



Javas klasse-filer, byte-kode og utførelse (og litt om C# sin CIL-kode)

Disse foilene er pensum

INF 5110, 30/4-2013,
Stein Krogdahl

Byte-koden for Java og .Nett (C#) kan leses her:

http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

http://en.wikipedia.org/wiki/List_of_CIL_instructions



Oversikt over Javas class-filer og byte-kode

- 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 (til byte-kode) kan dermed brukes på alle maskiner
- For effektivitet: Byte-koden blir nå oftest 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. Ellers ikke!



Hoppe over Javas class-filer og byte-kode?

- Finnes også Java-kompilatorer som hopper over hele byte-kode-steget
 - Får da en tradisjonell kompilator, som kan lage effektiv kode
 - Men det arbeides MYE med optimalisering av JVM'er. Neppe lette å slå!
 - Men det er mer problematisk å koble seg til Javas standard-bibliotek etc.
 - ORACLE saksøkte i 2010 Google fordi Android/Dalvik gjør det slik (men den eksekverer visst ikke fortere enn standard JIT-kompilering).
 - Google vant saken i mai 2012, men uklart (for meg) om dette er anket e.l.



Fra Java-program til utførelse

Hovedidé:

- Java-program → (vha. Java-kompilator, som det holder med én av!)
- class-filer med *tekstlig* byte-kode →
går til en Java Virtual Maskin (JVM)

En JVM kan så gjøre forskjellige ting:

- Opprinnelig var altså tanken at:
 - Den tekstlige byte-koden oversettes til et internt utførbart format (ikke standardisert), der alle navn er oversatt til binære lageradresser, relativadresser etc.
 - Så blir denne utført ved interpretasjon
- Men det vanligste nå er at:
 - Den tekstlige byte-koden blir oversatt av en JIT-kompilator, til maskinkode for den aktuelle maskin.
 - Denne blir utført på tradisjonell måte



Oblig 2 og Byte-kode

- Byte-koden likner mye på den lavnivå-koden dere oversetter til i Oblig 2
 - Men i Oblig 2 blir også "Loading" gjort samtidig som koden blir laget, og hver instruksjon blir lagret direkte i et binært format
 - Men i byte-kode blir altså alle navn beholdt på tekstlig form når kompilatoren legger koden på fil, som er det sentrale element i en klasse-filer.
 - Først når denne taes inn av loaderen blir den laget binær kode.

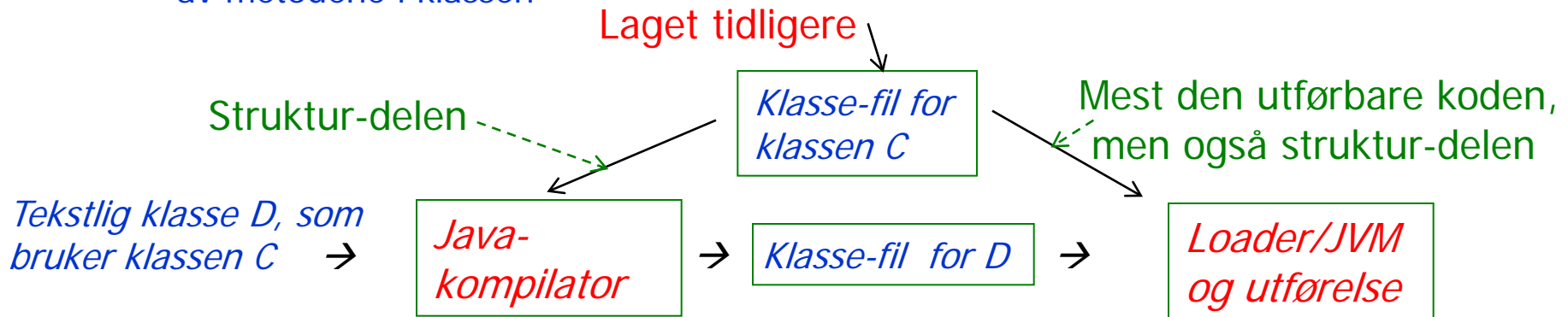


"Byte-kode" for C#

- Det som tilsvarer Javas bytekode hos C# og .NET heter CIL (Common Intermediate Language), men den kalles også ofte for bytekode.
 - CIL ble laget *etter* Javas bytekode, og kunne bruke erfaringene derfra
 - CIL er ikke beregnet på interpretering, *bare* for oversettelse videre til maskinspråk.
 - Det gjør at den ikke er begrenset av å være på et format som er lett å interpretere
 - Kan derfor også klare seg uten type-informasjon f.eks. i add-instruksjonen. Typen kan den se ut fra typen på stakk-verdiene.
 - CIL-koden ble fra starten av laget ikke bare med tanke på C#, men med tanke på at flere språk skulle kunne kompileres til den. Den ble altså designet litt "bredere".
 - Om man oversetter sitt språk til CIL, får man også automatisk grei tilgang til masse biblioteker etc.

Litt om Javas class-filer og byte-kode

- Klasse-filer inneholder både:
 - 1. Info om strukturen av klassen, med info om navn, variable, metoder, parametere, typer, etc.
 - 2. Den utførbare koden for hver metode i klassen (som byte-kode = sekvenser av byte-instruksjoner)
 - Disse to informasjonstypene ble tradisjonelt lagt på to forskjellige filer: For C og C++, på **.c-filer** (som inneholder utførbar kode) 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) kompiles: Da ser man bare på struktur-delen av klasse-filen
 - Når klassen skal loades: Da ser man «mest» 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, med navn, type, synlighet,
 - Byte-koden for alle metodene



Mer om formatet av Javas class-filer

- 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.



Oppsummering: 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 (men noen er satt av til fremtidig bruk)
 - Byte-koden har f.eks. en add-instruksjon for hver aritmetisk type
- Byte-instruksjonenes "adressefelt" (der dette trengs) angis ved fullt navn (tekstlig, dog som indeks), 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 antydnet 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 altså enten *interpretare* eller *oversette til maskinkode*
 - Men også om den vil interpretare vil den først gjøre bytekoden om til en *intern* bytekode-form der *alt er tall og fysiske adresser*
- Loaderen har en *verifikator*
 - Denne kan sjekke innkommende byte-kode. Mer senere.
- 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.



«Run-time descriptoren»

- JVM lager en "run-time descriptor" for hver klasse (ref. kap 6 og 7, m.m.)
 - Denne vil ligge fast under hele utførelse av programmet.
 - Alle objekter vil få en peker til descriptoren for sin klasse.
 - Descriptoren inneholder virtuell-tabellen for klassen, en peker til descriptoren for superklassen, etc.
- Dersom man har angitt debugging eller man bruker refleksivitet:
 - Da vil klasse-descriptoren også inneholde info om alle variable/metoder, deres navn, typer, osv.
 - Kanskje også metodenes kode i tekstlig Java-form (eller en peker til hvor denne kan hentes)



Loaderen i en Java Virtual Machine

- 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, fordi subklassen må vite hvilken relativ-adresse dens variabler skal starte på.
- Dersom interpretering:
 - Legger sekvensen av byte-instruksjoner (nå med tall både for funksjonsangivelse og adresser) ut i et passelig format (ikke standardisert)
 - Starter å interpretere dette
- Dersom oversetting: **Se neste foil**



Mer om loaderen

- 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 bytekode-steget
 - Får da en tradisjonell kompilator, som kan lage effektiv kode (og her var altså saken mellom Oracle og Google).
 - Men det er da mer problematisk å koble seg til Javas standardbibliotek etc.

Verifikatoren

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

```
0: iconst_2
1: istore_1
2: iload_1
3: sipush 1000
6: if_icmpge 44
9: iconst_2
10: istore_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;
34: iload_1
35: invokevirtual #85
38: iinc 1, 1
41: goto 2
44: return
```



Verifikatoren

- 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 typemessig 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 et Java-program har flere tråder:
 - Man setter for hver tråd vanligvis av et “stort” område til hver tråds stakk (typisk $\frac{1}{4}$ - 1 Mbyte).
 - Størrelsen av dette området kan vanligvis settes av brukeren.
 - Om det for stakken til en tråd trengs mer enn det avsatte området vil programmet stoppe (kanskje ikke i alle systemer??)



Tråder, og utførelse i (ekte) parallell

- Men merk: På grunn av “paging-systemet” vil man ikke bruke så mye *fysisk* memory.
 - Én page i memory-management-systemet er vanligvis 4 kbyte (altså 1/250 Mbyte), og man tar bare en ny page når nødvendig.
 - 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 aksessereres fra alle tråder
 - Det er satt av et bit i hvert objekt som angir om objektet er “låst”

Typisk Byte-kode, ferdig til interpretering:

Merk: Også funksjonskodene (f.eks. *sipush*, *iconst_2* og *goto*) skal tenkes 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
```