket en hændelse. Det har man brug for, hvis man f.eks. vil tegne noget i appletten.

Herunder er et eksempel, hvor lytter-objektet (Linjelytter) giver informationer om klik videre til appletten (Linjetegning), sådan at en grøn linje tegnes mellem det punkt, hvor man trykkede museknappen ind, og det punkt, hvor man slap museknappen. Lytteren giver appletten besked vha. applettens to public variabler trykpunkt og slippunkt.

Lad os først kigge på appletten:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Linjetegning extends Applet
{
  public Point trykpunkt;
  public Point slippunkt;

  public void init()
  {
    Linjelytter lytter = new Linjelytter();
    lytter.appletten = this; // initialiserer lytterens reference til appletten
    this.addMouseListener(lytter);
  }

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

```
  }
}
```

Linjelytter er nødt til at have en reference til Linjetegning-appletten:

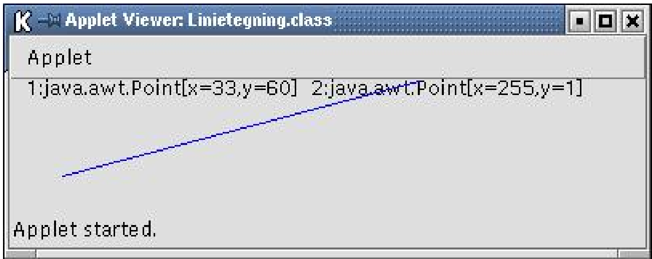
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Linjelytter implements MouseListener
{
  public Linjetegning appletten;                  // Reference til appletten

  public void mousePressed(MouseEvent hændelse)   // kræves af MouseListener
  {
    appletten.trykpunkt = hændelse.getPoint();
  }

  public void mouseReleased(MouseEvent hændelse) // kræves af MouseListener
  {
    appletten.slippunkt = hændelse.getPoint();
    appletten.repaint(); // Gentegn appletten 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

```
    appletten.repaint();
```

fortæller vi Linjetegning-appletten, at den skal gentegne sig selv. Det forårsager kort efter et kald til dens paint()-metode.

Figur 13-1. Java



13.2.1. Linjetegning i én klasse

Herunder er Linjetegning igen, men nu som en applet, der *selv* implementerer MouseListener.

Det er linjen

```
this.addMouseListener(this);
```

der registrerer applet-objektet selv som lytter.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Linjetegning2 extends Applet implements MouseListener
{
  private Point trykpunkt;
  private Point slippunkt;

  public void init()
  {
    this.addMouseListener(this);
  }

  public void paint(Graphics g)
  {
    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 vores trykpunkt og slippunkt-variabler 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.

Her er en applet til at tegne kruseduller. Vi husker punktet, når musen trykkes ned (mousePressed()), og tegner en linje fra forrige punkt til musen, når den trækkes med nedtrykket knap (mouseDragged()).

Tegningen af grafikken sker direkte i håndteringen af hændelsen.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Kruseduller extends Applet
                         implements MouseListener, MouseMotionListener
{
  public void init()
  {
    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
}
```

Da vi ikke husker de gamle punkter, kan vi ikke gentegne krusedullen, når systemet kalder paint().

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.applet.*;
import java.awt.*;
import java.awt.event.*;

public class LytTilKnap extends Applet implements ActionListener
{
  private TextArea t1, t2;
  private Button kopierKnap;

  public void init()
  {
    String s = "Her er en tekst.\nMarkér noget af den og tryk Kopier...";
    t1 = new TextArea(s, 5,20);
    add(t1);
    kopierKnap = new Button("Kopiér>>");
    kopierKnap.addActionListener(this);
    add(kopierKnap);
    t2 = new TextArea( 5,20);
    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 applet-objektet selv) hos knappen.

13.4. Lyttere og deres metoder

Det følgende er en oversigt over lytter-interfaces og deres hændelser.

13.4.1. 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);
}
```

13.4.2. ComponentListener

Sendes af alle grafiske komponenter (Button, TextField, Checkbox osv., og Frame, Applet, Panel,...), 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);
}
```

13.4.3. FocusListener

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.

```
public interface FocusListener {
    public void focusGained(FocusEvent e);
    public void focusLost(FocusEvent e);
}
```

13.4.4. ItemListener

Sendes af afkrydsningsfelter og radioknapper, når en mulighed bliver krydset af eller fravalgt.

```
public interface ItemListener {
    void itemStateChanged(ItemEvent e);
}
```

13.4.5. KeyListener

Sendes af komponenten, der har fokus. 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);
}
```

13.4.6. MouseListener

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);
}
```

13.4.7. 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);
}
```

13.4.8. TextListener

Sendes af tekstfelter (TextField og TextArea), når brugeren ændrer teksten.

```
public interface TextListener {
    public void textValueChanged(TextEvent e);
}
```

13.4.9. WindowListener

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);
}
```

Kapitel 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 vigtigt, når man skal finde fejl i sit program.

Forudsætter Kapitel 5, Definition af klasser (Kapitel 6, 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 vektors grænser:

```
import java.util.*;

public class SimpelUndtagelse
{
  public static void main(String[] args)
  {
    System.out.println("Punkt A");       // punkt A
    Vector v = new Vector();
    System.out.println("Punkt B");       // punkt B
    v.elementAt(5);
    System.out.println("Punkt C");       // punkt C
  }
}
```

og

```
Punkt A
Punkt B
java.lang.ArrayIndexOutOfBoundsException: 5 >= 0
  at java.util.Vector.elementAt(Vector.java:417)
  at SimpelUndtagelse.main(SimpelUndtagelse.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 elementAt(5) på den tomme Vector 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 vektoren, 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):

```
Vector v = null;
v.addElement("x");
```

Resultatet bliver:

```
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);
```

Resultatet bliver:

```
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;
```

Resultatet bliver:

```
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) // Undtagelsestype er 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.

Man kan fange alle slags ved at angive Exception eller en bestemt slags undtagelser, f.eks. ArrayIndexOutOfBoundsException.

Ser vi på vores Vector-eksempel igen, kunne det med undtagelseshåndtering se sådan ud:

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

```
public class SimpelUndtagelse2
{
  public static void main(String[] args)
  {
    System.out.println("Punkt A");               // pkt. A
    try
    {
      Vector v = new Vector();
      System.out.println("Punkt B");           // pkt. B
      v.elementAt(5);
      System.out.println("Punkt C");           // pkt. C
    }
    catch (Exception u)
    {
      System.out.println("Der opstod en undtagelse!");
    }
    System.out.println("Punkt D");             // pkt. D
  }
}
```

Resultatet bliver

```
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, og en reference til dette 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();
    }
    ...
```

Resultatet bliver

```
Punkt A
Punkt B
Der opstod en undtagelse!
java.lang.ArrayIndexOutOfBoundsException: 5 >= 0
        at java.util.Vector.elementAt(Vector.java:441)
        at SimpelUndtagelse2.main(SimpelUndtagelse2.java:11)
Punkt D
```

Staksporet er nyttigt, når man skal finde ud af, hvordan fejlen opstod. Det viser præcist, at undtagelsen opstod i elementAt() i Vector, som blev kaldt fra SimpelUndtagelse2.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 en IOException-undtagelse. Oversætteren tvinger os til at tage højde for den mulige undtagelse:

```
TastaturbrugerFejl.java:8: unreported exception java.io.IOException; must be caught or decl
              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 tvunget til enten at *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();
    }
```

eller *erklære, at den bliver kastet*, dvs. at den kan opstå i main(). Det gør man med ordet *throws*:

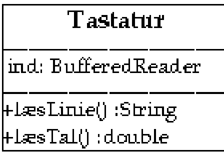
```
  public static void main(String arg[]) throws IOException
```

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

Figur 14-1. Java



Det har konsekvenser at sende undtagelser videre, for da skal kalderen håndtere dem. Her er et eksempel: Lad os sige, at vi har uddelegeret læsningen fra tastaturet til en separat klasse, der kan læse en linje fra tastaturet med læsLinje() og evt. omsætte den til et tal med læsTal():

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

```
public class Tastatur
{
  BufferedReader ind;

  public Tastatur()
  {
    ind = new BufferedReader(new InputStreamReader(System.in));
  }

  public String læsLinje()
  {
    try {
      String linje = ind.readLine();
      return linje;
    } catch (IOException u)
    {
      u.printStackTrace();
    }
    return null;
  }

  public double læsTal()
  {
    String linje = læsLinje();
    return Double.parseDouble(linje);
  }
}
```

Herover fanger vi undtagelsen IOException ved dens "rod" i læsLinje().

Den kunne gøres simplere 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
{
  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 følgefejl, 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 tallene ind. Til sidst spørger vi, om han vil prøve igen.

```
public class SumMedTastatur
{
  public static void main(String arg[])
  {
    Tastatur t = new Tastatur();
    boolean stop = false;
    try
    {
      while (!stop)
      {
        System.out.print("Hvor mange tal vil du lægge sammen? ");
        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);
        System.out.print("Vil du prøve igen? (j/n)");
        if ("n".equals(t.læsLinje())) stop = true;    // undersøg om det er "n"
      }
    }   catch (Exception u) {
      System.out.println("Der opstod en undtagelse!");
      u.printStackTrace();
    }
  }
}
```

Resultatet bliver:

```
Hvor mange tal vil du lægge sammen? 2
Indtast tal: 1
Indtast tal: 2
Summen er: 3.0
Vil du prøve igen? 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? 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.

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.

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;
```

```
    }
  }
}
```

Resultatet bliver:

```
Hvor mange tal vil du lægge sammen? 5
Indtast tal: 1
Indtast tal: x2z
Indtastningsfejl - java.lang.NumberFormatException: x2z
Vil du prøve igen? 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? 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. At 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");
  u1.printStackTrace();
}
catch (IOException u2)
{
  System.out.println("Inddata kunne ikke læses:"+u2);
```

```
}
catch (NullPointerException u3)
{
  System.out.println("Intern fejl i programmet:");
  u3.printStackTrace();
}
```

Alle undtagelses-klasser arver fra Exception, og man kan også fange *enhver* undtagelse, ved at fange deres fælles superklasse.

Fejlhåndteringen bliver så generel, ligegyldigt hvilken type undtagelse der opstod

```
try
{
  ...
  ...
}
catch (Exception u)
{
  System.out.println("Fejl:");
  u.printStackTrace();
}
```

14.6. Opgaver

1. Flyt try og catch i SumMedTastatur2 sådan, at programmet smider den aktuelle sum væk og prøver igen, men uden at spørge brugeren.
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. Flyt try og catch sådan, at programmet smider den aktuelle indtastning væk, men lader brugeren fortsætte med at regne på den samme sum.

Kapitel 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 af Kapitel 16, Netværkskommunikation og Kapitel 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 a 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. 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 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:

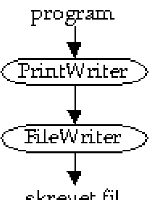
```
FileWriter fil = new FileWriter("tekstfil.txt");
```

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").

Figur 15-1. Java



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.

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[] args) 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.");
  }
}
```

Resultatet bliver:

```
Filen er gemt.
```

Eventuelle IO-undtagelser (f.eks. ikke mere plads på disken) tager vi os ikke af, men sender dem videre til styresystemet.

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();
```

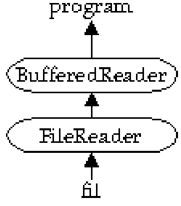
Efter at programmet har kørt, findes filen "skrevet fil.txt" på disken, med indhold:

```
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.

Figur 15-2. Java




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

public class LaesTekstfil
{
  public static void main(String[] args) 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. Indlæsning fra tastatur

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 en InputStreamReader, der konverterer til tegnbaseret (tekst)indlæsning

```
InputStreamReader tegnlæser = new InputStreamReader(System.in);
```

... og derpå gøre den linjeorienteret ved at pakke den yderligere ind i et BufferedReader-objekt:

```
BufferedReader ind = new BufferedReader( tegnlæser );
```

Derefter kan man læse inddata fra tastaturet linje for linje, ligesom vi gør med filer:

```
String linje = ind.readline();
```

15.4. Analysering af tekstdata

Ofte er det ikke nok bare at indlæse data, de skal også kunne behandles bagefter.

Det kunne være sjovt at udregne aldersgennemsnit i LaesTekstfil.java. Det kræver, at vi først opdeler data for at finde kolonnen med aldrene og derefter konverterer dem til tal, der kan regnes på.

15.4.1. Opdele inddata (StringTokenizer)

Har man brug for at dele strenge op i mindre dele, kan det gøres med StringTokenizer-klassen, der deler en streng op i bidder (eng.: tokens) efter bestemte skilletegn. Normalt opdeles efter blanktegn, og strengen bliver derfor delt op i ord.

En StringTokenizer oprettes med den streng, der skal opdeles:

```
StringTokenizer strengbidder = new StringTokenizer("Hej kære venner!");
```

Herefter kan man med metoden nextToken() få bidderne frem en efter en.

Metoden hasMoreTokens() er sand, hvis der er flere bidder, og falsk, når vi er nået forbi sidste bid:

```
while (strengbidder.hasMoreTokens())
{
  String bid = strengbidder.nextToken();
  System.out.println("bid: "+bid);
}
...
```

Resultatet bliver:

```
bid: Hej
bid: kære
bid: venner!
```

Ønsker man at opdele efter andet end mellemrum, kan man angive det i StringTokenizer's konstruktør. Herunder opdeler vi en formel efter både "+" og "-". Den sidste parameter angiver, at vi godt vil have skilletegnene, dvs. + og -, ud som selvstændige bidder (i stedet for at de smides væk som ligegyldige):

```
import java.util.*;
public class LaesFormel
{
  public static void main(String[] args)
  {
    StringTokenizer bidder = new StringTokenizer("2*x*x +8*x -5", "+-", true);
    while (bidder.hasMoreTokens())
    {
      String bid = bidder.nextToken();
      bid = bid.trim();            // fjern mellemrum omkring hver bid

      if (bid.equals("+"))      System.out.print(" plus ");
      else if (bid.equals("-")) System.out.print(" minus ");
      else                      System.out.print("'" + bid + "'");
    }
    System.out.println();
  }
```

```
}
```

Resultatet bliver:

```
'2*x*x' plus '8*x' minus '5'
```

Bemærk, at vi kalder trim() på strengene for at fjerne eventuelle blanktegn omkring hver bid.

15.4.2. Konvertere til tal

For at omsætte en streng til et tal (int eller double) skal strengen analyseres (eng.: parse), dvs. undersøges for, om den indeholder et tal, og tallet, som kan være repræsenteret på mange måder, skal findes frem. Det har Integer- og Double-klasserne funktioner til, nemlig hhv. parseInt() og parseDouble().

De tager en streng og returnerer den ønskede type:

```
  int i = Integer.parseInt("542");
  double d = Double.parseDouble("3.14");
```

Eksponentiel notation (hvor 9.8E3 betyder 9800) forstås også, og der kan også bruges andre talsystemer end titalsystemet. F.eks. giver Integer.parseInt("00010011",2) tallet 19 (19 svarer til 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
```

15.4.3. DecimalFormat og DateFormat-klasserne

Klasserne DecimalFormat og DateFormat giver ikke blot mange muligheder for at formatere tal/datoer som strenge, men kan også analysere strenge for forskellige tal- og tidsformater og trække data ud af strenge. De er nærmere beskrevet i slutningen af Kapitel 4.

15.4.4. Samlet eksempel: Statistik

Nu kan vi skrive 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[] args)
  {
    int antalPersoner = 0;
    int sumAlder = 0;

    try
    {
      BufferedReader ind =
        new BufferedReader(new FileReader("skrevet fil.txt"));

      String linje = ind.readLine();
      while (linje != null)
      {
        try
        {
          StringTokenizer bidder = new StringTokenizer(linje);

          String navn = bidder.nextToken();
          String køn = bidder.nextToken();
          int alder = Integer.parseInt(bidder.nextToken());

          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();
        }
        linje = ind.readLine();
      }

      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();
    }
  }
}
```

Resultatet bliver:

```
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, når vi dividerer med antalPersoner, og "Fejl ved læsning af fil" udskrives.

Under analyseringen af linjen kan der også opstå flere forskellige slags undtagelser: Konverteringen til heltal kan gå galt, og der kan være for få bidder, så nextToken() bliver kaldt efter sidste bid.

For eksempel giver linjen "person2 m24" (der mangler et mellemrum mellem m og 24)

```
Fejl. Linjen springes over.
java.util.NoSuchElementException
        at java.util.StringTokenizer.nextToken(StringTokenizer.java:241)
        at LaesTekstfil.main(LaesTekstfil.java:25)
```

Hvis disse fejl opstår, fortsætter programmet efter catch-blokken med at læse næste linje af inddata.

Da sumAlder og antalPersoner ændres sidst i try-catch-blokken, vil de kun blive opdateret hvis hele linjen er i orden, og statistikken udregnes derfor kun på grundlag af de gyldige linjer.

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:

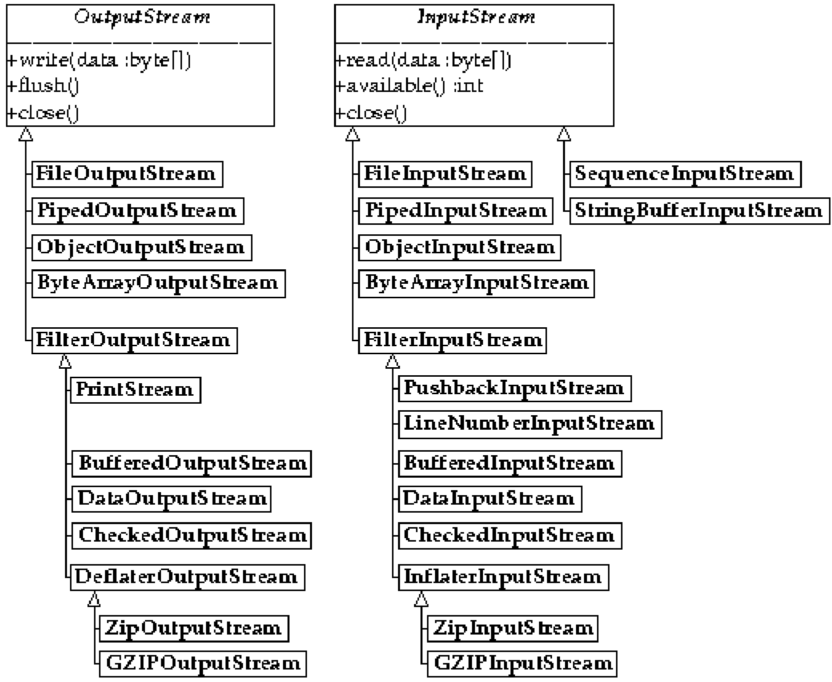
InputStream-objekter læser binære data. *OutputStream*-objekter skriver binære data.

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.

Figur 15-3. Java



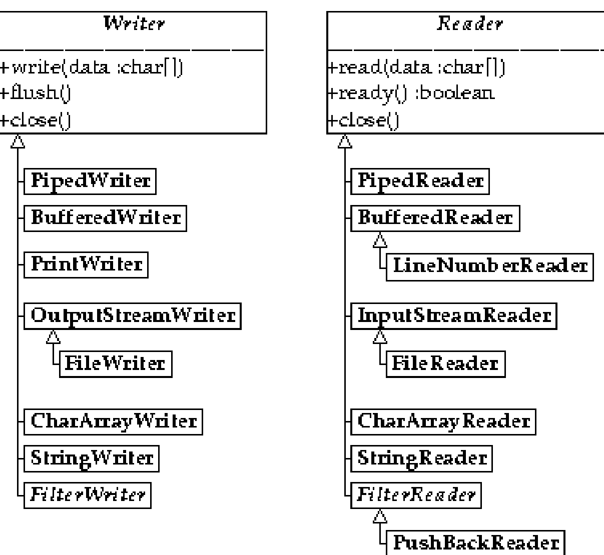
Af klassediagrammet ses, at metoderne i InputStream og OutputStream læser og skriver byte-data: write(byte[]) på OutputStream skriver et array (en række) af byte. Arvingerne har lignende metoder (disse er ikke vist).

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 og metoder 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 netværkskommunikation. Dette håndteres af klasserne, der nedarver fra Reader og Writer.

Figur 15-4. Java



Af klassediagrammet ses, at alle metoderne i Reader og Writer læser og skriver tegndata (datatype char). Tegn repræsenteres i Java som 16-bit unicode-værdier, og man kan derfor arbejde med ikke blot det vesteuropæiske tegnsæt, men også det østeuropæiske, kinesiske, russiske, ...

15.5.4. Fillæsning og -skrivning (File-)

Klasserne til filhåndtering er FileInputStream, FileReader, FileOutputStream og FileWriter.

15.5.5. Strenge (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 (sml. Afsnit 15.3, Indlæsning fra tastatur).

```
StringReader tegnlæser = new StringReader("Jacob\n4\n5.14\n");
BufferedReader ind = new BufferedReader( tegnlæser );
```

StringWriter er en datastrøm, der gemmer uddata i et StringBuffer-objekt, der kan konverteres til en streng.

15.5.6. Arrays (ByteArray- og CharArray-)

Et array er en liste eller række af noget (se Kapitel 9 om arrays). 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 fra et array af byte.

15.5.7. Læse/skrive objekter (Object-)

Det er muligt at skrive hele objekter ned i en datastrøm. Objektet 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 18Serialisering.

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 InputStream (tegn til byte).

15.5.10. Filtreringsklasser til konvertering og databehandling

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 tager to eller flere datakilder og læser dem 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 tjeksum 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 tjeksum-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.

Filtreringsklasser skydes ind som ekstra "indpakning" mellem de andre datastrømme. F.eks. kan

```
PrintWriter ud= new PrintWriter(new FileOutputStream("fil"));
```

ændres til også at komprimere uddata, simpelthen ved at skyde et GZIPOutputStream-objekt ind:

```
PrintWriter ud=new PrintWriter(GZIPOutputStream(new(FileOutputStream("fil.gz"));
```

15.6. Ekstra eksempler

Herunder læser vi en fil og udregner filens tjeksum og antallet af linjer i filen.

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

public class UndersoegFil
{
  public static void main(String[] args) throws IOException
  {
    FileInputStream fil = new FileInputStream("skrevet fil.txt");
    BufferedInputStream bstrøm = new BufferedInputStream(fil);
    CRC32 tjeksum = new CRC32();
    CheckedInputStream chkstrøm = new CheckedInputStream(bstrøm,tjeksum);
```

```
    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):" +tjeksum.getValue());
  }
}
```

Resultatet bliver

```
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 hægter datastrøm-objekterne sammen i en kæde ved hele tiden at bruge det forrige objekt i som parameter til konstruktørerne: Filindlæsning, buffering, tjeksum, 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).

15.7. 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. Skriv et program, 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(...))
3. Skriv et program, Diff.java, der sammenligner to tekstfiler linje for linje og udskriver de linjer, der er forskellige.
4. Ret SkrivTekstfil.java til SkrivKomprimeretTekstfil.java, der gemmer data komprimeret.
5. Lav den tilsvarende LaesKomprimeretTekstfilOgLavStatistik.java.
6. 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 150Peter 18 300 900 -950Lis 1000 13.5
   ```

Kapitel 16. Netværkskommunikation

Indhold:

- At koble til en tjeneste på en fjernmaskine
- At udbyde tjenester på netværket
- Eksempler: Hente en hjemmeside og en webserver

Kapitlet forudsættes ikke i resten af bogen.

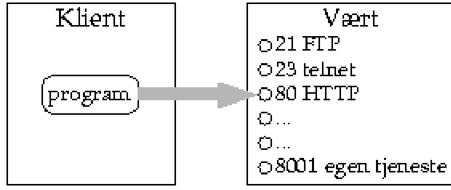
Forudsætter Kapitel 15, Datastrømme og filhåndtering.

Alle maskiner på et TCP-IP-netværk, f.eks. internettet, har et IP-nummer. Det er en talrække på fire byte, 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.

Figur 16-1. Java



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 sig til en port

Man opretter en forbindelse ved at oprette et Socket-objekt 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 fraOs = forbindelse.getOutputStream();
InputStream tilOs = forbindelse.getInputStream();
```

Hvis vi vil sende/modtage binære data, kan vi nu bare gå i gang: fraOs.write() sender en byte eller en række af byte til værten, og med tilOs.read() læser vi en eller flere byte.

Hvis det er tekstkommunikation, er PrintWriter og BufferedReader (der arbejder med tegn og strenge som beskrevet tidligere) dog nemmere at bruge :

```
PrintWriter ud = new PrintWriter(fraOs);
BufferedReader ind = new BufferedReader(new InputStreamReader(tilOs));
```

Nu kan vi f.eks. bede værten om at give os startsiden (svarende til adressen http://www.esperanto.dk/). Det gøres ved at sende linjen "GET / HTTP/0.9", derefter "Host: www.esperanto.dk" og til sidst en blank linje:

```
ud.println("GET / 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.

Nu sender værten svaret, der kan læses fra inddatastrømmen:

```
String s = ind.readLine();
while (s != null) {
  System.out.println("svar: "+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 fraOs = forbindelse.getOutputStream();
      InputStream  tilOs = forbindelse.getInputStream();
      PrintWriter    ud  = new PrintWriter(fraOs);
      BufferedReader ind = new BufferedReader(new InputStreamReader(tilOs));
      ud.println("GET / 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("svar: "+s);
        s = ind.readLine();
      }
      forbindelse.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Resultatet bliver:

```
svar: HTTP/1.1 200 OK
svar: Date: Tue, 17 Apr 2001 13:06:06 GMT
svar: Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/4.0.2 mod_perl/1.24
svar: Last-Modified: Thu, 05 Mar 1998 17:28:16 GMT
svar: Accept-Ranges: bytes
svar: Content-Length: 896
svar: Content-Type: text/html
svar:
svar: <HTML><HEAD><TITLE>Esperanto.dk</TITLE>
svar: <META name="description" content="Den officielle danske hjemmeside om plansproget esp
svar: <META name="keywords" content="Esperanto, Danmark, DEA, UFFE, Esperanto-nyt, bogsalg,
svar: </HEAD>
svar:
svar: <FRAMESET cols="22%,*"                  >
svar: <FRAME name="menu"    src="da/menu.htm"     marginwidth=0>
svar: <FRAME name="indhold" src="da/velkomst.htm">
```

```
svar: <NOFRAMES>
svar: Velkommen til Esperanto.dk!
svar: <p>
svar: Gå til <a href="da/velkomst.htm">velkomst-siden</a> eller til
svar: <a href="da/menu.htm">indholdsfortegnelsen</a>,
svar:
svar: </NOFRAMES>
svar: </FRAMESET>
svar:
svar: </html>
```

Det ses, at svaret starter med et hoved med metadata, der beskriver indholdet (dato, værtens 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.

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.

Nu 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 skal først skal spørge og værten 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 (afslutter samtalen):

```
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 soklen, ellers går svaret helt eller delvist tabt; forbindelse-objektet ved ikke, at ud-objektet 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 værtssokkel = new ServerSocket(8001);
      while (true)
      {
        Socket forb = værtssokkel.accept();
        PrintWriter ud = new PrintWriter(forb.getOutputStream());

        BufferedReader ind =
          new BufferedReader(new InputStreamReader(forb.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();
        forb.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Resultatet bliver:

```
Anmodning: GET / HTTP/0.9
Anmodning: GET / HTTP/1.1
Anmodning: GET /xxx.html HTTP/1.1
```

Du kan afprøve programmet ved at ændre på HentHjemmeside.java til at spørge 'localhost' på port 8001 eller ved i Netscape at åbne adressen http://localhost:8001/xxx.html

16.3. Opgaver

1. Læs Javadokumentationen for klassen URLConnection, og omskriv HentHjemmeside til at bruge denne klasse i stedet for selv af lave Socket-forbindelser.
2. Lav din egen proxy. En proxy er en "stråmand", der modtager en HTTP-forespørgsel og spørger videre for klienten.
3. Lav en virtuel opslagstavle. Den skal bestå af klasserne Opslagstavletjeneste, som udbyder tjenesten (brug port 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.

Kapitel 17. Flertrådet programmering

Indhold:

- Forstå tråde
- Eksempel på en flertrådet webserver

Kapitlet forudsættes ikke i resten af bogen.

Forudsætter Kapitel 12, Interfaces (Kapitel 16, Netværkskommunikation og Kapitel 10, Appletter og grafik 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 tekstbehandlingsprogram, hvor man ønsker at gemme eller sende en udskrift til printeren, mens man redigerer videre, eller man ønsker løbende stavekontrol samtidig med, at man skriver. Skrivningen må ikke blive forsinket af, at programmet sideløbende forbereder en udskrift eller kontrollerer stavningen. Disse delprocesser (også 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 flertrådet i Java. Man opretter en ny tråd med et objekt i konstruktøren:

```
  Thread tråd;
  tråd = new Thread(obj);
```

Objektet obj skal implementere Runnable-interfacet, f.eks.:

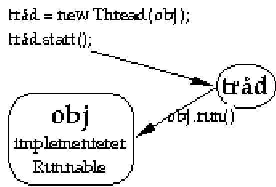
```
public class UdførbartObjekt implements Runnable
{
  public void run()  // kræves af Runnable
  {
    // her starter den nye tråd med at køre
    // ...
  }
}
```

Tråden er nu klar til at udføre run()-metoden på objektet, men den er ikke startet endnu. Man starter tråden ved at kalde start()-metoden på tråd-objektet:

```
tråd.start();
// her kører den oprindelige tråd videre, mens den nye kører i 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.

Figur 17-1. Java



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

```
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
    }
  }
}
```

Objekter af typen SnakkesaligPerson kan køre i en tråd (implements Runnable).

I konstruktøren angives navnet på personen og hvor lang tid, der går, mellem hver gang personen taler.

Når run() udføres, skriver den personens navn + bla bla bla ud så ofte som angivet.

Da Thread.sleep() kan kaste undtagelser af typen InterruptedException, er vi nødt til at indkapsle koden i en try-catch-blok (disse undtagelser 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);
    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);
    t = new Thread(p);
    t.start();

    // Det kan også gøres meget kompakt:
    new Thread(new SnakkesagligPerson("Henrik",200)).start();
  }
}
```

Resultatet bliver:

```
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

17.2.1. En flertrådet webserver

Herunder har vi lavet en flertrådet webserver (sammenlign med webserveren 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.io.*;
import java.net.*;
public class FlertraadetHjemmesidevaert
{
  public static void main(String arg[])
  {
    try {
      ServerSocket værtssokkel = new ServerSocket(8001);

      while (true)
      {
        Socket forbindelse = værtssokkel.accept();
        Anmodning a = new Anmodning(forbindelse);
        new Thread(a).start();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Resultatet bliver

```
start Tue Nov 28 15:37:31 GMT+01:00 2000 GET /xxx.html HTTP/1.0
start Tue Nov 28 15:37:38 GMT+01:00 2000 GET /yyy.html HTTP/1.0
start Tue Nov 28 15:37:42 GMT+01:00 2000 GET /zzz.html HTTP/1.0
slut  Tue Nov 28 15:37:42 GMT+01:00 2000 GET /xxx.html HTTP/1.0
slut  Tue Nov 28 15:37:49 GMT+01:00 2000 GET /yyy.html HTTP/1.0
start Tue Nov 28 15:37:50 GMT+01:00 2000 GET /qqq.html HTTP/1.0
```

```
slut  Tue Nov 28 15:37:53 GMT+01:00 2000 GET /zzz.html HTTP/1.0
slut  Tue Nov 28 15:38:01 GMT+01:00 2000 GET /qqq.html HTTP/1.0
```

Programmet er afprøvet ved i Netscape at åbne adressen http://localhost:8001/xxx.html hhv. yyy zzz og qqq.html. Man ser, at anmodningerne xxx, yyy og zzz bliver behandlet samtidigt.

17.2.2. En flertrådet applet med bolde

Lad os lave en applet 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 boldens run()-metode.

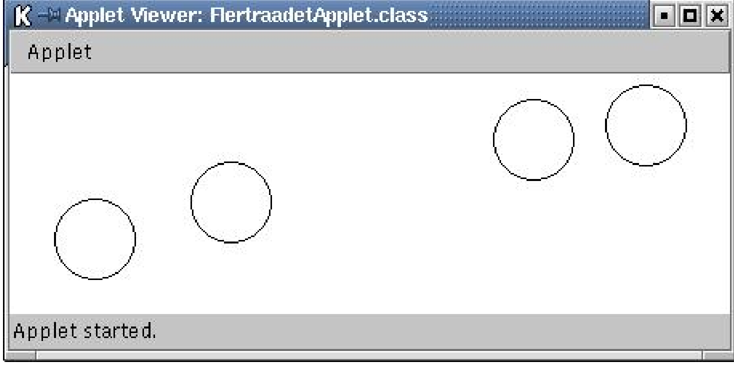
```
import java.awt.*;

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 fart 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) {};
      }
    }
  }
```

Lad os nu lave en applet med nogle bolde:

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

public class FlertraadetApplet extends Applet
{
  public void init()
  {
    setBackground(Color.white);
    Graphics g = getGraphics();
    new Bold(g,  0, 0);
    new Bold(g, 50,10);
    new Bold(g,100,50);
    new Bold(g,150,90);
  }
}
```

Figur 17-2. Java



17.3. Opgaver

1. Udvid FlertraadetApplet med andre figurer end bolde.
2. Skriv et program, der udregner primtal. 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.

Kapitel 18. Serialisering af objekter

Kapitlet forudsættes af Kapitel 19, RMI.

Forudsætter Kapitel 15, Datastrømme og filhåndtering.

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 [1]. 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 p = new ObjectOutputStream(datastrøm);
    p.writeObject(obj);
    p.close();
  }

  public static Object hent(String filnavn) throws Exception
  {
    FileInputStream datastrøm = new FileInputStream(filnavn);
    ObjectInputStream p = new ObjectInputStream(datastrøm);
    Object obj = p.readObject();
    p.close();
    return obj;
  }
}
```

Du kan benytte klassen fra dine egne programmer.

Her er et program, der læser en vektor fra filen [2] `venner.ser`, tilføjer en indgang og gemmer vektoren i filen igen.

```
import java.util.*;
public class HentOgGem
{
  public static void main(String arg[]) throws Exception
  {
    Vector v;
    try {
      v = (Vector) Serialisering.hent("venner.ser");
      System.out.println("Læst: "+v);
    } catch (Exception e) {
      v = new Vector();
      v.addElement("Jacob");
      v.addElement("Brian");
      v.addElement("Preben");
      System.out.println("Oprettet: "+v);
    }

    v.addElement("Ven"+v.size());
    Serialisering.gem(v,"venner.ser");
    System.out.println("Gemt: "+v);
  }
}
```

Resultatet bliver:

```
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 tilføjer "Jacob", "Brian" og "Preben" til vektoren. Derpå tilføjer vi "Ven3" og gemmer vektoren.

Næste gang er uddata:

```
Læst: [Jacob, Brian, Preben, Ven3]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4]
```

Køres programmet igen, ser man, at den hver gang tilføjer en indgang:

```
Læst: [Jacob, Brian, Preben, Ven3, Ven4]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4, Ven5]
```

og

```
Læst: [Jacob, Brian, Preben, Ven3, Ven4, Ven5]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4, Ven5, Ven6]
```

og

```
Læst: [Jacob, Brian, Preben, Ven3, Ven4, Ven5, Ven6]
Gemt: [Jacob, Brian, Preben, Ven3, Ven4, Ven5, Ven6, Ven7]
```

og

```
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 vektor, bliver elementerne i vektoren også serialiseret. Dette gælder også, hvis disse elementer selv indeholder eller er vektorer 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 forbindelse, der har været gemt på harddisken i tre uger? Den anden ende af netværksforbindelsen vil formentlig være væk på det tidspunkt (og filen kan være flyttet eller slettet).

18.2.1. Interfacet Serializable

Serializable-interfacet, der ingen metoder har defineret, bruges til at markere, at objekter *godt må* serialiseres. Hvis en klasse implementerer Serializable, har man fortalt Java at objekter af denne type godt kan serialiseres.

Prøver man alligevel at serialisere et objekt der ikke er Serializable, opstår der en køretidsfejl. Derfor implementerer f.eks. FileWriter og Socket ikke Serializable.

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. Eksempel

Eksemplet herunder viser en klasse, der kan serialiseres (implements Serializable), med en transient variabel (tmp). Hvis et Data-objekt serialiseres, vil a blive gemt i byte-sekvensen, 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

  public String toString()
  {
    return "Data: a="+a+" tmp="+tmp;
  }
}
```

Her er et program, der læser en vektor af Data-objekter, tilføjer et og gemmer den igen.

```
import java.util.*;
public class HentOgGemData
{
  public static void main(String arg[]) throws Exception
  {
    Vector v;
    try {
      v = (Vector) Serialisering.hent("data.ser");
      System.out.println("Indlæst: "+v);
    } catch (Exception e) {
      v = new Vector();
      System.out.println("Oprettet: "+v);
    }

    Data d = new Data();
    d.a   = (int) (Math.random()*100);
    d.tmp = (int) (Math.random()*100);
    v.addElement(d);

    System.out.println("Gemt: "+v);
    Serialisering.gem(v,"data.ser");
  }
}
```

Resultatet bliver:

```
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]
```

og

```
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]
```

og

```
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. 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, brugeren skal kunne tilføje flere, og gemme og hente listen i en fil (vha. serialisering).
2. Udvid programmet til, at brugeren angiver filnavnet, der skal hentes/gemmes i.

Slutbemærkning:

1. Eller netværket for den sags skyld.
2. Man bruger ofte filendelsen .ser til serialiserede objekter.

Kapitel 19. RMI - objekter over netværk

Kapitlet forudsættes ikke i resten af bogen.

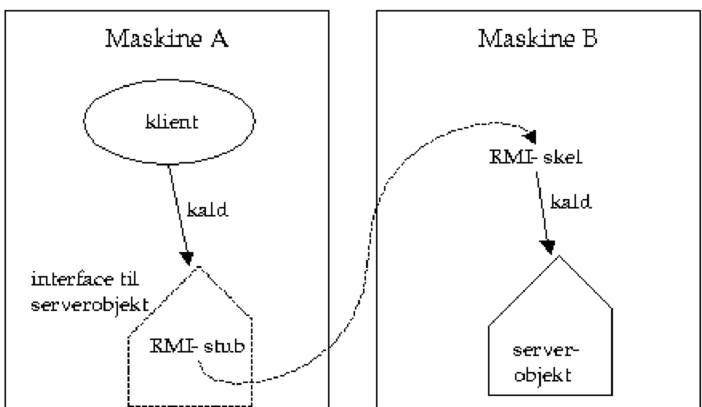
Forudsætter Kapitel 12, Interfaces, Kapitel 18, Serialisering og kendskab til netværk.

RMI (Remote Method Invocation) er en måde at 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 (da: værts-objekt), der er i maskine B.

Figur 19-1. Java



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.

Overordnet set foregår det på den måde, at der defineres et interface til de metoder på serverobjektet, der skal være tilgængelige for klienten.

19.2. Konkret

Konkret kunne man tænke sig, at serveren havde et konto-objekt, hvor man kan overføre penge, spørge om saldo og få bevægelserne på kontoen. Så ville man definere et Konto-interface (her kaldt KontoI):

```
import java.util.Vector;

public interface KontoI extends java.rmi.Remote
{
  public void overførsel(int kroner) throws java.rmi.RemoteException;
  public int saldo()                 throws java.rmi.RemoteException;
  public Vector bevægelser()         throws java.rmi.RemoteException;
}
```

Interfacet skal arve fra interfacet java.rmi.Remote, og alle metoder skal kunne kaste undtagelsen java.rmi.RemoteException.

Interfacet bliver brugt på både klientsiden og serversiden.

19.2.1. På serversiden

På serversiden skal vi implementere Konto-interfacet og programmere den funktionalitet der skjuler sig bag det, i et serverobjekt, som normalt ender på Impl (for at signalere, at det er implementationen af interfacet). Serverobjektet skal arve fra UnicastRemoteObject.

```
import java.util.Vector;
import java.rmi.server.UnicastRemoteObject;

public class KontoImpl extends UnicastRemoteObject implements KontoI
{
  public int saldo;
  public Vector bevægelser;

  public KontoImpl() throws java.rmi.RemoteException
  {
    // man starter med 100 kroner
    saldo = 100;
    bevægelser = new Vector();
  }
```

```
  public void overførsel(int kroner) throws java.rmi.RemoteException
  {
    saldo = saldo + kroner;
    String s = "Overførsel på "+kroner+" kr. Ny saldo er "+saldo+" kr.";
    bevægelser.addElement(s);
    System.out.println(s);
  }

  public int saldo()                 throws java.rmi.RemoteException
  {
    System.out.println("Der spørges om saldoen. Den er "+saldo+" kr.");
    return saldo;
  }

  public Vector bevægelser()         throws java.rmi.RemoteException
  {
    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. Det sker sådan her:

```
import java.rmi.Naming;
public class Kontoserver
{
  public static void main(String args[]) throws Exception
  {
    KontoI k = new KontoImpl();
    Naming.rebind("rmi://localhost/Kontotjeneste", k);
    System.out.println("Kontotjeneste registreret.");
  }
}
```

Derudover 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. Det skal kende definitionen af de klasser, der overføres.

Når vi skal køre vores server sker de i fire trin:

1. alle kildetekster oversættes til bytekode: `javac *.java` (eller i et udviklingsværktøj)
2. KontoImpl skal have en RMI-stub og en RMI-skel: `rmic KontoImpl.java`
3. rmiregistry startes i et separat vindue (fra samme katalog, som bytekoden ligger i): `rmiregistry`
4. til sidst kan Kontoserver startes fra et separat vindue: `java Kontoserver` (eller i et udviklingsværktøj)

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 en RMI-stub, der implementerer KontoI):

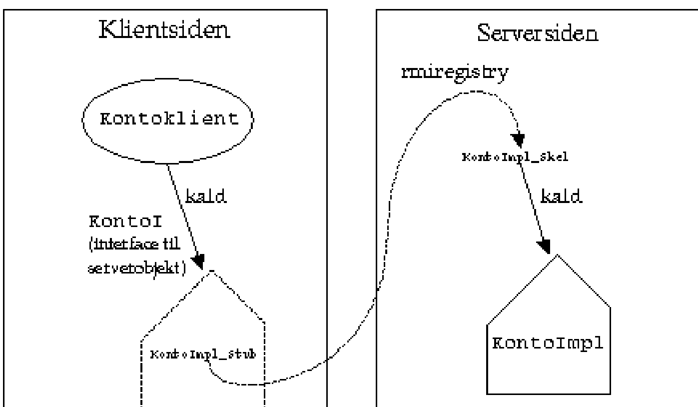
```
import java.util.Vector;
import java.rmi.Naming;

public class Kontoklient
{
  public static void main(String[] args)
  {
    try
    {
      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() );
      Vector bevægelser = k.bevægelser();

      System.out.println( "Bevægelser er: "+ bevægelser );
    }
    catch (Exception e)
    {
      System.out.println(e);
      e.printStackTrace();
    }
  }
}
```

Resultatet bliver:

```
Saldo er: 250
Saldo er: 101
Bevægelser er: [Overførsel på 100 kr. Ny saldo er 200 kr., Overførsel på 50 kr. Ny saldo er
```

Herunder ses de enkelte klassers funktioner.

Figur 19-2. Java



Kapitel 20. JDBC - databaseadgang

Kapitlet forudsættes ikke i resten af bogen.

Det forudsætter Kapitel 14 Undtagelser, og kendskab til databaser og 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 driveren
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. De nyeste drivere kan findes på http://www.javasoft.com (http://www.javasoft.com/) under JDBC.

For en Oracle-database hedder filen classes12.zip og passer til en bestemt udgave af Oracle-databasen. Bruger man Oracle Jdeveloper, er den som standard med i projektets klassesti. Ellers skal den føjes til CLASSPATH (i JBuilder gøres det 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", "stuk1001", "hemli’");
```

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");
```

Før du kan oprette forbindelsen, skal datakilden først være oprettet under ODBC i Windows' Kontrolpanel. Man angiver datakildens navn (her "datakilde1"), når forbindelsen oprettes, uden brugernavn og adgangskode:

```
Connection forb = DriverManager.getConnection ("jdbc:odbc:datakilde1");
```

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 som INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE og ALTER TABLE, der ikke giver et svar tilbage i form af data, sendes med executeUpdate()-metoden [1].

Her opretter vi f.eks. tabellen "kunder":

```
stmt.executeUpdate("create table KUNDER (NAVN varchar(32), KREDIT float)" );
```

... og indsætter et par rækker

```
stmt.executeUpdate("insert into KUNDER values('Jacob', -1799)");
```

```
stmt.executeUpdate("insert into KUNDER values('Brian', 0)");
```

Oftest har vi data i variabler, så vi skal sætte en streng sammen, der giver den rigtige SQL-kommando:

```
String navn = "Hans";
int kredit = 500;

// indsæt data fra variabler
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 returnerer et ResultSet-objekt, der repræsenterer svaret på forespørgslen. Data hentes som vist herunder

```
  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, hvorefter man går videre til næste række med next() osv. Når next() returnerer false, er der ikke flere rækker at læse.

20.3. Samlet eksempel

Det er en god idé at indkapsle databasekommunikationen 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 per tabel i databasen, sådan at et objekt kommer til at svare til en række [2]:

```
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.

```
import java.sql.*;
import java.util.*;

public class Databaseforbindelse
{
  private Connection forb;
  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","stuk1001","hemli’");
    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 Vector hentAlle() throws SQLException
  {
    Vector alle = new Vector();
    ResultSet rs = stmt.executeQuery("select NAVN, KREDIT from KUNDER");
    while (rs.next())
    {
      Kunde k = new Kunde( rs.getString("NAVN"), rs.getDouble("KREDIT"));
      alle.addElement(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
      Vector  v = dbf.hentAlle();
      System.out.println("Alle data: "+v);
      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();
    }
  }
}
```

Resultatet bliver:

```
Alle data: [Jacob: -1799.0 kr., Brian: 0.0 kr.]
Alle data nu: [Kurt: 1000.0 kr.]
```

20.4. Opgaver

1. 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).
2. Udvid Databaseforbindelse, så den kan give en liste med alle kunder med negativ kredit.
3. 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.
4. 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?
5. Udvid programmet, så hver gruppe har en genre som f.eks. rock, tekno, klassisk (tabellen GRUPPER udvides med genreID, og tabellen GENRER med kolonnerne genreID og navn oprettes).

Slutbemærkning:

1. SQL fra JDBC er normalt forpligtigende, d.v.s. en kommando kan ikke fortrydes når den først er givet. I Avancered-afsnittet i slutningen af kapitlet vises hvordan automatisk forpligtigelse (eng.: auto-commit) kan slås fra.
2. Det kommer lidt an på, i hvor høj grad basen er normaliseret.

Kapitel 21. Avancerede klasser

Dette kapitels afsnit om final på variabler forudsættes af Kapitel 22, Indre klasser.

Forudsætter Kapitel 6, Nedarvning og Kapitel 7, Pakker og ønske om at vide mere om emner som først bliver relevante når man laver større programmer.

21.1. public, protected og private

Det er vigtigt at styre adgangen til at kalde metoder og ændre på variabler, i særdeleshed når programmerne bliver store. Det kan lette overskueligheden meget hvis interne variabler, metoder og klasser er skjult for programmøren.

Adgang styres med nøgleordene public, protected og private. Adgangskontrol ud over public/private bliver først interessant når der er mange klasser og programmet spænder over flere pakker. Se eventuelt kapitlet om pakker.

21.1.1. Variabler og metoder

Variabler og metoder erklæret *public* er altid tilgængelige, inden 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:

Tabel 21-1. Adgangsregler

Adgang	public	protected	(ingenting)	private
i samme klasse	ja	ja	ja	ja
klasse i samme pakke	ja	ja	ja	nej
arving i en anden pakke	ja	ja	nej	nej

Adgang	public	protected	(ingenting)	private
ej arving og i en anden pakke	ja	nej	nej	nej

Holder man sig inden for samme pakke, er der altså ingen forskel mellem public, protected og ingenting.

21.1.2. 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 have flere klasser i en fil, men højst en, der er public, og denne klasse skal hedde det samme som filnavnet.

21.2. Nøgleordet final

Noget, der er erklæret *final*, kan ikke ændres. Både variabler, metoder og klasser kan erklæres final.

21.2.1. Variabler

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.

final foran en objektvariabel angiver ikke 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 args[])
  {
    final Vector v = new Vector();

    //v = new Vector();  // ulovligt! v 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:

```
    v.addElement("Hans");  // lovligt, v refererer stadig til samme objekt
```

21.2.2. Metoder

En metode erklæret final kan ikke tilsidesættes i en nedarving.

```
public class X
{
  public final void a()
  {
    // ..
  }
}
```

og

```
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

En klasse erklæret final må man overhovedet ikke arve fra (og alle dens metoder bliver final).

```
public final class X
{
  // ..
}
```

og

```
public class Y extends X  // ulovligt! X er final
{
}
```

21.3. Nøgleordet abstract

Noget der er erklæret *abstract* er ikke implementeret og skal defineres i en nedarvning.

Det skrives i kursiv i UML-notationen.

21.3.1. 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 args[])
  {
    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 args[])
  {
    X x;            // lovligt

    x = new Y();    // lovligt, Y er ikke abstrakt
```

Basisklasserne for IO-systemet, InputStream og OutputStream, abstrakte, fordi programmøren altid skal bruge en mere konkret klasse, f.eks. FileInputStream (se Afsnit 15.5.4).

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).

21.3.2. 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()
  {
    //..
  }
}
```

Kapitel 22. Indre klasser

Indhold:

- 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

Kapitlet er en forudsætning for at forstå den måde, mange værktøjer laver kode til at håndtere hændelser.

Forudsætter Kapitel 12, Interfaces, afsnittet om final i Kapitel 21, Avancerede klasser (og Kapitel 13, Hændelser og Kapitel 17, Flertrådet programmering for at forstå nogle af eksemplerne).

Indre klasser er mindre "hjælpeklasser" defineret inde i en anden klasse. Dette kapitel 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 kapitlet):

- Den indre klasse er knyttet til den ydre klasse og kan kun anvendes i denne.Man behøves 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.
- Den indre klasse kan arbejde direkte på den ydre klasses variabler og metoder, også de private. Det skyldes, at et objekt af en indre klasse altid hører til et objekt af den ydre klasse.

22.1. 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 klasses variabler og metoder - også de private. Den indre klasse er knyttet til et objekt af den ydre klasse.

22.1.1. Eksempel - Linjetegning

Man benytter ofte indre klasser i forbindelse med at lytte efter hændelser. Her kommer Linjetegning-eksemplet fra Kapitel 11 igen, men hvor vi lader en indre klasse lytte efter museklik.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class LinjetegningIndre extends Applet
{
  // Selv private variabler er synlig for den indre klasse
  private Point trykpunkt = null;
  private Point slippunkt = null;

  public void init()
  {
    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 paint (Graphics g)
  {
    if (trykpunkt != null && slippunkt != null)
      g.drawLine (trykpunkt.x, trykpunkt.y, slippunkt.x, slippunkt.y);
  }
}
```

Læg mærke til, at den indre klasse uden videre har adgang til den ydre klasses variabler og metoder.

22.2. Lokale 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 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 args[]){
    YdreKlasseMedLokalKlasse ydre = new YdreKlasseMedLokalKlasse();
    ydre.prøvLokaltObjekt(2);
  }
}
```

Resultatet bliver:

```
4
3
2
1
```

22.3. Anonyme klasser

En anonym klasse er en klasse uden navn, som der oprettes et objekt ud fra der, hvor den defineres.

```
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).

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.

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.

22.3.1. Eksempel - filtrering af filnavne

Følgende program udskriver alle javafiler i det aktuelle katalog. 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 afgører, om en fil skal tages med i listningen (se evt. Javadokumentationen).

```
import java.io.*;
public class FilnavnfiltreringMedAnonymKlasse
{
  public static void main(String arg[])
  {
    File f = new File( "." );      // det aktuelle katalog

    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] );
  }
}
```

Resultatet bliver

```
YdreKlasseMedLokalKlasse.java
FilnavnfiltreringMedAnonymKlasse.java
LinjetegningAnonym.java
AnonymeTraade.java
A.java
BenytIndreKlasser.java
```

22.3.2. 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. med det tidligere eksempel).

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class LinjetegningAnonym extends Applet
{
  private Point trykpunkt = null;
  private Point slippunkt = null;

  public void init()
  {
    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 paint (Graphics g)
  {
    if (trykpunkt != null && slippunkt != null)
```

```
      g.drawLine (trykpunkt.x, trykpunkt.y, slippunkt.x, slippunkt.y);
  }
}
```

22.3.3. 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 løber de i en anden løkke 1.000.000 gange 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();
    }
  }
}
```

Resultatet bliver:

```
111122221223311332441114433221144332211442255115544332115544332211Færdig med 1.
54442233332233Færdig med 2.
544335544Færdig med 3.
54Færdig med 4.
```

```
555555555Færdig med 5.
```

Man ser, hvordan objekt nummer 1, der blev startet først, også er det første, der afslutter.

22.4. Opgaver

1. Tag TegnbareObjekter.java fra Kapitel 12 og lav (i init()-metoden) fem forskellige objekter, der implementerer Tegnbar-interfacet (brug anonyme klasser). De fem objekter skal have forskellig måde at reagere på tegn() og sætPosition().
2. Kig på javadokumentationen til interfacet Comparator 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 (Vector) med ti strenge, og test din sortering med Collections.sort(liste, Comparator-objekt).

Kapitel 23. Objektorienteret analyse og design

Indhold:

- Analyse: Finde vigtige ord, brugsmønstre, aktivitetsdiagrammer ogskærmbilleder
- 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 6, 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.

I traditionel systemudvikling udføres de fem faser efter hinanden, og en ny fase påbegyndes først når den forrige er afsluttet. Hver fase udmøntes i et dokument som senere kan bruges dokumentation af systemet.

Dette er i skarp modsætning til den måde som en selvlært umiddelbart ville programmere. 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.

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 idé 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 kapitel viser gennem et eksempel (et Yatzy-spil) en grov skitse til analyse og design af et program. Det er tænkt som inspiration til, hvordan man kunne gribe sit eget projekt an.

23.1. 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.

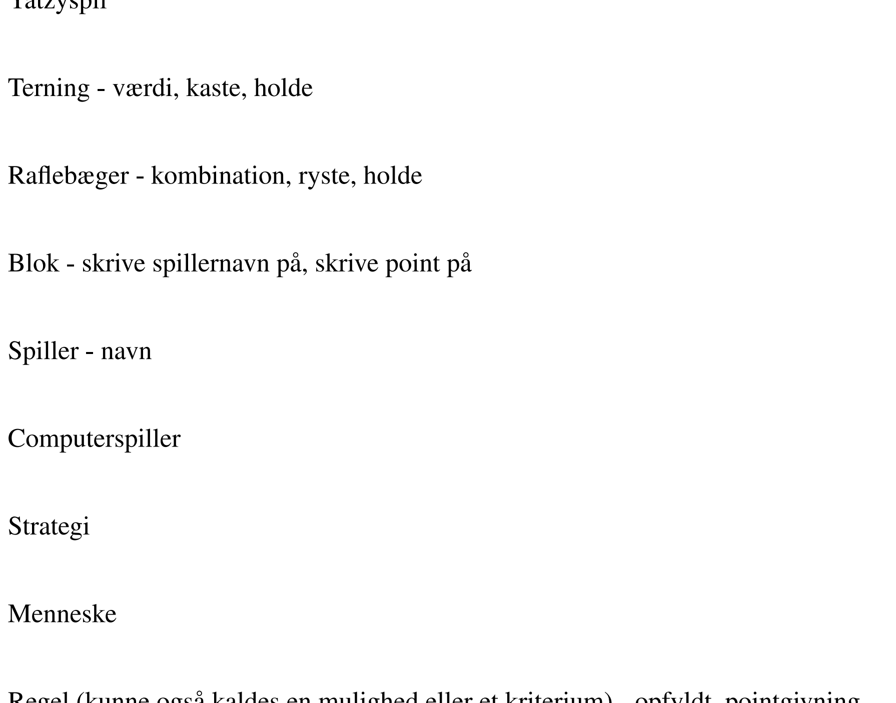
Efter at spillet er afsluttet, huskes resultatet i et lager, hvorfra man kan generere en hiscore-liste.

23.2. Objektorienteret analyse

Analysen skal beskrive hvad det er for ting og begreber programmet handler om. Analysens formål er at afspejle virkeligheden mest muligt.

23.2.1. 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) der knytter sig til tingen.



Yatzyspil

Terning - værdi, kaste, holde

Raflebæger - kombination, ryste, holde

Blok - skrive spillernavn på, skrive point på

Spiller - navn

Computerspiller

Strategi

Menneske

Regel (kunne også kaldes en mulighed eller et kriterium) - opfyldt, pointgivning

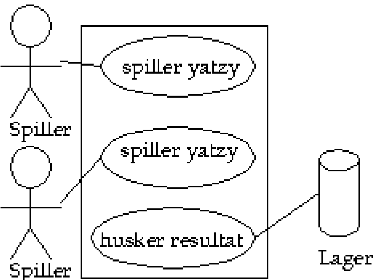
Lager

23.2.2. Brugsmønstre

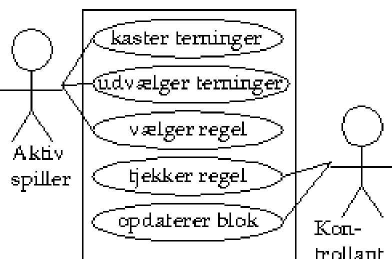
Brugsmønstre (eng.: Use Case) beskriver en samling af aktører og hvilke brugsmønstre, de deltager i. Man starter helt overordnet og går mere og mere i detaljer omkring hvert brugsmønster.

Man kan hævde, at Yatzy-spillet er på grænsen til at være for simpelt til at lave brugsmønstre. Herunder to brugsmønstre. Til venstre et meget overordnet der beskriver to spillere og lageret som aktører. Til højre brugsmønstret omkring en tur.

Figur 23-1. Java



Figur 23-2. Tur

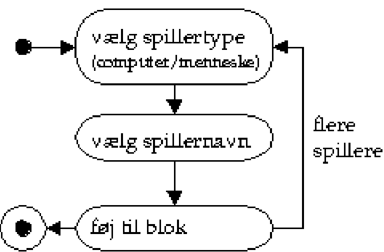


23.2.3. Aktivitetsdiagrammer

Aktivitetsdiagrammer beskriver den rækkefølge, som adfærdsmønstre og aktiviteter foregår i.

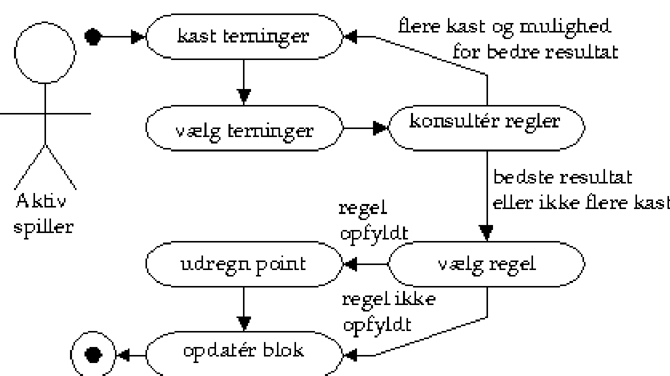
Eksempel: Aktiviteten "definere deltagere i spillet":

Figur 23-3. Java



Herunder ses et diagram for spillets gang, "en tur":

Figur 23-4. Java



23.2.4. Skærmbilleder

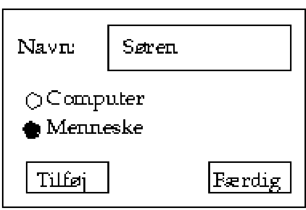
Hvis skærmbilleder er en væsentlig del af ens program, er det en god hjælp at tegne de væsentligste for at gøre sig klart, hvilke elementer programmet skal indeholde.

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 skærmbillede, så man kan også med det samme give dem sigende navne.

Når programmet startes skal vælges 2-6 spillere, hvoraf nogle kan være computerspillere:

Figur 23-5. Java

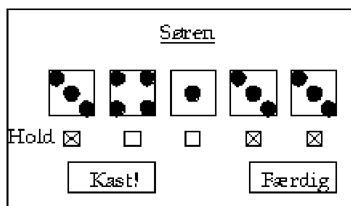


TilfoejSpillervindue

Under selve spillet skiftes spillerne til at få tur.

For menneske-spillerne dukker dette billede op:

Figur 23-6. Java



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:

Figur 23-7. Java

	Jacob	**Søren**
Ettere	4	
Toere		
Treere		9
etc...		
Sum		
Bonus		
Et par		
etc...		
Sum		

Blokvindue

23.3. Objektorienteret design

Designets formål er at beskrive, hvordan programmet skal implementeres.

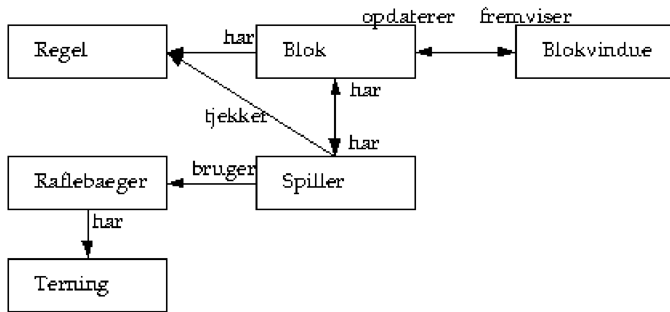
Her skal man bl.a. identificere de vigtigste klasser i systemet og lede efter ligheder mellem dem med henblik på nedarvning og genbrug.

23.3.1. Kollaborationsdiagrammer

Nyttige diagramformer under design er kollaborationsdiagrammer (samarbejdsdiagrammer), hvor man beskriver relationerne mellem klasserne eller objekterne på et overordnet plan.

Her er et eksempel:

Figur 23-8. Java



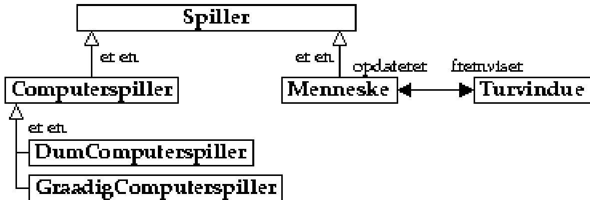
Har-relationer giver et vink om, at et objekt har en reference til (evt. ejer) et andet objekt. F.eks.:

- 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), og spillerne ved de hører til en blok hvor deres resultater skal skrives ind på.
- lokkens 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, og 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 Spillere i diagrammet ovenfor. Så kan man tegne et separat diagram, der viser rollerne.

Figur 23-9. Java



Er-en-relationer angiver generalisering eller specialisering (hvor nedarvning kan være en fordel). Det tegnes oftest med en hul pil.

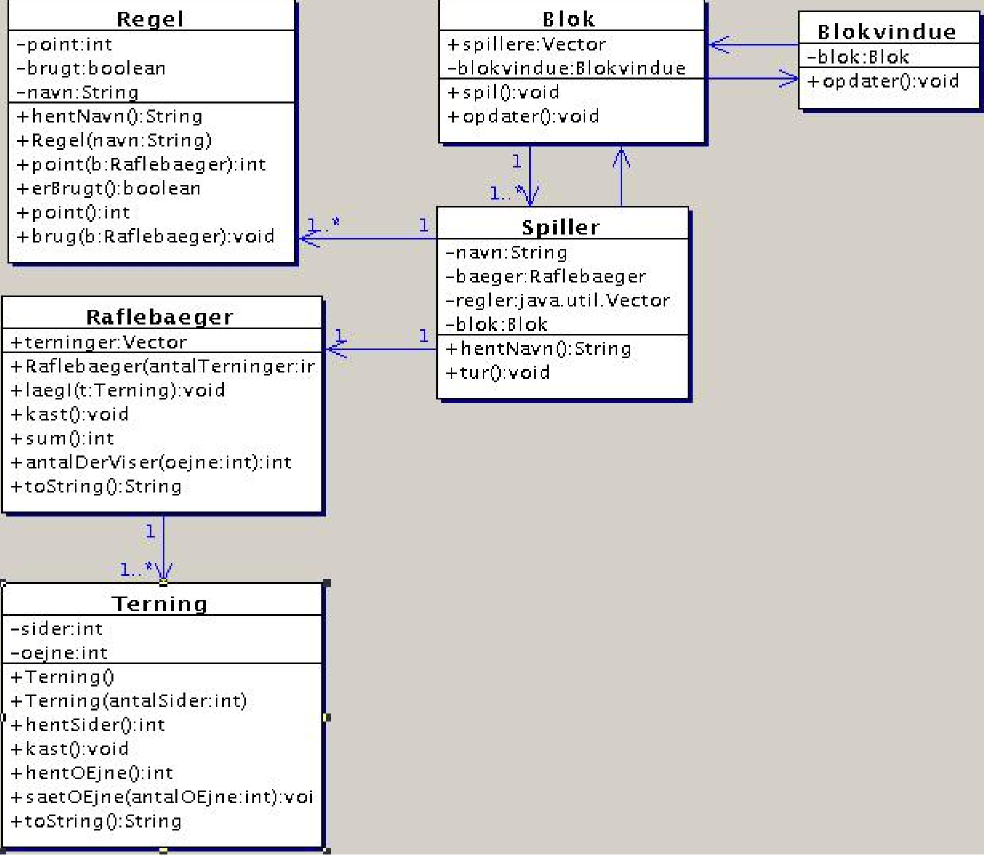
Her er det lidt specielle, at én type spiller (nemlig Menneske) har et vindue tilknyttet. Dette vindue skal jo have adgang til at vise terningerne, så man skal huske at sørge for, at spillere har en reference til raflebægeret.

23.3.2. Klassediagrammer

Herefter kan skitseres klassediagrammer, hvor man fastlægger nedarvning (er-en-relationerne), de vigtigste variabler og referencerne mellem objekterne (har-relationer) og de vigtigste metoder.

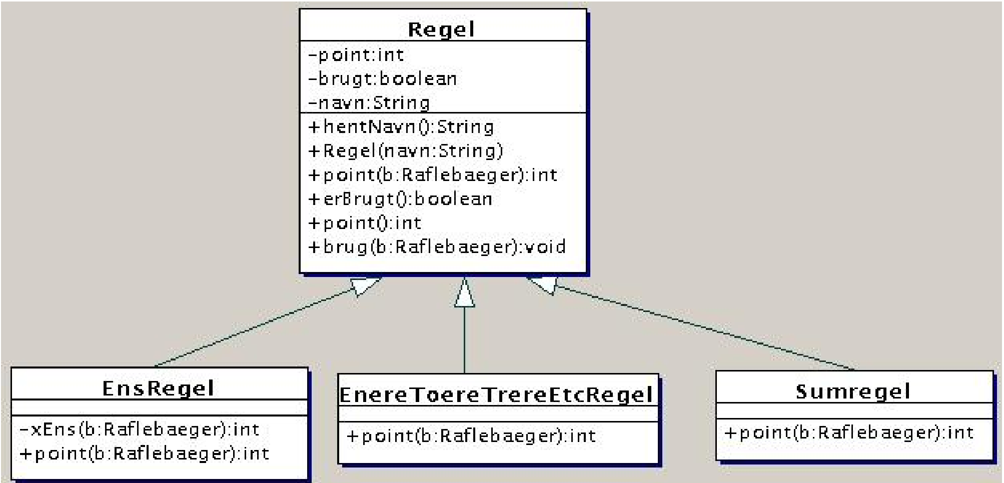
Dette kan eventuelt tegnes med et UML-udviklingsværktøj (f.eks. TogetherJ der kan hentes på http://www.togethersoft.com (http://www.togethersoft.com/)), der samtidig kan generere kode til programmeringsfasen.

Figur 23-10. Java



Herunder ses, hvilke typer regler der kunne forekomme.

Figur 23-11. Java



Kapitel 24. Internationale programmer

24.1. Indledning

Java hjælper dig med at gøre dine programmer platformsuafhængige, men hvad hvis de også skal være sproguafhængige? Når et program skal anvendes af brugere med anden sproglig og kulturel baggrund så opstår der behov for at programtekster, beløb og datoangivelser afhænger af brugerens land, sprog og eventuelt andre faktorer. Dette benævnes *lokalafhængighed*.

En mulig måde at løse problemet på er at vedligeholde kildeteksten til sit program i flere forskellige versioner, der hver især understøtter et bestemt sprog. Man finder hurtigt ud af at denne løsningsmetode er uholdbar i længden idet ændringer og opdateringer til programmet medfører at samtlige versioner skal ændres. Ligeledes skal kildeteksten oversættes hver eneste gang. En bedre fremgangsmåde er at internationalisere og lokalisere sit program.

Internationalisering er den proces, hvor du laver et programdesign, som er sprogligt og kulturelt neutralt. Al formatering og fortolkning af tal-, beløbs-, dato- og klokkeslætangivelser skal med andre ord være afhængigt af en række parametre, der på kørsels- eller oversættelsestidspunktet bestemmer den konkrete sproglige og kulturelle kontekst som programmet skal indgå i. Ligeledes skal sproglige tekster og direkte stinavne til billeder adskilles fra kildeteksten.

Internationalisering er ikke nok i sig selv. *Lokalisering* er den process, hvor du tilpasser dit internationaliserede program så det imødekommer de sproglige og kulturelle krav, der stilles af den givne målgruppe.

Data som skal fremvises til brugeren kan kategoriseres ud fra om det er lokalafhængigt eller lokaluafhængigt.

- Lokalafhængige data dækker f.eks. over tekst, billeder og lyd, der er udformet med en sproglig og geografisk målgruppe in mente. Disse kan umiddelbart vises til brugeren hørende til den bestemte målgruppe.
- Lokaluafhængige data dækker f.eks. over datoer, tal, beløb og klokkeslæt, der i sig selv ikke har tilknytning til nogen bestemt sproglig og geografisk målgruppe. Disse skal først formateres med en lokalafhængig formateringsfunktion før de kan vises til brugeren.

JDK stiller værktøjer til rådighed som hjælper dig med både internationaliserings- og lokaliseringsprocessen, herunder håndtering af lokalafhængige og uafhængige data. De næste mange sektioner går i dybden med disse værktøjer.

I Afsnit 24.2 beskrives hvordan man i sin programkode angiver en bestemt sproglig og geografisk region.

I Afsnit 24.3 beskrives hvordan lokalafhængige data håndteres, mens Afsnit 24.5 og Afsnit 24.6 beskæftiger sig med formatering af lokaluafhængige data.

24.2. Lokalindstillinger

I JDK anvendes objekter af typen `java.util.Locale` til at angive brugerens sprog og geografiske region. Vi vil kalde et konkret `Locale`-objekt for en *lokalindstilling*. Klasser, der varierer deres adfærd på baggrund af en lokalindstilling, vil vi kalde for *lokalafhængige*.

Lokalindstillinger udfører intet i sig selv, men overgives til andre lokalafhængige objekter, som udfører det egentlige arbejde, f.eks. at formatere en beløbsstørrelse.

24.2.1. Oprettelse af en lokalindstilling

En lokalindstilling kan konstrueres ved at angive sprog- og landekode, f.eks.

```
Locale xLocale = new Locale("da", "DK");
```

En lokalindstilling kan oprettes med et tredje, brugerdefineret argument. Dette kan f.eks. signalere platform.

```
Locale xLocale = new Locale("da", "DK", "UNIX");
Locale yLocale = new Locale("da", "DK", "WINDOWS");
```

Det brugerdefineret argument har ingen speciel betydning i JDK. Programmøren står selv for at bestemme betydningen af indholdet.

Det er også muligt at undlade landekoden ved at lade den være tom, men sprogkoden er obligatorisk.

```
Locale xLocale = new Locale("da", "");
```

Som oftes er vi blot interesseret i at benytte en global lokalindstilling, hvilket opnås ved at invokere en klassemetode på `Locale`:

```
Locale currentLocale = Locale.getDefault();
```

Dette objekt angiver den virtuelle maskines nuværende sprog- og regionsindstillinger. Ved opstart af den virtuelle maskine svarer denne lokalindstiling til brugerens indstillinger i operativsystemet.

Det er muligt at ændre den globale lokalindstilling ved at kalde klassemetoden `Locale.setDefault(Locale aLocale)` men vær opmærksom på at *alle* programmer i den virtuelle

maskine vil blive berørt, da de lokalafhængige klasser anvender den globale lokalindstilling i de situationer, hvor man ikke explicit fortæller hvilken lokalindstilling, der skal anvendes.

24.2.2. Tilgængelige lokalindstillinger

Man kan undersøge hvilke lokalindstillinger der er tilgængelige ved at læse afsnittet om understøttede lokalindstillinger i JDK-dokumentationen.

- http://java.sun.com/j2se/1.3/docs/guide/intl/locale.doc.html (http://java.sun.com/j2se/1.3/docs/guide/intl/locale.doc.html)

Det er også muligt at afgøre med `Locale.getAvailableLocales()`.

```
import java.util.Locale;

public class AvailableLocales
{
  public static void main(String[] args)
  {
    Locale[] l = Locale.getAvailableLocales();
    for (int i=0; i<l.length; i++) System.out.print(l[i]+" ");
    System.out.println( );
  }
}
```

Kørsel af ovenstående program giver en masse lokalindstillinger

```
en en_US ar ar_AE ar_BH ar_DZ ar_EG ar_IQ ar_JO ar_KW ar_LB ar_LY ar_MA ar_OM
ar_QA ar_SA ar_SD ar_SY ar_TN ar_YE be be_BY bg bg_BG ca ca_ES ca_ES_EURO cs
cs_CZ da da_DK de de_AT de_AT_EURO de_CH de_DE de_DE_EURO de_LU de_LU_EURO el
el_GR en_AU en_CA en_GB en_IE en_IE_EURO en_NZ en_ZA es es_BO es_AR es_CL es_CO
es_CR es_DO es_EC es_ES es_ES_EURO es_GT es_HN es_MX es_NI et es_PA es_PE es_PR
es_PY es_SV es_UY es_VE et_EE fi fi_FI fi_FI_EURO fr fr_BE fr_BE_EURO fr_CA
fr_CH fr_FR fr_FR_EURO fr_LU fr_LU_EURO hr hr_HR hu hu_HU is is_IS it it_CH
it_IT it_IT_EURO iw iw_IL ja ja_JP ko ko_KR lt lt_LT lv lv_LV mk mk_MK nl nl_BE
nl_BE_EURO nl_NL nl_NL_EURO no no_NO no_NO_NY pl pl_PL pt pt_BR pt_PT pt_PT_EURO
ro ro_RO ru ru_RU sh sh_YU sk sk_SK sl sl_SI sq sq_AL sr sr_YU sv sv_SE th th_TH
tr tr_TR uk uk_UA zh zh_CN zh_HK zh_TW
```

Som nævnt i sidste afsnit består en lokalindstilling af sprogkode, landekode og valgfrit argument. Ud fra ovenstående ser vi at der f.eks. er lokalindstillinger for

- fr_BE: Fransk i Belgien
- fr_BE_EURO: Fransk i Belgien med euro-valuta
- fr_CA: Fransk i Canada
- fr_FR: Fransk i Frankrig

- fr_LU: Fransk i Louxembourg

Sprog- og landekoderne er ISO-standarder, som kan findes på nettet.

- Sprog, ISO-639 http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt (http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt)
- Landekoder, ISO-3166 http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html (http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html)

Lad os nu se på hvordan lokalindstillinger kan anvendes.

24.3. Ressourcebundter

Ressourcebundter bruges til at isolere lokalafhængige data fra din kode, f.eks. tekster eller billeder på knapper. Dette afsnit går i dybden med klasserne `java.util.ResourceBundle`, `java.util.PropertyResourceBundle` og `ListResourceBundle` som er nyttige i denne sammenhæng.

24.3.1. Generelt om ressourcebundter

Et ressourcebundt er en samling af lokalafhængige data, som hver især er associeret med en unik tekstnøgle. Nøglen bruges når der skal hives et bestemt element ud af bundtet, f.eks. et billede.

Et ressourcebundt hører til en familie af ressourcebundter, som deler et fælles basisnavn, f.eks. "dk.sslug.LogInd" eller "dk.sslug.MitRessourcebundt". Alle medlemmer i familien har et unikt navn på formen `basisnavn[_sprogkode[_landekode[_variant]]]`, der afspejler den lokalindstilling, som de understøtter.

F.eks. kunne vi have følgende ressourcebundter

```
dk.sslug.LogInd
dk.sslug.LogInd_de
dk.sslug.LogInd_da_DK
dk.sslug.LogInd_da_DK_LINUX
```

Alle familiemedlemmer indeholder de samme data i en lokaliseret udgave. I en given familie identificeres et bestemt dataelement altid med den samme nøgle - uanset hvilket familiemedlem man har fat i.

Når du i dit program ønsker at tilgå data hørende til et ressourcebundt så gøres det ved at bruge klassemetoden `getBundle(String, Locale)` på klassen `ResourceBundle` som vist forneden.

```
Locale daLocale = new Locale("da", "DK");
ResourceBundle logIndBundt =
  ResourceBundle.getBundle("dk.sslug.LogInd", daLocale);
```

Metoden vil først forsøge at finde ressourcebundtet "dk.sslug.LogInd_da_DK", dernæst "dk.sslug.LogInd_da" og endeligt "dk.sslug.LogInd". Hvis ingen af dem findes så kastes en `MissingResourceException`. På grund af dette bør der altid være et standard ressourcebundt med basisnavnet som eneste navn.

I JDK er der to måder at definere et ressourcebundt på: som en ressourcefil eller som en klasse. De to næste afsnit beskriver dette nærmere.

24.3.2. Lagring af tekst i ressourcefiler

Det er muligt at lagre lokalafhængig tekst i en dedikeret ressourcefil kaldet for en property-fil. Det er en simpel tekstfil, der bruges til at associere tekstnøgler med lokalafhængige tekster. En association er på formen `nøglenavn = værdi`, hvor nøglen kan være navndøbt mere eller mindre abitrært. I praksis er det fornuftigt at vælge et signende nøglenavn.

Forneden ses et eksempel på en ressourcefil. Kommentarlinjer starter med havelågetegn (#).

```
# LogInd_da_DK.properties
# Dansk lokalindstilling for Log ind-skærmbilledet
IndtastNavnLabel = Indtast dit navn
IndtastAdgangskodeLabel = Indtast din adgangskode
LogIndKnap = Log ind
AnnullerKnap = Annuller
```

Hvis tekstfilen placeres i "dk/sslug/" relativt fra classpath så får ressourcebundtet navnet "dk.sslug.LogInd_da_DK". Helt analogt får ressourcebundet for de to nedenstående ressourcefiler navnene "dk.sslug.LogInd" og "dk.sslug.LogInd_de".

```
# LogInd.properties
# Standard lokalindstilling for Log ind-skærmbilledet
IndtastNavnLabel = Enter name
IndtastAdgangskodeLabel = Enter password
LogIndKnap = Log in
AnnullerKnap = Cancel

# LogInd_de.properties
# Mangelfuld tysk lokalindstilling for Log ind-skærmbilledet
AnullerKnap = Abbrechen
```

Ressourcefiler håndteres af `PropertyResourceBundle`, der er en subklasse af `ResourceBundle`. Som programmør behøver man dog kun benytte sig af moderklassen, hvilket følgende eksempel viser.

```
package dk.sslug;

import java.util.Enumeration;
import java.util.Locale;
import java.util.ResourceBundle;

public class PropertyResourceBundleEksempel
{
  public static void main(String[] args)
  {
    udskrivVaerdier(new Locale("da", "DK"));
    udskrivVaerdier(new Locale("de"));
    udskrivVaerdier(new Locale("fr", "CA"));
  }

  private static void udskrivVaerdier(Locale locale)
  {
    ResourceBundle logIndBundt =
      ResourceBundle.getBundle("dk.sslug.LogInd", locale);
    Enumeration enum = logIndBundt.getKeys();
    while (enum.hasMoreElements()) {
      String noegle = (String) enum.nextElement();
      System.out.println(noegle +" = " + logIndBundt.getString(noegle));
    }
    System.out.println();
  }
}
```

Køres programmet så fås

```
IndtastAdgangskodeLabel = Indtast din adgangskode
LogIndKnap = Log ind
IndtastNavnLabel = Indtast dit navn
AnnullerKnap = Annuller

AnullerKnap = Abbrechen
IndtastAdgangskodeLabel = Enter password
LogIndKnap = Log in
IndtastNavnLabel = Enter name
AnnullerKnap = Cancel

IndtastAdgangskodeLabel = Enter password
LogIndKnap = Log in
IndtastNavnLabel = Enter name
AnnullerKnap = Cancel
```

`PropertyResourceBundle` benytter sig internt af `java.text.Properties` som repræsenterer en persistent mængde of nøgle/værdi-par. Den er nyttig såfremt du ønsker dit program skal benytte sig af opsætningsfiler. Man kan finde mere information i JDK's API dokumentation.

24.3.3. Lagring af ressourcer i klasser

TODO

24.4. Parametriserede beskeder

TODO

24.5. Formatering af datoer og klokkeslæt

`Date`-objekter repræsenterer datoer og klokkeslæt. Dette afsnit går i dybden med de lokalafhængige klasser `java.text.DateFormat` og `java.text.SimpleDateFormat` som er velegnet til at formattere `Date`-objekter.

24.5.1. Prædefineret formater

Klassen `DateFormat` indeholder en række statiske fabriksmetoder, som returner specialiseret formateringsobjekter. Seks af disse er

```
getDateInstance(int style)
getDateInstance(int style, Locale aLocale)
getTimeInstance(int style)
getTimeInstance(int style, Locale aLocale)
getDateTimeInstance(int dateStyle, int timeStyle)
getDateTimeInstance(int dateStyle, int timeStyle, Locale aLocale)
```

Metoderne tager imod en eller flere `style`-parametre, som anvendes til at angive længden af formateringsresultatet. De mulige værdier er defineret som konstanter i `DateFormat`-klassen.

- `SHORT` forsøger at være numerisk og kort, f.eks. 12/11/01 og 2:31 AM
- `MEDIUM` er længere, f.eks. Dec 11, 2001 og 2:31:35 AM
- `LONG` er endnu længere, f.eks. December 11, 2001 og 2:31:35 AM CET
- `FULL` er fuldstændig specificeret, f.eks. Tuesday, December 11, 2001 og 2:31:35 AM CET
- `DEFAULT` svarer til `DateFormat.MEDIUM`.

Det præcise resultat afhænger af lokalindstillingen. Hvis vi i det ovenstående havde brugt en dansk lokalindstilling så ville der ikke være nogen synlig forskel på brugen af LONG og FULL.

Her er et simpelt eksempel, der viser brugen af `DateFormat` med den globale lokalindstilling.

```
import java.text.*;
import java.util.*;

public class DateFormatExample
{
  public static void main(String arg[])
  {
    DateFormat klformat, datoformat, dkf;
    klformat   = DateFormat.getTimeInstance(DateFormat.MEDIUM);
    datoformat = DateFormat.getDateInstance(DateFormat.FULL);
    dkf = DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.SHORT);

    Date tid = new Date();
    System.out.println( tid );
    System.out.println( "Kl   :"+ klformat.format(tid) );
    System.out.println( "Dato :"+ datoformat.format(tid) );
    System.out.println( "Tid  :"+ dkf.format(tid) );
  }
}
```

Kørsel af ovenstående program med dansk lokalindstilling (da_DK) giver

```
Mon Dec 03 13:28:06 GMT+01:00 2001
Kl:   13:28:06
Dato: 3. december 2001
Tid:  03-12-2001 13:28
```

Hvis lokalindstillingen er amerikansk (en_US) så fås imidlertid

```
Mon Dec 03 13:27:57 GMT+01:00 2001
Kl:   1:27:57 PM
Dato: Monday, December 3, 2001
Tid:  Dec 3, 2001 1:27 PM
```

Læg i øvrigt mærke til at `Date`-objektets `toString()`-metode ikke er lokaliseret. Den bør kun bruges til testudskrifter og logning, og ikke i tekst som brugeren skal læse (med mindre du bevidst ønsker at irritere ham/hende).

24.5.2. Egne formater

Ønsker man som programmør større kontrol over hvordan datoen bliver formateret så må man selv specificere formatet med `SimpleDateFormat`.

Lad os starte ud med et eksempel.

```
import java.text.*;
import java.util.*;
```

```
public class BenytSimpleDateFormat
{
  public static void main(String arg[])
  {
     DateFormat df = new SimpleDateFormat("EEEE 'den' d. MMMM 'år' yyyy.");

     Date tid = new Date();
     System.out.println( df.format(tid) );
  }
}
```

Kørsel af ovenstående program med dansk lokalindstilling giver

```
mandag den 10. december år 2001.
```

En tilsvarende kørsel af programmet med amerikansk lokalindstilling giver

```
Monday den 10. December ?r 2001.
```

Ud fra eksemplet ses at formateringsresultatet afhænger af to faktorer: lokalindstillingen samt argumentet angivet til `SimpleDateFormat`s konstruktør. Argumentet angiver et mønster, som angiver hvordan dato- og klokkeslætformateringen logisk bør tage sig ud. Dato- og klokkeslætformater specificeres ved hjælp af bogstaverne a-z og A-Z som har en speciel betydning så længe de ikke er omgivet af apostroffer. F.eks. dækker M over måneden i et år, mens E dækker over dagen i en uge. Antallet af gange et bogstav gentages er ikke helt uden betydning. Gentages M fire gange som i eksemplet så skrives måneden som en tekst, f.eks. 'december', men vises den kun en gang så skrives måneden som et tal, f.eks. 12. Bemærk at konstanterne `SHORT`, `MEDIUM`, `LONG` og `FULL` i `DateFormat`-klassen afspejler dette.

I skrivende stund er det ikke alle bogstaver, der har fået tillagt en betydning. Nedenstående tabel viser dem som findes i JDK 1.4. En lignende tabel findes også i API-dokumentationen for `SimpleDateFormat`.

Tabel 24-1. Dato og klokkeslætmønstre

Tegn	Betydning	Eksempler med dansk lokalindstilling (1-4 ens)
G	Tidsregningbetegnelse	AD, AD, AD, AD
y	År	02, 02, 02, 2002
M	Måneden i et år	6, 06, jun, juni
w	Ugen i et år	24, 24, 024, 0024
W	Ugen i en måned	2, 02, 002, 0002
D	Dagen i et år	162, 162, 162, 0162
d	Dagen i en måned	11, 11, 011, 0011
F	Ugedagen i en måned	2, 02, 002, 0002

Tegn	Betydning	Eksempler med dansk lokalindstilling (1-4 ens)
E	Dagen i en uge	ti, ti, ti, tirsdag
a	Am/pm	PM, PM, PM, PM
H	Timen i en dag (0-23)	19, 19, 019, 0019
k	Timen i en dag (1-24)	19, 19, 019, 0019
K	Timen i am/pm (0-11)	7, 07, 007, 0007
h	Timen i am/pm (1-12)	7, 07, 007, 0007
m	Minuttet i en time	49, 49, 049, 0049
s	Sekundet i et minut	22, 22, 022, 0022
S	Millisekund	689, 689, 689, 0689
z	Tidszone (Generel)	CEST, CEST, CEST, Central European Summer Time
Z	Tidszone (RFC 822)	+0200, +0200, +0200, +0200

Ugedagen i en måned kræver en kort forklaring. Hvis datoen er tirsdag den 11. maj 2002, så fortæller den os at det er den 2. tirsdag i maj måned 2002. Er datoen derimod lørdag den 15. maj 2002 så fortæller den os at det er den 3. lørdag i maj måned.

24.6. Formatering af tal og beløb

Denne sektion går i dybden med den lokalafhængige klasse `java.text.NumberFormat` som er velegnet til at formatere tal, beløb og procentstørrelser.

24.6.1. Prædefineret formater

Klassen NumberFormat indeholder en række statiske fabriksmetoder, som returnerer specialiseret formateringsobjekter.

```
getInstance()
getInstance(Locale locale)
getCurrencyInstance()
getCurrencyInstance(Locale locale)
getIntegerInstance()
getIntegerInstance(Locale locale)
getNumberInstance()
getNumberInstance(Locale locale)
getPercentInstance()
getPercentInstance(Locale locale)
```

`getInstance(..)` er synonym med `getNumberInstance(..)`, og returnerer det normale lokalafhængige `NumberFormat`-objekt. `getCurrencyInstance(..)` er beregnet til beløb, `getIntegerInstance(..)` til heltal og `getPercentInstance(..)` til procentstørrelser.

Uanset om man arbejder med heltal eller decimaltal så kan det være nyttigt at justere på antallet af cifre der skal vises før og efter et eventuelt decimaltegn. I Danmark er det f.eks. meget normalt at beløbsstørrelser vises med præcis to betydende decimaler. Dette hjælper følgende metoder med:

```
setMaximumIntegerDigits(int vaerdi)
setMinimumIntegerDigits(int vaerdi)
setMaximumFractionDigits(int vaerdi)
setMinimumFractionDigits(int vaerdi)
```

De to første metoder omhandler heltalsdelen af et tal, men de to andre omhandler decimaldelen. `setMaximumXXX(8)` betyder at der vises højest 8 cifre i enten heltal- eller decimaldelen, mens `setMinimumXXX(2)` betyder at der vises mindst 2 cifre. Overskydende cifre skæres væk mens manglende cifre erstattes med nuller.

Nedenstående eksempel viser brugen af `NumberFormat`-klassen.

```
package dk.sslug;

import java.util.Locale;
import java.text.DecimalFormat;
import java.text.NumberFormat;

public class Talformatering
{
  public static void main(String[] args)
  {
    Locale[] locales = NumberFormat.getAvailableLocales();
    for (int i = 0; i < locales.length; i++) {
      System.out.println(locales[i].toString());
      formaterTal(locales[i]);
      formaterBeloeb(locales[i]);
      formaterProcent(locales[i]);
      System.out.println();
    }
  }

  public static void formaterTal(Locale locale) {
    NumberFormat nf = NumberFormat.getNumberInstance(locale);
    String heltal     = nf.format(123456789);
    String decimaltal = nf.format(123456.789);
    System.out.println(heltal);
    System.out.println(decimaltal);
  }

  public static void formaterBeloeb(Locale locale) {
    NumberFormat cf = NumberFormat.getCurrencyInstance(locale);
```

```
    cf.setMaximumFractionDigits(2);
    String beloeb = cf.format(123456.789);
    System.out.println(beloeb);
  }

  public static void formaterProcent(Locale locale) {
    NumberFormat pf = NumberFormat.getPercentInstance(locale);
    String procent = pf.format(1.42);
    System.out.println(procent);
  }
}
```

Eksemplet gennemløber alle lokalindstillinger, som NumberFormat explicit kender. Et udsnit af resultatet ved at køre ovenstående program er vist her forneden.

```
...

da
123.456.789
123.456,789
¤ 123.456,79
142%

da_DK
123.456.789
123.456,789
kr 123.456,79
142%

...

en_CA
123,456,789
123,456.789
$123,456.79
142%

en_GB
123,456,789
123,456.789
£123,456.79
142%

...
```

24.7. Tekster og tegn

Java tilbyder klasser til at

- Analysere tegn
- Sammenligne strenge
- Finde text boundaries i sætninger
- Konvertere non-unicode tekst

24.7.1. Analyse af tegn

```
Følgende kode duer ikke:
char ch;

if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
 //ch is a letter

if (ch >= '0' && ch <= '9')
 //ch is a digit

if (ch == ' ' || ch == '\n' || ch == '\t')
 //ch is a whitespace

Benyt altid metoderne på klassen Character:
isDigit
isLetter
isLetterOrDigit
isLowerCase / isUpperCase
isSpaceChar
isDefined
```

24.7.2. Sammenligning af strenge

```
Sikker sammenligning af strenge gøres vha. metoden compare() på klasssen Collator.

Sammenligning kan ikke gøres sikkert med String.compareTo(), idet den
sammenligner binært på Unicode-niveau, hvilket ikke altid stemmer
overens med et sprogs tegn-orden.

Således opnås en instans af Collator for en given Locale:
Collator myCol = Collator.getInstance(aLocale);

To strenge sammenlignes således:
myCol.compare(firstString, secondString);

compare() returnerer -1, 0 el. 1, afhængig af om firstString er
hhv. mindre end, lig med el. større end secondString.
```

24.7.3. Analyse af grænser i tekst

Klassen BreakIterator gør det muligt at finde grænser (boundaries) i form af positionsnumre i en tekst for en given Locale på følgende niveauer:

tegn
ord
sætning
linje

Ved instantieringen af BreakIterator invokeres én af følgende klassemetoder til at konstruere instansen med den ønskede egenskab:

getCharacterInstance
getWordInstance
getSentenceInstance
getLineInstance
Eksempel:

```
BreakIterator bi = BreakIterator.getWordInstance(aLocale);
```

Appendiks A. Revisionshistorie for bogen

"Linux - friheden til programmere i Java" er en nystartet bog. Vi frigiver ofte nye versioner, når der er kommer en del rettelser ind, eller nye afsnit er blevet skrevet. Kommentarer, ris og ros, og specielt fejl og mangler bedes sendt til linuxbog@sslug.dk (mailto:linuxbog@sslug.dk), men er du medlem af SSLUG, så skriv til sslug-bog@sslug.dk (mailto:sslug-bog@sslug.dk). Alle kan bidrage - se om tilmelding se på http://www.sslug.dk/tilmeld.

Her er en liste over, hvad der er ændret i bogen.

- Version 0.7.20040516 - 16. maj 2004: Jacob Sparre Andersen: Retter sprog. Claus Teglman: VisualAge er erstattet med WebSphere.
- Version 0.7 - 25. januar 2004: Jacob Sparre Andersen: Retter sproglige fejl.
- Version 0.6 - 7. oktober 2003: Jacob Sparre Andersen: Mindre sproglige rettelser.
- Version 0.5 - 5. maj 2003: Janet Chemnitz har rettelse til kapitelet omkring arrays. Lasse Thernøe havde en præcisering til navngivning af java filer mht. store og små bogstaver.
- Version 0.4 - 1. september 2002: Claus Madsen har kommet med rettelser, samt mere om Ant til kapitelet "Udviklingsmiljø". Alfred Jensen har kommet med en række rettelser til kapitelet "Basal programmering"
- Version 0.3 - 14. juni 2002: Jacob Nordfalk og Christian Damsgaard har lavet en generel oprydning af bogen. Erik Søe Sørensen retter en masse trykfejl i historie-afsnittet. Jacob Sparre Andersen: Rettet sproglige småfejl.
- Version 0.2 - 10. marts 2002: Ny licens for bogen - Åben dokumentlicens. Jacob Nordfalk, Jonas Kongslund og Christian Damsgaard har fået opdateret materialet fra Jacob's bog, mht. id attributter på alle <chapter>, <sect> og <figure> så filerne i afsnittene får rigtige navnet, samt at det er muligt at krydsreferere mellem de forskellige afsnit. Peter Toft har fået bogen til at oversætte med Jacob Nordfalks bog inde. Tilføjet indhold til sektionen Test. Tilføjet sektion om IDE'er Kapitlet om virtuelle maskiner er indlagt.
- Version 0.1 - 29. december 2001: Påbegyndt kapitler omhandlende historien bag Java, udviklingsmiljøer og udvikling af sproguafhængige programmer.
- Version 0.0 - 10. december 2001: Påbegyndt kapitler omhandlende historien bag Java, udviklingsmiljøer og udvikling af sproguafhængige programmer. Første offentlige version.