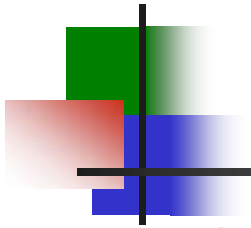
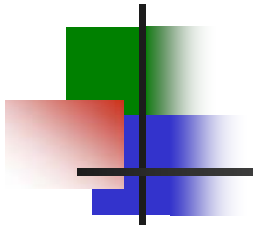


Litt om Javas class-filer og byte-kode

INF 5110, 11/5-2010, Stein Krogdahl (Dessverre litt få figurer)

- Disse formatene ble planlagt fra start som en del av hele Java-ideen
  - Byte-koden gir portabilitet ved at den utføres av en interpretator (som må skrives for hver enkelt maskin, gjerne i C eller C++)
  - Gir, sammen med et standard bibliotek, et enhetlig grensesnitt til operativsystemet, grafikk, osv. fra Java
  - Samme Java-kompilator kan dermed brukes på alle maskiner
  - For effektivitet: Byte-koden blir nå ofte oversatt til maskinkode for den aktuelle maskinen før utførelsen (JIT-kompilering), f.eks. slik:
    - Hele programmet oversettes som en del av ”loadingen”.
    - Klassene oversettes etter hvert som eksekveringen når dem
    - Avansert: Når man ser at noe utføres ofte, oversettes dette

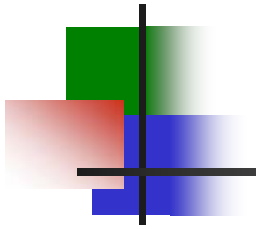
- Likner mye på den lavnivå-koden dere oversetter til i Oblig 2
  - Men hos dere blir også ”Loading” gjort samtiding som koden blir laget.
  - I Javas byte-kode blir alle navn beholdt på tekstlig form i den byte-koden som legges på fil (som klasse-filer)
  - Først når denne taes inn av loaderen blir den laget om til binær kode.



Litt om Javas class-filer og byte-kode

- Class-filer inneholder både:
  - Den utførbare koden (som byte-kode = sekvenser av byte-instruksjoner),
  - Og hele strukturen av klassen, med info om navn, variable, metoder, parametere, typer, etc.
  - Disse to informasjonstypene ligger tradisjonelt på to forskjellige filer: For C og C++, på .c-filer (som oversettes til maskinkode) og på .h-filer

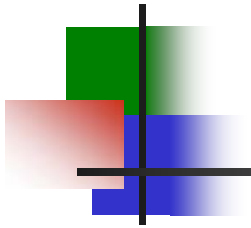
- En class-fil leses derved i to sammenhenger i forbindelse med kompilering/kjøring:
  - Når en annen klasse som referer til denne (f.eks. en subklasse) kompileres: Da ser man bare på struktur-delen av klasse-filen
  - Når klassen skal loades: Da ser man også på den utførbare byte-koden for hver av metodene i klassen



Formatet av Javas class-filer

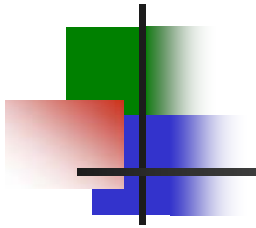
- På hver class-fil er det bare beskrivelse av én klasse eller ett grensesnitt
- Class-filene har all informasjon om:
  - Navn på klassen, og på superklassen og implementerte grensesnitt
  - Hvilke variable klassen har, ved navn, type og synlighet
  - Hvilke metoder den har, ved navn, type, synlighet, parametere (med typer), samt deres byte-kode-sekvens

- class-filene er rasjonalsiert slik at navn/strenger i programmet bare er lagret en gang.
- Dette gjøres i et ”navne-område” (også kalt ”konstant-området”)
  - Her ligger alle tekster, navn, og tall-verdier (på tekstlig form) som brukes i programmet, pent etter hverandre, og kan aksesseres ved sin “indeks”
  - Når disse skal brukes ellers i selve byte-koden, angir man bare indeksen for dette navnet i navne-området.



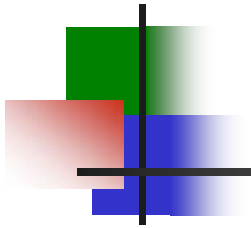
Selve den utførbare byte-koden

- Byte-koden har samme idé som P-koden, ved at arbeids-dataene skal ligge på en stakk under utførelsen
- Funksjons-delen av instruksjonen er på én byte. Det er altså plass til 256 forskjellige instruksjoner (som faktisk er litt lite)
  - Byte-koden har en add-instruksjon for hver aritmetisk type
- Byte-instruksjonenes ”adressefelt” (der dette trengs) angis ved fullt navn (tekstlig), type og klasse-tilhørighet
  - Det eneste unntaket fra dette er lokale variable i metoder. Disse angis som relativ-adresse (i byte) i aktiverings-blokken.
- Som antydet på forrige foil: Det blir mange navn det stadig skal refereres til.
  - Derfor ligger navnene bare én gang hver i et eget ”navne-område”, og de angis andre steder bare ved sin indeks i dette området



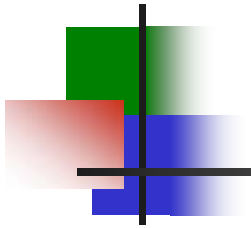
Hva foregår i en Java Virtual Machine (JVM)

- En JVM kan enten *interpretere* eller *oversette til maskinkode*
  - Men også om den vil interpretere vil den først gjøre bytekoden om til en *intern* bytekode-form der *alt er tall og fysiske adresser*
- Loaderen har en *verifikator* som kan sjekke innkommende byte-koden
- Loaderen starter med å lese inn og behandle den angitte class-fil
  - Leser så etter hvert inn alle class-filer som det referers til fra denne, osv.
- Lager en ”descriptor” for hver klasse.
  - Denne vil ligge fast under den kommende utførelse av programmet
  - Alle objekter har en peker til descriptoren for sin klasse
  - Descriptoren inneholder virtuell-tabellen for klassen, en peker til descriptoren for superklassen, etc.
  - Dersom debugging eller refleksivitet: Descriptoren inneholder også info om alle variable/metoder
  - Descriptoren kan også ha en peker til et sted der selve Java-koden ligger



Mer om: Hva foregår i en JVM

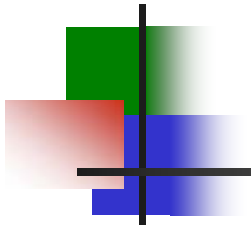
- Loaderen gjør ”allokering” av variable i klassene, dvs.:
  - Går gjennom byte-kode-sekvensen og gjør hver av de tekstlige operandene om til relativadresser, og alle klasse-angivelser (f.eks. i casting og ved ”new C...”) om til pekere til klassens descriptor
  - Merk at allokering av én klasse må gjøres *før* allokering for dens subklasser
- Dersom interpretering:
  - Legger sekvensen av byte-instruksjoner (nå med tall både for funksjonsangivelse og adresser) ut i et passelig format
  - Starter å interpretere dette
- Dersom oversetting:
  - Oversetter sekvensen byte-instruksjoner til maskinkode for gitt maskin
  - Kan også gjøres som en vanlig ”forhåndskompilering”, eller en JIT-kompilering (Just-In-Time) i forbindelse med at programmet skal startes opp.
- Kan også gjøre noe midt i mellom:
  - Interpretere først, men etter hvert oversette de metoder som brukes mest.
- Finnes også Java-kompilatorer som hopper over hele byte-kode-steget
  - Får da en tradisjonell kompilator, som kan lage effektiv kode
  - Men det er mer problematisk å koble seg til Javas standard-bibliotek etc.



Verifikatoren

- Denne kan gå gjennom klassefiler og sjekke at de er konsistente
  - Spesielt sjekke at bytekoden er konsistent (se under)
  - Og at den ikke gjør noe den ikke har autorisasjon til
  - Dette er spesielt aktuelt for class-filer som hentes inn over nettet

- Hva gjør verifikatoren med byte-koden:
  - ”Simulerer” hele tiden hva som vil ligge på stakken under utførelsen
  - Sjekker at det som ligger på toppen av stakken hele tiden stemmer typemessig etc. med den instruksjonen som skal utføres
  - Sjekker at når det gjøres hopp så er stakken helt lik der det hoppes fra og der det hoppes til.
  - Sjekker at de operasjonene som gjøres er lovlige (i henhold til autorisasjon)



Tråder, og utførelse i (ekte) parallell

- Når man i et Java-program kjører flere tråder setter man vanligvis av et "stort" område til stakken for hver tråd (typisk ¼ - 1 Mbyte).
  - Størrelsen av dette området kan vanligvis settes av brukeren.

- Men merk: på grunn av "paging-systemet" vil man ikke bruke så mye *fysisk* memory.
  - Én page er vanligvis 4 kbyte (altså 1/250 Mbyte) så man bruker hele antall av dette.
  - Men om stakken blir stor i starten og mindre siden vil man nok fortsette å bruke plass tilsvarende det største stakken noengang har vært

- Objektene som genereres legges på en heap som er felles for alle tråder.
  - Disse skal jo kunne aksesseres fra alle tråder
  - Det er satt av et bit i hvert objekt som angir om objektet er "låst"

Langsomme metodekall gjennom interfacer?

- Problemet er at de forskjellige klassene som implementerer en gitt interface har forskjellige adresse (= virtuell-indeks) for metodene. Det må derfor et tabellverk til for å få rask aksess, f.eks.:
  - Hver interface i programmet får sin egen "interface-indeks" (kan bli *store* tall)
  - Hver metode i en interface får sin egen "metode-indeks" for denne interfacen.
  - I forbindelse med descriptoren til hver klasse legges så:
    - En "interface-tabell" for hver interface denne klassen implementerer. Den er like lang som antall metoder i interfacen, og ved hver metode-indeks ligger den riktige virtuell-indeks til denne metoden i denne klassen.
    - En liten "interface-oversikt-tabell" over de interfacer denne klassen implementerer. Hver "linje" i denne tabellen representerer en implementert interface, og linjen inneholder interface-indeksen (stort tall), og en peker til den riktige interface-tabellen.
  - Når man gjør et kall: "pI.m(3)", der pI er typet med interfacen I, så skjer følgende:
    - Man finner objektet pI sin descriptor, og man *leter* (!) i dens interface-oversikt-tabell etter linjen med interface-indeksen til I.
    - I denne linjen ligger også den riktige virtuell-indeksen til m(...) i dette objektet, og vi kan gå til metodenpå vanlig måte.
    - Om man vil unngå letingen trenger man her en tabell med størrelse: (totalt antall klasser) X (totalt antall interfacer) med pekere til interface-tabeller

Typisk Byte-kode, ferdig til interpretering:

Merk: Også funsjonskodene (f.eks. *sipush, iconst_2 og goto*) er nå gjort om til tall-koder (mellom 0 og 255)

Java-program:

```
outer:
for (int i = 2; i < 1000; i++) {
   for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
   }
   System.out.println (i);
}
```

```
0: iconst_2
1: istore_1    // "i" har reladr 1
2: iload_1
3: sipush 1000
6: if_icmpge 44
9: iconst_2
10: istore_2  // "j" har reladr 2
11: iload_2
12: iload_1
13: if_icmpge 31
16: iload_1
17: iload_2
18: irem # remainder
19: ifne 25
22: goto 38
25: iinc_2, 1
28: goto 11
31: getstatic #84;  // Området for println() ?
34: iload_1
35: invokevirtual #85; // println()
38: iinc 1, 1
41: goto 2
44: return
```