Programmering Fortsättning

Föreläsning 1

Joachim von Hacht

Föreläsningsanteckningarna innehåller den teori jag går igenom förutom en hel del bilder (t.ex. UML som jag kommer att rita på tavlan) och kod som visas i Eclipse (anges med BILD resp. KOD nedan). Bilder får ni rita av, kod finns i kataloger med matchande namn (f1.ep, f2.ep, o.s.v., importera till Eclipse). Övriga saker som tas upp men inte syns här är labbarna, tekniska tips m.m. Jag försöker ange engelska termer, sök dessa i boken eller på nätet om ni behöver mer info. Wikipedia rekomenderas.

Detta är version 1.3 av anteckningarna. Konstruktiv kritik tas tacksamt emot på hajo@chalmers.se.

Innehåll

1 Introduktion

Målsättning: Vi vill komma så långt det går, på nuvarnade nivå, för att producera korrekta, begripliga och gärna effektiva program[1]

---

2 Java och Java Specificationen

Vi kommer att använda språker Java. Hur språket fungerar beskrivs i The Java Language Specification, Third Edition (JLS) se t.ex. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html. Anges med JLS nedan. Dessutom finns specifkationen av Java's virtuella maskin. http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html anges med JVMS nedan.

[1] "Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning." //Unknown.

2.1 Stil

Java har en officiell beskrivning hur program skall skrivas (parenteser m.m.), se kurssida. Detta för att underlätta kommunikationen mellan programmerare. Lättare om alla gör på samma sätt.

OBS! Vi skriver alla program helt på Engelska, även kommentarer.

2.2 Språk och grundläggande terminologi

Kursspråket är Svengelska. Många saker har flera olika namn både på Svenska och Engelska fråga!
Exempel; Vi kan deklarera klasser. Dessa kan ha följande medlemmar (members);

- (inre) Klasser (återkommer)
- Interface (sällsynt, vi använder inte)
- Instance variables = instansvariabler = variabler = attribut
- Class variables = klass variabler = static variabler
- Instance methods = instansmetoder (funktioner i andra språk/sammanhang)
- Class methods = klass metoder = static metoder

Metoderna kan ha;

- Parametrar = argument. Vid deklaration; formella parametrar. Vid anropet: aktuella parametrar.
- Lokala variabler (bara synliga i metoden).
- En returtyp, typen för returvärdet (eller void om inget returvärde).

Vi kan även deklarera variabler av arraytyp = fält (ofta förekommand på svenska, skall försöka undvika).

- Obs! Att deklarationen inte skapar någon array, till detta måste vi använda ett initieringsuttryck (initializer) eller new.

3 Grundläggande problem vid imperativ programutveckling

3.1 Kompexitet

- Imperativ programmering innebär att variabler är namn på "lådor" (som dels håller (delar av) resultatet dels styr hur resultatet skall produceras).
- Mängden av alla värden för alla variabler (i alla klasser m.m.) kallas *programmets tillstånd* (lokala variabler räknas inte, däremot static variabler, återkommer...).
- Under programmets exekvering kommer tillståndet att förändras (variablernas värden ändras).
- Att på ett naivt sätt försöka ha en fullständig kontroll över tillståndet så att;
  - inte fel värde hamnar i låden (eller i fel låda, eller rätt värde i fel låda)...
  - fel *sorts* värde läggs i lådan (eller används för t.ex. en beräkning).
  - värdena läggs i lådorna i rätt tidsordning (inte skriver över värden) m.m....

  ...övergår mänsklig förmåga (även för relativt små program)!!! Komplexiteten blir för stor.

Observation Att utveckla imperativa program innebär att behärska komplexitet.

- Mycket i kursen handlar om att just detta.
- ∀koncept: På vilket sätt reducerar detta komplexiteten?
- Se t.ex. http://www.leepoint.net/notes-java/principles_and_practices/complexity/complexity_enemy.html (länk från kurssidan).

3.1.1 Code smell - Duplicate code

Code smell är en beteckning för erkänt dåliga lösningar, se t.ex. http://www.codinghorror.com/blog/archives/000589.htm. Duplicate code (också känt som Don't repeat yourself) innebär att man har samma eller snarlik kod (tecken för tecken eller logiskt) på flera ställen[2]. Otroligt dåligt.

- Vi får mer kod, ökar komplexiteten.
- Vi måste eventuellt synka de olika kodavsnitten.
- Svårt att hitta om väldigt mycker är identiskt, utom på "ett" ställe.

Observation Duplicate code ökar komplexiteten, får aldrig förekomma!

3.1.2 Refactoring

Refactoring innebär att man strukturerar om koden för att på något sätt få den bättre t.ex skapa en metod om man har

[2] Kan uppkomma vid s.k. Copy/Paste-kodning, man klistrar in kod.

samma kod på flera ställe. Inget skall ändras "utåt", beteendet är oförändrat. Märker man att koden stinker, ...kör refactoring (Eclipse har visst stöd).

Andra exempel på refactoring;

Namnbyten (bättre, begripligare), slå samman eller bryta ned metoder, slå samman eller bryta ned klasser, omorganisation av moduler, klasser (minska beroenden, se nedan), kod gemensam mellan klass A och B bryts ut och läggs i ny klass C, därefter referens till C i A och B. Kallas delegering eller komposition (composition)[3] .

3.2 Modifierbarhet

P.g.a. komplexiteten kanske vi gör fel eller missar något. Alternativt vi väljar bort vissa saker (som vi senare vill lägga till). Det innebär implicit att det måste vara möjligt att ändra i program.

Observation Program måste vara modifierbara (och därmed begripliga, ha en begriplig struktur (design)).

- ∀koncept: På vilket sätt bidrar detta till modifierbarhet?

KOD constants.*

3.3 Sidoeffekter

(side effects) I imperativa språk finns s.k. sidoeffekter. En sidoeffekt innebär att förutom att ett värde produceras sker något mer (tillståndet förändras). Exempel (metoden m returnerar int);

```
A a = new A();
int r = a.m(1) - a.m(1);
```

Är r = 0 efter detta? Om nej, svårt att utifrån koden resonera om programmet!

KOD sideeffect.*

Imperativa program är *inte* referentiellt transparenta (referential transparent) d.v.s. de ger inte samma resultat givet samma indata vid alla tillfällen[4]. Jämför funktionella språk...!

4 Strategier för imperativ programutveckling

Hur löser vi detta? Några strategier vi kommer att undersöka (forts. följande föreläsningar);

Språk Bygg in hjälpmekanismer i språket, t.ex. typsystem, stöd för information hiding och abstraktion, stöd vid exekvering, finns normalt i OO-språk (t.ex. Javas klasser), mer kommer...

Nedbrytning Dela upp problemet i mindre problem, lös dessa, sätt ihop dellösningar till lösning för hela problemet. Fungera *om* det går att dela upp problemet! Delproblemen får inte var alltför "intrasslade" i varann! Delarna måste enkelt kunna sättas ihop. Kallas ofta *modulär programutveckling* (modular software development).

Begränsa tillståndsrymden Om vi har färre tillstånd minskar antalet saker som kan gå fel.

[3] Finns många varianter men i denna kurs behöver vi inte vara så noga.

[4] Vissa språk skiljer på metoder som kan ändra tillståndet (command) och funktioner som inte gör detta (query). Se Eiffel.

Testning Kontinuerliga systematiska tester för att hitta och åtgärda fel (kallas ofta testdriven development).

Återanvändning Om vi kan använda tidigare utvecklad (och testad) kod utan att förstå hur den fungerar minskas komplexiteten (och utvecklingstiden).

Best practices Försöka konstruera programmet utifrån vad som erfarenhetsmässigt brukar leda till bra program (tyvärr finns ingen 100%-ig enighet).

Verktyg Ta hjälp av verktyg (utveckling, testning, kvalitetsbedömning,...). Datorer är bra på att hålla ordning på saker (cirkelresonemang..?)!

Semiformella resonemang Rent formella resonemang (bevis) har visat sig vara svåra att få till (som indikerat ovan). Semiformella resonemang kan dock ge en hel del (aldrig fel att tänka till).

5 Språk

5.1 Typsäkerhet

Genom att märka alla värden med en sort = en typ, kan kompilatorn lösa problemen ovan med "fel sort".

5.2 Typer

> "The Java programming language is a strongly typed language, which means that every variable and every expression has a type that is known at compile time. Types limit the values that a variable (§4.12) can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong typing helps detect errors at compile time." //(JLS 4)[5]

> "Compatibility of the value of a variable with its type is guaranteed by the design of the Java programming language,...all assignments to a variable are checked for assignment compatibility (§5.2), usually at compile time, but, in a single case involving arrays, a run-time check is made"//JLS 4.12

Medför att vi slipper till stora delar av problemet med "fel sort...".

Java är skapat för att vara typsäkert d.v.s. de operationer man skall kunna utför på datan i programmet skall bara vara de som anges av typen[6][7]. Lyckas dock inte helt, återkommer... OBS! Att typinformation används vid två olika tillfällen;

1. Vid kompilering (compile time). Den statiska, compile time, typinformtionen bara används vid själva kompileringen, efter kompileringen är den borta (finns t.ex. inga int, double,... i .class filen).

2. Vid körning (run time). Viss typinformation sparas och kan användas under

[5] Man kan se en typ som en mängd + operationer på denna.

[6] Hurvida Java verkligen är typsäkert eller vad typsäkert innebär råder det delade meningar om, se t.ex. http://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html.

[7] En synpunkt; "A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data."//Vijay Saraswat

körning (t.ex. vid instanceof). Kallas Runtitme type information RTTI.

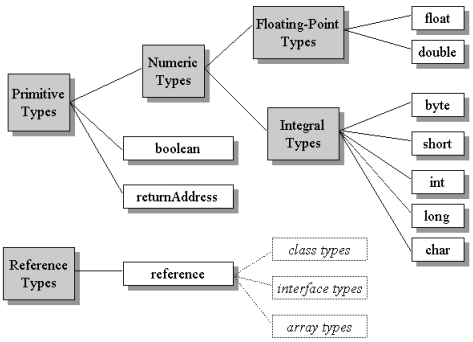
Observation Alla konstruktioner i Java vi i fortsättningen går igenom utgår från att typsäkerheten *måste bibehållas*.

Förklaringen till varför saker är som de är bottnar ofta i typsäkerheten.

5.3 Repetition: Javas typer

Java har två olika sorters typer[8];

- Primitiva, t.ex. char, int, double, boolean, m.fl. Dessa är inbyggda, vi kan inte ändra eller skapa nya primitiva typer.
- Referenstyper, klass-, gränssnitts och arrayer. Vi kan skapa nya typer genom att t.ex skapa nya klasser (en klass introducerar en ny referenstyp).



5.3.1 Som värde eller referens?

Att det finns två olika sorters typer får följder för betydelsen av språkliga konstruktioner.

[8]Null anges i standarden (JLS 3.10.7) som en egen typ med det enda värdet null.

- Värdesematik (by value). Inget delat tillstånd (kopier skapas). Primitiva typer.
- Referenssematik (by reference). Tillståndet kan vara delat. Referenstyper.

Återkommande frågeställning; Har vi värde- eller referenssematik här...?

BILD

5.3.2 Delade objekt

(Shared objects) Så fort vi har referenstyper kan innehållet i två olika variabler syfta på samma objekt! Komplexiteten ökar! Kallas alias-problem, shared state.

5.3.3 Metodanrop

Metodanrop sker i Java "by value" d.v.s. parametervärden från anropet kopieras till de formella parametrarna (gäller alltså även referenser, d.v.s. adresser kopieras).

BILD

5.3.4 Typen Object

I Java finns en speciell referenstyp, Object, kan hålla referenser till vilken referenstyp som helst, återkommer...

5.3.5 Typen för arrayer

JLS 10.7-10.8 Arraytyper anges genom att vid deklarationen lägga till [ ] efter elementtypen (som kan vara vilken typ som helst). Runtimetypen för en array är mystisk, anges som [+ bokstav t.ex. [D (double array) eller [L + klass + ';' [java.lang.String (string array);.

5.4 Implicita typomvandlingar

(JLS 5) Typsystemet behöver inte vara hur strikt som helst, att använda en long istället för en int innebär aldrig någon fara (mindre till större (i bytes, samma representation)). Det finns ett stort antal regler för vilka typer som får användas istället för en given typ, kallas *typkompabilitet.* Kompilatorn känner till dessa och utför vid behov *implicita typomvandlingar* (t.ex. vid parameteröverföring och returvärden). Exempel på godtagbara omvandlingar (widening).

- char->int, int->float, float->double, long->float
- Byte, Integer, Long, Float och Double går alla att omvandla till Number

Gör vi tvärtom riskerar vi att tappa infomation (narrowing).

Återkommande frågeställning; Är typerna kompatibla...? Går det att omvandla A till B utan att typsäkerheten bryter samman?

- Två orelaterade[9] referenstyper (t.ex. klasser) är aldrig typkompatibla.

5.4.1 Boxing och unboxing

Omvanling mellan wrapper-typer t.ex Interger (referens!) och motsvarande primitiva typ, int, är säkert och sker automatiskt. Exempel;

> "If p is a value of type int, then boxing conversion converts p into a reference r of class and type Integer, such that r.intValue() == p"//(JLS 5.1.7)

> "If r is a reference of type Integer, then unboxing conversion converts r into r.intValue()"//(JLS 5.1.8)

5.5 Explicita typomvandlingar

(typecast, cast) Vi kan säga åt kompilatorn att inte kontrollera en viss sak t.ex.;

```
//Reference to anything
Object o = ...
// Explicit typecast
String s = (String) o;
```

Vi gör en *explicit typomvandling*[10]. Eftersom vi här säger att o är av typen String så kontrolleras inte detta. Ansvaret är vårt! Förhoppningsvis stämmer det, annars exekveringsfel (ClassCastException).

- Typomvandlingar innebär alltid en risk! Sätter typcheckningen ur spel. Kan leda till runtim-fel. Undvik!
- Type casting mellan helt orelaterade referenstyper är inte tillåtet (enlig ovan).

KOD types.*

5.5.1 Omvandlingsfel

Omvandlingsfel påminner om typfel. Kan dyka upp vid användning av wrapperklasser t.ex.;

```
String s = “abc”;
//NumberFormatException
int i = Integer.valueOf(s);
```

KOD types.*

[9] Inget arvsförhållande, se vidare arv.

[10] Kräver att det finns ett arvsförhållande, återkommer...

- Alla typer kan omvandlas till String (primitiva, referenser och null-typen).

5.6 Exekveringmiljö

(runtime environment) Datorn kan ge oss annan hjälp också. Om man använder en exekveringsmiljö så kan denna under körning hjälpa oss med vissa fel. Java använder en sådan (Java Virtual Machine, JVM)[11]. JVM:en sköter minneshantering (skräpsamlar), indexeringkontrollerar arrayer, m.m. (tar tid och minne)[12].

Om vi t.ex. indexerar utanför en arrays gränser kommer JVM:en att generera ett undantag. Tanken är att felen skall visas så snabbt och tydligt som möjligt. Alternativet, utan kontroller, kan vara att fel uppstår men inte ger sig till känna förrän långt senare och då på ett svårtolkat sätt.

KOD arraycheck

5.7 Abstraktion

Man brukar skilja på funktionell- (metoder) och data-abstraktion. Båda går ut på att skapa begrepp på "högre nivå" (d.v.s. minska antalet detaljer). Java stödjer båda typerna.

5.7.1 Funktionell abstraktion

Bortse från hur något görs, fokusera på in- och utdata. Vad har vi? Vad vill vi ha? Används för att skapa metoder.

5.7.2 Dataabstraktion

Bortse från hur någor är uppbyggt, fokusera på hur man använder det (jämför en mikrovågsugn, Hur är den uppbyggd? Hur använder man den?). Används för klasser m.m.

Observation Abstraktion minskar (döljer) komplexiteten .

6 Unified modelling language (UML)

Vi kommer att använda UML för att på ett övergripande sätt kommunicera struktur (klassdiagram). Tas upp efter hand. Anges ofta med BILD (UML) i föreläsningsanteckningarna. Detaljer se t.ex. Wikipedia.

7 Nedbrytning

7.1 Beroenden

Ett central begrepp. Med beroenden menas i denna kurs att en klass A på något sätt refererar till en annan klass B (namnet B finns i A:s kod). B kan vara typ på attribut, parameter, lokal variabel m.m.

BILD (UML)

Om man har ett program där "allt" är berodende av "allt" har man en "big ball of mud" (spagettikod). Allt sitter ihop, ändrar vi på ett ställe så måste man ofta ändra på andra ställen, rättar men ett fel dyker ett annat upp, o.s.v.. En grundläggande strategi är att minska mängden beroenden.

Observation Beroenden ökar komplexiteten.

7.2 Information hiding

Saker som vi inte kan komma åt, eller känner till, minskar bereoenden (trivialt, vi *kan* inte använda dem). Vi strävar generellt

[11] Det är denna vi startar då vi skriver java i terminalen.

[12] I motsats till t.ex. C/C++ där man själv måste sköta dylikt.

efter att minska åtkomsten (eller området där något kan användas (scope)).

Observation Information hiding minskar komplexiteten.

Allmänt gäller;

- Dölj allt som går att dölja. I synnerhet sånt som kan påverka andra delar av programmet ifall de behöver ändras.
- Om det inte går att dölja helt, dölj så mycker som möjlig.
- Om det inte alls går att dölja gör det skrivskyddat.
- Om allt inte går att skrivskydda, skydda så mycket som möjligt.

7.2.1 Inkapsling

(Encapsulation) För att åstadkomma information hiding kan man använda inkapsling (encapsulation) d.v.s samla data och operationer på datan i en klass. Detta är dock ingen garanti för information hiding, att sätta alla fields till private och sedan skapa set/get för dessa innebär ingen information hiding (men kan ha andra fördelar).

7.3 The Single Responsibility Principle

Om en klass gör väldigt mycket kommer den troligen att ha kopplingar till många andra klasser. Många andra klasser (som använder denna) riskerar att behöva ändras och klassen riskerar att behöva modifieras då andra klasser modifieras. En grundläggande princip är en klass skall ha ett väldefinierat, väl avgränsat ansvarsområde (gäller även metoder och paket (se nedan)). Kallas The Single Responibility Principle (SRP). Motsatsen är en s.k. "code smell" ett klass som vet/gör för mycket (God object, Large class, Bloated class).

Observation Bryt ned stora saker, minskar komplexiteten.

Finns liknande begrepp:

- Separation of concerns.

Förutom beroenden blir det lättare att hantera klasser som gör en sak;

- De kan återanvändas (kombineras på nya/andra sätt).
- Blir lättare att förstå, lättare att hitta.
- Undviker problem med "blandade" objekt för datatunga klasser (substantivklasser, Bil, Konsult, Projekt,...) de representerar ett koncept. Viktigt om programmet skall kopplas mot en databas.

7.3.1 KISS

(keep it short (= small) and stupid (=simple)[13]) Relaterat begrepp.

> "The KISS principle states that simplicity should be a key goal in design, and that unnecessary complexity should be avoided."//Wikipedia

7.4 Moduler

Ett alternativ var alltså att dela upp det stora problemet (programmet) och lösa det bit för bit. Delarna kallar vi *moduler*. Uppdelningen görs enligt;

- Saker (klasser, submoduler) som är beroende av varann eller som är logiskt sammanhängande skall placeras i samma modul. Exempel; alla klasser som har med nätverket att göra läggs i en nätverksmodul. Hög samhörighet (cohesion).

[13] Keep it simple, stupid!

- Mellan modulerna skall det finnas så få beroenden som möjligt. Vill ha så fristående delar som möjligt! Låg koppling (low copuling).
- Vill inte ha cirkulära beroenden mellan moduler (om så sitter modulerna ihop alla är beroende av varann).

BILD

7.5 Paket i Java

(JLS 7) Klasser är i allmänhet för små delar för att motsvara en modul. I Java kan man använda paket (package) som moduler.

BILD (UML)

- Ger en större (än klass) logisk/funktionell enhet eller samlar ihop klasser på något sätt. t.ex s.k. modellobjekt (≈problemets substantiv t.ex. kund, bakaxel,...) eller hjälpklasser (utils t.ex. listor).
- Gör det möjligt att dölja klasser, "paket privata" (utelämna public före class).
- Delar upp den globala namnrymden (namespace). D.v.s. klasser kan heta samma sak bara de ligger i olika paket. Klassens *kvalificerade* namn är paket.subpaket.subsubpaket.....KlassNamn. Paketnamn skriv vanligen med *enbart* små bokstäver (klasser inleda som bekant med stora, därefter camelcase). Paketnamn får inte inledas med siffror eller vara reserverade ord.
- Om vi utelämnar accessspecificationen för ett fält i en klass (public, protected[14], private) så tillämpas "default access". Innebär att klasser i samma paket kommer åt fält, metoder[15]. Skall normalt aldrig användas (använd private)!

7.5.1 Teknik

- Paket motsvaras av mappar i filsystemet, heter klassen a.b.c.Main skall klassen ligga enligt (antag ett Eclipse project);

```
myproj.ep <-- Projektkatalog
 |-src <-- källkod
   |--a
      |--b
         |--c
            |--Main.java
```

- Klasser måste ange package först i filen d.v.s i Main,

```
// Main.java
package a.b.c;
```

- Normalt måste man i koden ange det kvalificerade namnet. För att slippa detta kan man ange import a.b.c.Main i andra klassfiler. Då räcker det att man skriver bara Main.

```
//OtherClass.java
import a.b.c.Main
:
Main m = new Main();
```

- Kompilerade klasser hamnar (skall hamna) i en egen pakethierarki[16]. För att hitta kompilerade klasser använder Java en "classpath" (sökväg utifrån katalogen man startas java). Denna

[14] Se vidare vid arv.

[15] ...default access, which is permitted only when the access occurs from within the package in which the type is declared. JLS 6.6.1

[16] Skall hamna i egen hieraki, kan sättas med flagga vid kompilering t.ex. Eclipse sköter detta.

sätt vid exekvering[17]. Antag katalogstruktur

```
myproj.ep    <-- står här
 |--bin      <--kompilerad kod
     |--a
       |--b
         |--c
            |--Main.class
```

Klassen Main (som innehålla mainmetoden) exekveras då med (-cp är classpath);

```
$ java -cp ./bin a.b.c.Main
```

D.v.s. starta i katalogen bin, sök därefter i a efter b och i b efter c. I c skall klassen finnas.

KOD edu.gu.hajo.demo.*

8 Sammanfattning

- Komplexiteten är huvudproblemet vid programutveckling.
- Vi går igenom ett antal strategier.
- Typer gör att vi slipper typfel, det hanteras av kompilatorn (vi får felen direkt, inte senare vid körning). Explicita typomvandlingar innebär alltid risker.
- JVM:en ger oss visst "runtime" stöd.
- Abstraktion är ett sätt att behärska komplexitet.
- Beroenden ökar komplexiteten.
- Information hiding miskar komplexiteten.
- Modulär programutveckling är ett sätt att skala komplexiteten, vi arbeter med delar av problemet. Konkret i Java så delar vi upp applikationen m.h.a. paket.
- Moduler motsvaras i Java av paket.

[17] Tidigare satts ofta en s.k. miljövariabel, rekommenderas inte.