Objektorienterad programmering fortsättningkurs

Version 0.1

Joachim von Hacht

9 januari 2012

Dessa anteckningar är samlade "område för område". Tyvärr kan man inte följa denna ordning då man går igenom materialet. Vi kommer alltså under genomgångar att hoppa runt. När vi är klara skall vi dock ha täckt det mesta. Förhoppningsvis blir det även på detta sättet enklare för er studenter att vid kursens slut få en överblick.

Marginalanteckningarna, Kod, hänvisar till kodexempel på kursidan. Namnet efter syftar på ett paket (koden är i form av Eclipse projekt).

OBS! Att detta *inte* är en bok, det är "anteckningar". Vissa saker upprepas nog på en del ställen (finns också i boken). Konstruktiv kritik och fel mottages tacksamt. *Språket är svengelska!*

Innehåll

1 Introduktion

1.1 Imperativ programmering

Problemet är...

```
x = ...;   // Oh,oh...
```

- 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).
- Under programmets exekvering kommer tillståndet att förändras (d.v.s. variablernas värden ändras).
- Att på ett naivt sätt försöka ha en fullständig kontroll över tillståndet så att;
  - inte fel värde hamnar i lådan (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).

Observation Att utveckla imperativa program innebär att behärska komplexitet (och i synnerhet tillståndet)[1]

- Mycket i kursen handlar om att just detta.
- ∀koncept: På vilket sätt reducerar detta komplexiteten?
- Se Complexity is your enemy.

[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.

1.2 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.m will return an int value
A a = new A();
// This should always be 0, or..??
int r = a.m(1) - a.m(1);
```

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

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

Vi kommer trots detta att göra några enkla försök att resonera utifrån koden.

1.3 Objektorientering

Objektorientering är (i vårt fall[2]) ett specialfall av imperativ programmering. Vi har fortfarande tillståndet att kämpa mot. Det objektorientering tillför är att övergången mellan "verkligheten" och programmet blir enklare. "Verkligheten" utgörs av ett antal samverkan objekt. Om programmet är strukturerat på samma sätt finns goda chanser att man lättare kan avbilda problemet, hitta en modell (i bästa fall en ett-till-ett mappning).

1.3.1 Domänmodell

Har ingenting med programmering att göra utan är bara en modell av ett problemområde. Vilka objekt ingår i problemet, hur samverkar dessa?

- Domänmodellen är lösningen på vårt problem!

Domänmodellen tas fram under den objekt orienterade analysen. Görs inte i denna kurs (ni får givna modeller). Viktigt att alltid försöka håll domän modell "ren" (annars grumlas lösningen, lösningen blir svårare att ändra/flytta).

1.3.2 Designmodell

Designmodellen är en påbyggnad av domänmodellen som gör det möjligt att implementera modellen i form av ett program.

- En viktig princip är att designmodellen inte skall ändra något i domänmodellen. Håll isär domän och design modell. Se vidare Open Closed Principle.

[2]Finns funktionell objektorientering m.m.

1.3.3 The Unified Modeling Language

Att kommunicera modeller (design) på kodnivå är för otymplig. Detaljerna skymmer idéerna. Kod: uml.*

Vi kommer att använda (mycket begränsade delar av) The Unified Modeling Language (UML) för att beskriva modeller och för att på ett övergripande sätt kommunicera design. Det vi använder är;

- Symboler för gränssnitt (abstrakt klass), klass och paket.
- Pilar för association, implementations- och gränssnittsarv samt beroende.
- Kvalificerade association och multiplicitet (vid pilarna skrivs t.ex. 0..n. visar antal objekt som är inblandade).
- Vi tar inte så hårt på ”is-a”, ”has-a” o. dyl (har mer med modellering att göra).

Finns en hel del på nätet (av skiftande kvalité).

1.4 Java och Java-specifikationen

För att utveckla våra OO-program har vi valt språket Java. Hur språket fungerar beskrivs i The Java Language Specification, third edition.

Dessutom finns specifikationen The Java Virtual Machine Specification (JVM).

- Detta är den slutliga referensen till Java. Ändpunkten för ditt sökande efter information. Här står allt!

1.5 Design

När vi har domänmodellen klar skall vi som sagt skapa designmodellen.

- Att ta fram en designmodell är en icke-trivial uppgift! Många olika villkor (ibland motstridiga) måste uppfyllas (samsas).

Det finns inga formler för hur man tar fram en bra designmodell men vissa principer och en hel del erfarenhetsmässigt bra lösningar finns. En inledande anmärkning.

- Design (formgivning) innebär att programmet ges en form (struktur). Vi skapar en struktur som gör det möjligt att begripa hur programmet är uppbyggt.

Inte helt ovanligt att program har en (mycket) dålig design, näst intill avsaknad av design. Sådan program kan (eller törs) t.ex. ingen ändra i (man kanske inte ens hittar vart man ev. behöver ändra).

1.5.1 Designprinciper

Vi kommer att titta på ett antal principer som förhoppningsvis hjälper oss att skapa väldesignade program (dock inga naturlagar).

1.5.2 Designmönster

Designmönster är mer konkreta än designprinciper. Ett designmönster (Design Pattern) är en återanvändbar lösning på ett återkommande design problem (ej språkberoende). Erfarna programmerare har i alla tider använt speciella tekniker men 1995 kom en bok med namnet:

> "Design Patterns: Elements of Reusable Object-Oriented Software (ISBN 0-201-63361-2) is a software engineering book describing recurring solutions to common problems in software design. The book's authors are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides with a foreword by Grady Booch. They are often referred to as the GoF, or Gang of Four."//Wikipedia

Mönstren kategoriseras efter: Creational (skapa objekt), Structural (skapa en löst kopplad, modifierbar struktur), m.fl

1.5.3 Design och arkitektur

Dessa begrepp är inte klart åtskilda. Vissa säger design andra arkitektur. Vi använder arkitektur för mer övergripande struktur. Vi kommer att se skillnaden i laboration 3 (en klient/server arkitektur).

1.6 Testning

Eftersom det är svårt att med formella metoder ta fram program blir testning en mycket viktig metod. Finns hela processer som bygger på testning t.ex. Test driven development;

> "...is a software development process [arbetsflöden] that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors [strukturerar om] the new code to acceptable standards. "//Wikipedia

Vi kommer inte att vara fullt så strikta. Vi utvecklar klasser och tester för dessa parallellt.

- Testning är icke-trivialt, att ta fram de testfall som behövs är en konst.

1.7 Övriga angreppssätt

Att utveckla imperativa program innebär som sagt att behärska komplexitet. Tyvärr finns ingen enskild "Silverbullet" d.v.s. garanterad metod som löser problemen. Förutom det ovan får vi använda ett batteri av angreppssätt.

Abstraktion Man skapar begrepp på "högre nivå" (d.v.s. bortser/dölja detaljer). Man brukar skilja på funktionell- (metoder) och data-abstraktion.

- Funktionell: abstraktion Bortse från hur något görs, fokusera på in- och utdata. Vad har vi? Vad vill vi ha? Används för att skapa metoder.
- Dataabstraktion: Bortse från hur något är uppbyggt, fokusera på hur man kan använda det (en lista kan t.ex. byggas på många olika sätt).

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).

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

Språkmekanismer Bygg in hjälpmekanismer i språket, t.ex. typsystem, stöd för information hiding och abstraktion, stöd vid exekvering (ett exempel är Java's runtime-stöd för indexkontroll i arrayer)

Kod: arraycheck

Tumregler (heurestiker) Försöka konstruera programmet utifrån vad som erfarenhetsmässigt brukar leda till bra program (som sagt finns ingen 100%-ig enighet).

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

2 Kodningsteknik

Några grundläggande saker.

2.1 Kodstil

Mycket inom programutveckling handlar om kommunikation, man arbetar tillsammans med andra. Att kommunikationen fungerar är otroligt viktigt. Ett sätt att underlätta kommunikation är använda en gemensam kodstil.

2.1.1 Java

Java har en officiell kodstil, hur program bör skrivas (layout, parenteser, namngivning, m.m.), se länk kurssida. Lättare om alla gör på samma sätt.

- Vi skall använda Java's officiella stil.

Undvik trista men eventuellt krävande småfel genom att alltid använda krullparenteser (även om det inte är strikt nödvändigt). Exempel;

```
if( ... ){  // Always braces...
...
}else{  // Yes, like this, one row
...
}
```

2.1.2 Formatering

Steve McConnell's Fundamental Theorem of Formatting;

> "good visual layout shows the logical structure of a program".

2.1.3 Namngivning

- Generellt: Lagom långa beskrivande namn.
- Skilj på singularis och pluralis (s på slutet).
- Använd standard förkortningar t.ex. n för number (nPlayers, antal spelare)
- Variabler inom ett (mycket) litet synlighetsområden kan ha korta namn t.ex. i loopar, där "i" och "j" är vanliga.

- Ofta används samma parameternamn och instansvariabelnamn i "set" -metoder eller konstruktorer. Instansvariabeln prefixas med this.

```
public void setValue( int value ){

    this.value = value;

}
```

2.1.4 Klasslayout

En grundläggande layout som vi skall använda;

```
/**
* Class comment
* @author
*/
class{
// constants
// variables
// public methods
// private methods
}
```

2.2 Hårdkodade värden

Inga hårdkodade värden. Följande skall aldrig förekomma.

```
// What is 16? What does it mean?
if( value < 16 ){
...
}
```

- Ersätt 16 med en konstant eller en enum (bättre).

2.3 Konstanter

Genom att ge värden namn (alltså namnge konstanter);

- ökar vi förståelsen.
- behöver vi bara ändra på ett ställe.

2.3.1 Konstanta variabler

Tidigare (eller i andra språk) använde man t.ex. heltal;

```
public final static int RED = 0;
public final static int GREEN = 1;
public final static int BLUE = 2;

int myColor = RED;
```

Här kan kompilatorn inte kontrollera om vi råkar tilldela myColor värdet 99 (en ogiltig färg!). För att lösa detta har man infört typsäkra enums.

2.3.2 Enum

En enum är ett specialfall av en vanlig klass. Ett bättre alternativ till konstanterna ovan.

- enum's används då man behöver ett litet antal värden av någon typ t.ex. veckodagar, färger, m.m.

```
// Better than above. All valid
// values belong to type Color
public enum Color {
   RED, GREEN, BLUE,...
}
```

- En enum-klass innebär att det automatiskt skapas exakt så många instanser som värden vi räknar upp. Varje instans är ett värde.
- För enum's gäller att == och equals ger samma resultat.
- Enum's används också i designmönstret Singleton, se detta.

2.3.3 Namngivna parametrar

Om en metod har ett begränsat antal parametrar, döp dessa genom att skapa en inre publik enum i klassen med metoden. Användaren väljer på så sätt mellan (namngivna) parametrar.

Kod: named-params

2.3.4 Strängmatchning

Ofta har man behov av en identisk sträng på flera olika ställen i ett program. Att hårdkoda strängen är ett dåligt alternativ. Alla fel ovan plus felstavningar som kan leder till runtime-fel. Kan använda enum's till detta också.

```
// No misspelling and no nonexisting color
label.setText(Color.RED.toString());
```

Kod: string-match

2.4 Data som strängar

Använd inte strängar för att representera komplex data.

```
// Bad
String person = 9812120354#pelle:pellesson:23:...
```

Omvandla alltid strängdata till korrekt datatyp. Komplex data blir klasser.

- Omvandling sköts typiskt av Converter-klasser, en för varje klass (metodet toObject och toString)

Kod: converter

2.5 Duplicerad code

Duplicerad kod[1] innebär att man har samma eller snarlik kod (tecken för tecken eller logiskt) på flera ställen. Uppkommer ofta då man klipper och klistrar kod. Otroligt dåligt.

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

Observation Duplicerad kod ökar komplexiteten, får aldrig förekomma!

2.6 Redundant kod

Självklart otillåtet. Vi skall ha exakt det som behövs, allt annat måste plockas bort, se The Single Responsible Principle .

2.7 Faktorisering

(Refactoring) Innebär att vi strukturerar om koden för att skapa en bättre struktur (eller begriplighet). Funktionaliteten påverkas inte. Om vi kör programmet innan vi faktoriserat och efter märker vi ingen skillnad. Det är de inre kvalitéerna som har förbättrats.

- Faktorisering skall användas kontinuerligt (aggressivt). Åtgärda omedelbart t.ex. konstiga namn, för stora metoder, klasser som ligger i fel paket eller blivit för stora...

2.8 KISS

En tumregel. Keep it short/small and simple[2].

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

[1] Också känt som Don't repeat yourself.
[2] Eller: Keep it simple, stupid!

2.9 Best practices

Best practices är tumregler som de flest programmerare ställer upp på.

- Ett exempel; Om en metod skickar en lista som resultat, returnera aldrig null, returnera en tom lista.

Finns mycket på nätet, får dock tänka till själv. Alla råd är inte goda råd!

2.10 Code smell

Inversen till best practises. En beteckning för erkänt dåliga lösningar, se t.ex. Coding Horror: Code Smell.

2.11 Effektivitet

Effektivitet är viktigt men ... vi börjar alltid med en väldesignad applikation. Har vi det är chanserna stora att vi kan gå in och åtgärda effektivitetsproblem.

3 Designprinciper

Det vi generellt vill undvika är s.k. ”ripple effects”. Att ändringar sprider sig (okontrollerat) genom hela programmet.

3.1 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.

Om man har ett program där “allt” är beroende 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.

- Detta gäller framför allt beroenden mellan moduler.

Observation Beroenden ökar komplexiteten.

3.2 Associationer

Associationer skapar beroenden. När man går över från domänmodell till designmodell kan man försöka dra ner på antalet.

- Behövs alla associationer i domänmodellen?
- Undvik dubbelriktade, försök att ersätta med enkelriktade eller kvalificerade.
- Undvik många-till-många associationer.
- Ersätt instansvariabler med lokala variabler (parametrar och/eller returvärden).

3.2.1 Dubbelriktade associationer

Exemepel: Många länder har haft många goda idrottsmän (dubbelriktad association). Men kommer vi t.ex. att starta med ”Karolina Klüft” och fråga vilket land hon kom från? Är det inte troligare att vi startar med ett land och frågar efter en idrottare? Kanske kan vi ersätta med enkelriktad?

Dubbelriktad association skapar dessutom cirkulära beroende mellan två klasser. Är detta egentligen *en* klass? En variant är att ersätta dubbelriktade associationer med följande;

```
// A needs B, B needs A

    public class A{
        B b;
    }
    public class B {
        // No A here
    }
    // Somewhere (accessible for B)
    Map<B, A> map = ...
```

3.3 Information hiding

Saker som vi inte kan komma åt, eller känner till, minskar beroenden (trivialt, vi *kan* inte använda dem). Vi strävar generellt efter att minska åtkomsten (eller området där något kan användas (scope)). Allmänt gäller;

- Dölj allt som går att dölja. I synnerhet sådant som kan påverka andra delar av programmet ifall de behöver ändras.
- Om det inte går att dölja helt, dölj så mycket som möjlig.
- Låt den om har ”kännedom” utföra det som skall göras.
- Detta gäller ”allt” och ”överallt”.

Observation Information hiding minskar komplexiteten eftersom vi får färre beroenden.

3.3.1 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 instansvaribler till private och sedan skapa set/get för dessa innebär troligtvis ingen information hiding.

3.4 new considered harmful

Så fort vi använder ”new” i koden skapar vi normalt ett beroende på en konkret implementation (se anonyma klasser). Bryter mot information hiding. Vi bör alltså centralisera tillverkning av objekt så att vi idealt bara har new på ett enda ställe i programmet.

Speciellt då vi vill ha tag i ett objekt som ”representerar” en modul (eller ännu tydligare: ett subsystem) vill vi bara att användaren har tillgång till gränssnittet d.v.s. det som skapar objektet skall returnera gränssnitttypen.

3.5 Designmönster: Factory method

Löser problemet med new ovan. Dessutom om det som skall returnerar är komplext så hjäper fabriken till att separera byggprocess från användningen.

Kod: factorymethod, standalonefactory

3.5.1 När en konstruktor räcker

- Klassen är typen (den implementerar inget interface, ingår inte i någon arvshierarki, används inte polymorft).
- Konstruktionen är enkel.
- Inga andra (ytterligare) objekt skapas i konstruktorn.

3.6 Synlighetsområde

En deklaration har ett visst synlighetsområde (scope), det område i koden där man kan använda identifieraren (namnet) (JLS 6.3).

- I Java gäller ungefär block-scope (mellan { och }) för deklarationer, finns många fall (JLS 6.3);
- "The scope of a declaration of a member m declared in or inherited by a class type C is the entire body of C, including any nested type declarations."
  - "The scope of a parameter of a method (§8.4.1) or constructor (§8.8.1) is the entire body of the method or constructor."
  - "The scope of a local variable declaration in a block (§14.4.2) is the rest of the block in which the declaration appears..."
- Synlighetområde för gränssnitt, enum och klasser är: Paket

3.7 Minimera synlighetsområde

Vi försöker att alltid minimera synlighetsområdet.

Kod: minscope

3.8 Programmering mot gränssnitt

Ett citat:

> "This so greatly reduces implementation dependencies between subsystems that it leads to the following principle of reusable object-oriented design : Program to an interface, not an implementation.
>
> Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class. You will find this to be a common theme of the design patterns in this book."//Design Patterns

Vi användet till exempel alltid

```
// Not ArrayList<String>
public void doit( List<String> s ){
}
```

Observation Genom att programmera mot gränssnitt döljer vi de konkreta implementationerna. Infomation hiding.

3.9 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 att det måste vara möjligt att ändra i program.

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

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

Exempel: Hårdkodning minskar modifierbarheten, programmering mot gränssnitt och modulära program ökar.

3.10 Granularitet

Med granularitet avses storleken på t.ex. klasser och metoder. Generellt gäller små klasser och små metoder, "one liners" (metoder med en kodrad är helt ok).

Följs The Single Responsible Principle så minskar risken för "svullna" klasser, metoder, m.m..

3.11 The Single Responsibility Principle

Om en klass gör väldigt mycket kommer det troligen att finnas många beroenden mellan klassen och de klasser som använder respektive används av klassen. Dessa klasser riskerar då att behöva modifieras då den aktuella klassen modifieras.

Vid modifieringar blir det svårare att passa in klassen, den är för speciell (enklare att passa in en tegelsten än en hel vägg).

En klass som gör mycket riskerar att bli svår att förstå och svår att testa. Eventuellt har man blandat ihop flera koncept i en klass, man har missförstått något.

En grundläggande princip är därför en klass skall ha *ett* väldefinierat, väl avgränsat ansvarsområde (på sin abstraktionsnivå). 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). Principen gäller på alla nivåer (och över allt);

- En variabel skall ha ett väldefinierat ansvar.
- .. så ock en metod...
- ...så ock en klass...
- ..så ock en modul...

Kallas även: Separation of concerns.

Observation Väldefinierade koncept minskar komplexiteten. Vi vet att här sker det *en* sak (finns *ett* syfte).

Exempel;

```
// Represent a player and a
// technical service, bad
// Clutter up domain model
player.saveToDatabase();
// Same as above
document.print();

// Better
database.save(player);
printer.print(document);
```

3.12 The Open Closed Principle

(OPC) Att gå in i existerande (testad) kod och ändra innebär alltid en risk. Dessutom grumlar vi kanske konceptet. OPC säger att vi skall förändra eller bygga ut applikationen genom att lägga till kod, inte ändra i gammal d.v.s. ”open” för ny kod men ”closed” för förändring av gammal. Kan ske m.h.a. polymorfism, mix-in, komposition (delegering) och/eller arv.

Ett vanlig scenario är att vi vill utöka en klass. Den utökade klassen måste ofta implementera API:et för klassen som skall utökas;

- Enklare med arv, kan bara lägga till (men arv är problematiskt som vi skall se).
- Med komposition får man implementera hela API:et i den utökade klassen, lite jobbigt men ger en säker implementering. De flesta metoder ”forward”:ar bara anropen till den ursprungliga klassen. Kod: opc
- Genom att använda polymorfism skyddar vi mot förandringar t.ex. genom att använda grässnittstyper som parametrar. Vi behöver inte ändra inte i metoden eller klassen. Skapa en ny klass som implementerar grässnittet. Metoden fungerar med objekt av den nya typen.

3.13 Dependency Inversion Principle

(DIP) I en skiktad desig skall inte klasser på hög nivå ha direkta beroenden på klasser på låg nivå, de skall ha ett gränssnitt mellan (alltså programmering mot gränssnitt). Kod: dip

- Om lågnivåklassernas implementation ändras så påverkar det ev. högre liggande nivåer. Orimligt att detaljer skall på verka övergripande design. Beroendena går åt fel håll. Det är lågnivå delarna som skall ändras om högnivådelarna så kräver (alltså beroenden åt andra hållet).

3.14 Interface segregation principle

Gränssnitt skall inte vara för stora. Kan tvinga klasser att implementera metoder som inte används. Dela upp gränssnitt efter koncept.

4 Testning

> "Program testing can be used to show the presence of bugs, but never to show their absence!" // Edsger Dijkstra

4.1 Enhets- och regressions-testning

Två exempel på tester. Finns en mängd olika tester (varianter) förutom dessa;

Whitebox/Blackbox-testning Om man har tillgång till implementeringen respektive inte. Vi gör whitebox-testning eftersom det är vi själva som utvecklar koden och har full tillgång till allt.

Enhetstestning (unit testing). Test de minsta delarna i ett program (klasser, paket, moduler). I vårt fall klasser.

Regressionstestning (regression testing). Alla tester sparas. Vi tillägg/ändringar skapas nya tester för ändringen. Denna + alla tidigare tester måste bli godkända. D.v.s. ändringen får inte leda till att någon existerande test misslyckas.

4.2 Testdriven utveckling

Vi arbetar ungefär så här;

- Skriv tomma klasser (tills det går att kompilera), Sätt returvärden till 0 eller null om det krävs;
- Börja med att implementera de mest fristående klasserna (ofta på lägsta abstraktionsnivån).
- Implementera en metod i taget, börja med de mest oberoende (de som kan testas utan att blanda in andra metoder). Inte säkert att det alltid går. I så fall utveckla några beroende metoder och testa.
- För varje klass vi implementerar skapas en motsvarande testklass (en Java klass).
- I testklassen skriver vi testmetoder så fort någon/några metod(er) är klara. En testmetod för varje implementerad metod i klassen under test (eller för några beroende metoder).
- Fortsätt tills en hel klass är testad och klar. I nästa steg implementerar vi klasser som är beroende av den senast utvecklade o.s.v.

Det är oerhört mycket rationellare att ha testerna nedskrivna och exekverbara än att försöka komma ihåg, skriva kommentarer eller testa för hand. Att testa går mycket smidigt på detta sätt.

4.2.1 JUnit

Vi använder ramverket JUnit 4. Ramverket finns färdigt I Eclipse. Vi skapar en testklass (vanliga Java-klass) som anropar klassen vi vill testa (class under test). Testklassen "körs" av JUnit och vi får en rapport.

- Var noga med namn på metoderna i testklassen Använd lååååånga beskrivande). Exempel: testIfLastNodeContainsReferenceToHead(...)

Vi skiljer på testkod och applikationskod. Detta görs genom att använda två olika "source folders" (källkodsfoldrar) i projektet (src för applikationskod och test för testkod). I testfolderna skapar vi samma paketstruktur som i src. Innebär att vi kommer att kunna testa paket-privata klasser.

Eclipse kan skapa testklasser (New...). I dessa skapar vi metoder. Följande annoteringar används för testmetoder;

```
// Somewhere in a test class...
// Testmethod must be public void, no parameters
@Test
public void doSometest(){

    // ...

}

// If we expect an exception
@Test(expected=IllegalArgumentException.class)
public void doSometest(){

    // ...

}
```

Testerna är tänkta att vara automatiska, ingen mänsklig interaktion skall förekomma (testen lyckas eller misslyckas)[1]. För att avgöra om testen lyckades eller ej använder vi en klass från JUnit, Assert. Typiskt efter något anrop till en metod i klassen under test;

```
// In test method
int result = classUnderTest.methodTotest(testData);
// Yes or no, no human intervention
Assert.assertTrue( result == KNOWN_VALUE );
```

[1] I synnerhet inte System.out.println.

4.2.2 Klass under test

Vad och hur skall vi testa klassen vi skrivit?

- Normalt testas publika icke-triviala metoder (ej set/get) och invarianter.
  - Använd representations invarianter.
  - Typiska att testa är extremfall; Tom lista, tomma strängen, full lista, ...första, sista, ...största minsta,...udda/jämt antal, ...
  - Försök hitta ”sjukt” konstiga fall...
- Att skriva tester är icke-trivialt!

Om metoden är void måste vi ha tillgång till någon metod som kan avläsa eventuell tillståndsförändring. Man kan lägga till en get-method som endast används vid testning (ingår absolut inte i något gränssnitt).

Om man behöver testa privata metoder kan det tyda på dålig design. Finns det en annan klass inuti den vi testar?

Om man verkligen har behov av att testa privata metoder använder man specialla hjälpklasser eftersom man inte vill ändra i klassen vi testar!

Tyvärr blir det lite för avancerat för oss. Vi får ändra private till public och sedan återställa.

4.2.3 Fixturer

För att testerna skall vara tillförlitliga måste datan de använder vara korrekt. Man kan initiera datan som använd m.h.a. @BeforeClass och @Before

- @BeforeClass, anges på en statisk metod i testklassen. Metoden körs en gång innan övriga metoder körs (finna även @AfterClass, för att ev. återställa något).
- @Before efter @After kan anges för valfri metode i testklassen. Innebär att metoden körs före varje testmetod.

4.2.4 Testsviter

För att sätta samman alla utvecklade klasser skapar man en testsvit (ytterligare en Java-klass). Genom att köra sviten kör man alla tester i följd. D.v.s. har hela tiden kontroll att inget helt oväntat fel dyker upp (i annan klass). Kod: aTest.*

4.3 Code coverage

Om testerna skall ha något värde måste vi veta att en stor del av koden har exekverats och därmed testats, t.ex. olika grenar i if-satser, körs loopen?, o.s.v. Kallas code coverage. Det gäller att utforma testerna så att detta uppfylls!

4.3.1 ECLEmma

ECLEmma är en plugin till Eclipse som undersöker hur stor del av koden som har körts av testerna. Innan man kör ECLEmma skall man köra JUnit-testen.

4.4 Testning och avlusning

Man kan avlusa testerna (och därmed klassen under test). Kombinationen testning och avlusning är mycket effektiv. Betydligt snabbare och enklare än att sitta med hela programmet (avlusning av hela programmet behövs iofs också). Se upp med kod som är svår-avlusad.

```
// Hard to debug
a.doOne().doTwo().doThree()....getResult();

// Also hard to debug
a.doIt(b.doIt(c.doIt(...))).getResutl();
```

5 Modulära program

Vi vill att ett program skall bestå av lätt identifierbara delar där varje del har ett tydligt ansvar, ett modulärt program.

Idealt utvecklas programmet som ett antal väl testade moduler vilka man slutligen sätter samman till ett helt program. Om modulerna är tillräckligt oberoende kan de utvecklas parallellt. Är de tillräckligt generella kan de återanvändas.

5.1 Moduler

De delar ett program är uppbyggt av (på översta nivån) kallar vi *moduler*. En liknelse. Om programmet är en bok så: "modules are the chapter in a book"

- Internt skall modulerna vara väl sammanhållna, kallas high cohesion. Åstadkoms genom att kod som implementerar ett koncept placeras i samma modul.

  Exempel; GUI:et implementeras av ett antal GUI klasser. Läggs i samma modul. Vi utgår ifrån att "allt" i en modul är beroende.

- Mellan modulerna skall det vara få beroenden, low coupling.

- Vill inte ha cirkulära beroenden mellan moduler. Om så sitter modulerna ihop, alla är beroende av varann. Finns verktyg för att testa cirkulära beroenden.

  - Viktigt att komma ihåg när man kör vertygen är att ta bort ev. testklasser (annars skapas cirkulära beroenden pga dessa) . Förvirrande!
  - Om vi har cirkulära beroenden får vi analysera vad det beror på och åtgärda (lågnivå använder högnivå, klasser ligger i fel paket, dela upp/slå samman klasser,.. d.v.s. refaktorisera).

- Domänmodellen håller vi för sig (i en eller flera moduler).

- Klasser i applikationslagret, se nedan, håller vi också separat (i en eller flera moduler).

- Dessutom finns ofta moduler typ "util" (utilities) för klasser som inte direkt passar i någon annan modul (alltså inte ett koncept).

5.2 Skiktade design

(Layered design/architechture) Förutom att dela upp programmet i (vertikala) moduler delar vi det i olika horisontella skikt (lager) utifrån abstraktionsnivåer. Vi försöker använda följande skikt (uppifrån och ned)

- Presentationskikt (GUI), ansvarar för interaktionen med användaren (om det inte är en mänskligt användare blir det ett UI eller kanske MUI).
- Applikationslager, ett tunt lager som samordnar och använder domänmodellen och tjänster från infrastrukturlagret för att utföra de kommandon som användaren ger. Om lagret är tillräckligt tunt kan det slås samman med GUI siktet.
- Domänlagret, här finns domänmodellen. Alltså lösningen till problemet!
- Infrastruktur, här finns olika tjänster, datalagring, nätverk, .. Används av framför allt applikationslagret.

Högre nivåer gör (metod) anrop mot lägre. Högre nivåer kan hoppa över en underliggande nivå (undvik).

En viktig princip är att lägre nivåer aldrig gör "uppåt"-anrop" (anrop av metoder i skikt ovanför). För att skicka data uppåt används indirekta metoder (se Observer mönstret).

5.3 Subsystem

Ett subsystem är en en modul som utåt har ett funktionellt sammanhållet gränssnitt t.ex. en nätverksmodul, en databasmodul,... (en funktionell enhet).

- Normalt skall dessa vara tillståndslösa (utåt).
- Återfinns i infrastruktur skiktet.

5.4 Bibliotek

Bibliotek är (samlingar av) moduler som utvecklats speciellt med tanke på återanvändning. Java innehåller ett stort antal standardbibliotek (kallas ibland för API'n, Application Programming Interface). Vi kommet t.ex. att använda ett antal bibliotek i laboration 3 (vi vill inte skriva nätverkskoden själva).

5.5 Designmönster: Facade

Facade är ett desingmönster speciellt lämpat för subsystem. Subsystemet representers som en modul med en enda publik klass. Den synliga klassen implementerar något gränssnitt (som alltså ligger utanför modulen). Kod: facade

5.6 Designmönster: Adapter

Om man har moduler som inte passar ihop kan man låta en adapter anpassa gränssnittet. En adapter använda alltså som en koppling mellan saker som inte riktigt passar ihop.

- I Java används begreppet lite annorlunda. Med adapter avses en klass som innehåller default-implementationer av ett gränssnitt t.ex. MouseAdapter (tomma metoder). På så sätt behöver man bara implementera de metoder man är intresserad av.

Kod: adapter

5.7 Moduler i Java (Paket)

Den konkreta implementationen av moduler i Java görs m.h.a. paket (JLS 7). Klasser är i allmänhet för små för att motsvara en modul. Paket ger oss följande möjligheter (risker);

- Ger en större (än klass) logisk/funktionell enhet eller samlar ihop klasser på något sätt. t.ex modellen eller hjälpklasser (utils t.ex. listor).
- Gör det möjligt att dölja klasser, "paket privata" (utelämna public före class). Infomation hiding.
- 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.....KlassNar Paketnamn skriv vanligen med *enbart* små bokstäver (klasser inleda som bekant med stora, därefter camel-case). Paketnamn får inte inledas med siffror eller vara reserverade ord.
- (risk) Om vi utelämnar accesspecificationen för ett fält i en klass (public, protected[1], private) så tillämpas "default access". Innebär att klasser i samma paket kommer åt fält, metoder[2]. Skall normalt aldrig användas (använd private)!

[1] Se vidare vid arv.

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

6 Exceptionella händelser

Detta bygger på bokens kapitel 11. Jag har utökat med olika designaspekter. Begreppet exceptionella händelser förkortar vi till "undantag".

Undantagshantering dyker upp mer eller mindre direkt då man börjar ta fram designmodellen.

6.1 Olika kategorier av fel

Se bok.

6.2 Undantag vid exekvering

Viktigt, stort, svårt och omtvistat område! Två vanliga synpunkter

- Undantag skall användas för "speciella" fel (för undantag...hmmm).
  - Att en fil saknas är ett undantag. Vi/programmet har inte gjort något fel. Något utanför programmets ansvarsområde har gått fel. Rimligt att signalera med ett undantag (IOExeption).
  - Vi söker efter index för ett element i en ArrayList. Elementet saknas, jaha. Inget konstigt med det, kan hända. Inget undantag, returnera "saknas"-värde (-1).
  - Om vi däremot söker ett element med ett visst index och ger ett orimligt index (t.ex. -1) genererar t.ex. ArrayList ett undantag (returnerar inte null). Motiveringen: Felaktigt index är ett "oväntat" (felaktigt) användande av ArrayList, alltså undantag. Vanlig att kasta undantag för "felaktig" indata.
  - Man får själv utifrån detta försöka avgöra hur metoder skall bete sig.
- Undantag skall inte användas för programflöde. Programflöde styrs med villkor.
  - Ett avskräckande exempel;

```
try{
   Iterator i = collection.iterator();
   // Bad, should have used i.hasNext()
   while( true ){
     Foo foo = (Foo) i.next();
     ...
```

```
        }
    }catch( No SuchElementException e){
    }
```

– ...men vissa språk använder undantag för programflöde. Iteratorer i språket Python genererar ett StopIteration undantag då man når slutet på en lista?!

6.3 Manuell undantagshantering

Använd returvärden som felsignal.

```
// BAD! Lage part of code is
// checking
// Also mixed with logic
r1 = s.call()
if( r1 != null ){
  r2 = r1.call();
  if( r2 != null ){
     r3 = r2.call();
     :
  } else {
     :
  }
} else {
  :
}
```

- Bryter mot separation of concerns, vanlig kod mixad med felhanteringskod.
- Kan det vara svårt att hitta vettiga "fel"-värden (kan iofs lösas med s.k. flaggor, lite code smell).
- Olika programmerare gör på olika sätt ingen standard...
- Hur gör man för att lägga till/ta bort undantagshantering?

6.4 Javas undantagshantering

Java har en standardiserad (inbyggd) undantagshantering (exception handling)

6.4.1 Automatiskt genererade undantag

> "When a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an exception." //JLS 11

6.4.2 Undantagsklasser

Se bok. Viktigast är skillnaden mellan Checked och Unchecked-exceptions.

6.4.3 Att själv generera undantag

Se bok. Detta är en viktig del av failure atomicity.

6.4.4 Specifikation av undantag

Exception och alla subklasser till denna *utom* RuntimeException är s.k. checked exceptions;

> "A compiler for the Java programming language checks, *at compile time*, that a program contains handlers for checked exceptions, by analyzing which checked exceptions can result from execution of a method or constructor."

Målsättningen med checked exceptions har varit att tvinga programmeraren att ta ställning till tänkbara fel. Kod:checked

Unchecked exceptions: Används för programmeringsfel (sådana som vi är ansvariga för). Det skall smälla så högt och snabbt som möjligt (undantagen fångas inte)! Unrecoverable exceptions. Vi använder typiska objekt från klasserna NullPointerExceeption, IllegalArgumentException och IllegalStateException.

Checked exceptions: För fel möjliga att hantera (programmet skall inte terminera). Recoverable exceptions. En databasfråga som resulterar i att man inte hittade det eftersökta skall inte göra att programmet avslutas (kan ge checked SQLException (ej i denna kurs))!

6.4.5 The Exception debate

Java är ett av få språk som har checked exceptions (det enda?). Detta har skapat en livlig debatt. Är dylika en bra ide?

- För;
  - Man måste att hantera felen (Inte: vi fixar det senare....).
  - Det syns direkt vilka metoder som kastar undantag.
  - Ger en viss dokumentation.
- Emot;
  - Kan exponera implementationsdetaljer (om man inte packar om felen till högre abstraktionsnivå).
  - Instabila metodsignaturer.

- Om en klass implementrar många gränssnitt med mycket undantag blir det grötigt...
- I vissa fall svårläst kod med massor av catch-grenar.
- Om man vet att inget undantag kommer att kastas (det blir t.ex. aldrig EOF = End of file) måste man ändå hantera.

6.4.6 Att fånga undantag

Se bok.

Kod: _finally

6.4.6.1 Två varianter av catch och finally

finally-delen är mycket användbar eftersom den är garanterad att köras. Kan användas för att frigöra resurser (annars resource leak) och eventuellt för att återställa objekt till väldefinierade tillstånd. En metod kan välja mellan följande;

- If a method throws all exceptions, then it may use a finally with no catch
- If a method handles all of the checked exceptions that may be thrown by its implementation and the finally block throws the same exceptions as the rest of the code (which is common with java.io operations.)

Kod: catchfinally

6.4.7 Exekveringsförlopp vid undantag

6.4.7.1 Anropsstacken

Då ett Java program körs och en metod anropas används en anropsstack. All data som behövs vid anropet t.ex. parametrar m.m. sätts samman till en s.k. stackframe och push:as på anropsstacken. Då metoden är klar avläser man ev resultat och popar stackframe:en[1].

Antag att metod a anropar metod b som anropar metod c. Följande händer.

```
- a pushas på stacken anropar b
- b pushas anropar c
- c pushas.
- c klar popas, programmet fortsätter med b
- b klar popas, fortsätt med a
- a klar popas.
```

6.4.7.2 Icke-lokala hopp

Då ett undantag uppstår skapas en instans av Throwable. Det normala flödet avbryts och programmet vandrar upp genom anropsstacken (unwind the call stack). Hanteras inte undantaget någonstans kommer programmet att avbrytas (med en felutskrift)

[1] Stacken gör att programmet kan komma ihåg vart det skall fortsätta.

Vi kan alltså få ett s.k. icke-lokalt hopp (vi kan ju hoppa "igenom" en eller flera metoder).

- För att hantera undantaget (exception handler) används konstruktionen try...catch(..)..finally

  "When an exception is thrown, control is transferred from the code that caused the exception to the nearest dynamically-enclosing catch clause of a try statement that handles the exception."

- Om JVM:en hitter en felhanterare (catch-clause) levereras undantagsobjektet till denna. Vilken felhanterare som anropas beror på undantagobjektets klass (använder instanceof).
- Då en felhanterare har körts klart räknas felet som "hanterat" och det normala programflödet fortsätter (direkt efter felhanteraren).

6.4.7.3 Kritik

"The reasoning is that I consider exceptions to be no better than "goto's", considered harmful since the 1960s, in that they create an abrupt jump from one point of code to another. In fact they are significantly worse than goto's:

They are invisible in the source code. Looking at a block of code, including functions which may or may not throw exceptions, there is no way to see which exceptions might be thrown and from where. This means that even careful code inspection doesn't reveal potential bugs. They create too many possible exit points for a function. To write correct code, you really have to think about every possible code path through your function. Every time you call a function that can raise an exception and don't catch it on the spot, you create opportunities for surprise bugs caused by functions that terminated abruptly, leaving data in an inconsistent state, or other code paths that you didn't think about."//Joel Spolsky

6.4.8 Effektivitet

Undantagshantering är tänkt att användas i extrema (sällsynta) fall. JVM:en är inte optimerad för att hantera sådana sällsynta saker. Det är därför mycket resurskrävande att skapa, kasta och fånga undantag (ännu en motivering till att inte använda detta för programflöde).

6.5 Abstraktionsnivåer för undantag

Undantaget skall presenteras på rätt abstraktionsnivå, ett lågnivåundantag, t.ex. nätverket fungerar inte, bör översättast till något relevant för den nivå som hanterar undantaget.

```
// To low level

    interface MyRoutPlanner {
      // Hmm,...why IOException, ..
      public plan( RouteData r ) throws IOException;
    }
    // Better (exception should explain why)
    interface MyRoutPlanner {
      public plan( RouteData r ) throws RouteException;
    }
```

Kallas exception translation. Kod: translation

6.6 Exception tunneling

Ett sätt att hantera invändningarna mot checked exceptions, man tunnlar undantaget. Paketera en checked exception i en klass som ärver RuntimeException (alltså unchecked) och re-throw. Fånga sedan i lämplig klass. Kod: tunneling

6.7 Central undantagshantering

Man får försöka separera ut undantagshanteringen så mycket det går. Eventuellt skapa en speciell ExceptionHandler-klasser. Så fort ett undantag dyker upp skickas det till ExceptionHandlern. Centraliserad visning av t.ex. dialogrutor är också önskvärt, man skall inte behöva gå in i GUI koden för att ändra ett felmeddelande.

6.8 Undantag, kontrakt och arv

- Om man lägger till throws (checked exception) för någon metod, kommer kompilatorn alltså att kräva att användaren hanterar undantaget.
- Throws räknas med vid overriding. Om subklassens metod har en throws så måste superklassen ha det med;
  - kovarians gäller för undantagsklasser, d.v.s. en overridden metod i subklassen kan kasta en subklass till undantagsklassen i metoden i superklassen[2]. Kod: inherit

[2]Case 1: Overriding method throws runtime exception - Allowed Case 2:Overriding method throws subclass of the exception type declared by the parent method - Allowed Case 3: Overriding method throws base class of exception type declared by the parent method - NOT Allowed Case 4: Overriding method throws totally unrelated compile time exception replacing parent method throws clause - NOT Allowed

6.9 Generiska undantag

Fungerar inte.

6.10 Best practises

Mycket att tänka på...!

- Aldrig tomma catch block (lägg alltid till åtminstonde e.printStackTrace());

```
// Bad, exception swallowing..
try {
   // Code
} catch( AnyException e ){
  //Never empty!!!!
}
```

- Vid re-throw gäller (don't swallow nested exceptions);

```
try { ...
    } catch (Foo e) {
      // Include e!
       throw new Bar(e)
    }
```

- Huvudregeln är att man skall fånga undantaget där man kan göra något åt det. Oftast är det inte i samma metod som undantaget uppstod, t.ex. om FileNotFound uppstår långt ner i ett subsystem. Undantaget måste i extremfallet ända upp till GUI:et (eller kontrollobjekten i MVC) där användaren informeras och utifrån detta gör ett nytt val (d.v.s. undantaget får propageras uppåt).

- Om man anger en checked exception i ett gränssnitt tvingar man eventuellt användaren att hantera undantag i onödan, implementationen kanske inte kastar något undantag! Undvik!

```
// doIt() throws in IA
// but not in A
// A implements IA
A a = new A();
IA ia = new A();
// Same code  but...
ia.doIt()//...must handle
a.doIt()//...must NOT!
```

- Använd inte catch(Exception e), kommer att fånga RuntimeException (vill vi inte).

- Börja med subklasser först i catch-delarna för att få så exakt felhantering som möjligt.

- Se upp med nästlade try..catch.

7 Att hantera tillstånd

Tillståndet är en av de grundläggande orsakerna till komplexiteten. Uppenbar strategi:

- Vi försöker minimera tillståndsrymden.

7.1 Metoder utan sidoeffekter

Metoder som inte har sidoeffekter kallas funktioner[1]. Funktioner;

- ger alltid samma resultat för samma indata.
- kan kombineras fritt utan att oförutsedda saker händer.
- är mycket enklare att testa.

Placera så mycket logik som möjligt i funktioner

En annan iakttagelse. Operander förändras typiskt inte. Antag att vi skall addera två matriser a och b;

```
// 1).Add a to b. add state changing method
Matrix a, b;
a.add(b)
// 2) add returns new objec, no state change.
Matrix c = a.add(b);
```

I första fallet har vi en tillståndsförändring, a är numera summan av a och b. Betydligt bättre att låta add returnera ett nytt objekt som står för summan av a och b, d.v.s. operanderna är oförändrade (troligen bättre med helt lös klass som ansvarar för addition t.ex. som Math).

7.2 Begränsa antal objekt

Görs med t.ex. Singleton mönstret eller olika typer av ”pooler”.

7.3 Designmönster: Singleton

Garanterar att det bara finns ett enda objekt av typen. Ger dessutom en enkelt sätt att komma åt objektet (en global accesspunkt).

Kod: singleton.*

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

7.4 Värde och entitetsobjekt

Värdeobjekt Ett objekt som har tillstånd och vars identitet beror på (delar av) tillståndet (motsvarar ett värde).

Exempel: Antag att vi har en adress med med gata och ort. Vilket adressobjekt vi har spelar ingen roll det är bara tillståndet som är intressant. Alla som bor på adressen kan dela objektet.

Entitetsobjekt (dataobjekt) Ett objekt där inte tillståndet ensamt kan avgöra identiteten.

Exempel: Antag att vi har en order med köpare, summa och datum. Vi kan inte byta ut ett orderobjekt mot ett annat även om de har exakt samma tillstånd (det kanske kom två likadana från samma kund samma dag)[2]. Finns även t.ex. Thread, FileOutputStream m.fl.

7.5 Final

Final *i samband med variabeldeklaration* innebär att värdet inte kan ändras (en konstant).

- Konstanter (konstanta instansvariabler) måste ges ett värde vid deklaration eller...
  - ...sättas från konstruktorn.
  - Om man har referensvariabler kan inte referensen ändras men däremot det refererade objektet. Se upp!

Observation Final minskar komplexiteten, vi kan inte ändra denna del av tillståndet. Utgå från final och gör variabler skrivbara vartefter.

7.6 Icke-muterbara objekt

En klass som ges ett väldefinierat starttillstånd som sedan inte kan ändras kallas för icke-muterbart objekt (ett konstanta objekt, immutable object).

OBS! Att vi syftar på *logiskt* icke-muterbart. Omgivande objekt upplever objektet som oföränderligt men internt kan det förändras (sällsynta fall, stor försiktighet)[3].

- Icke muterade objekt är lätta och säkra att använda (även trådsäkra, se senare...), använd om möjligt (best practises).
- Ett icke-muterbart objekt fungerar utmärkt som ett värdeobjekt[4].
- Ett problem är att man ev. måste skapa många värden[5] eller nya värden (hela tiden). Kan leda till effektivitetsproblem.

[2] Därför har alla order normalt ett ordernummer, d.v.s. de är värdeobjekt.
[3] Tänkbara fall: Debugging, cache:ing.
[4] Enum fungerar precis på detta sätt.
[5] new är en av de mest tidskrävande operationerna i Java.

- Å andra sidan kanske man kan återanvända samma värde (riskfritt att t.ex. dela på värdet).

Observation Icke-muterbara objekt minskar komplexiteten eftersom tillståndet är konstant.

7.6.1 Implementation

Konkret i Java skapar man en klass som saknar muterande metoder (set-metoder). Alla attribut är privata och final och sätts i konstruktorn.

- Dessutom får man se upp med representationsexponering, arv, kanonisk form, serialisering och trådar, se vidare dessa.

Kod: immutable

7.6.2 Exempel: String och StringBuffer

Standardklasser för stränghantering: Den ena muterande en andra icke-muterbar.

- Icke-muterbar[6]: String, operationer på typen innebär att nya objekt skapas och data kopieras.
- Muterbar: StringBuffer används då vi "arbetar" med en sträng (lägga till, ta bort, m.m.). Omvandling StringBuffer till String sker m.h.a. toString().
- Se upp med +-operatorn för strängar, operatorn "skalar inte" (don't scale) d.v.s. den blir för ineffektiv (i tid) om man har för många, långa strängar.

Kod:string

Finns en ännu något snabbare men inte trådsäker (återkommer) klass StringBuilder.

7.7 Begränsat muterbara objekt

Ibland krävs en parameterlös konstruktor. Vi kan i dessa fall inte ge objektet ett väldefinierat starttillstånd. Se initiering.

7.8 Tillståndslösa objekt

En klass som saknar attribut är tillståndlös. Tillståndslösa objekt representerar ren funktionalitet utan sidoeffekter, kallas ibland för "pure" classes.

Ingen större idé att skapa tillståndslösa objekt använd statiska klasser eller Singleton.

7.8.1 Statiska klasser

Tillståndslösa objekt ersätts (implementeras) ofta som rent statiska klasser d.v.s. alla metoder är static och kontruktorn privat.

Observation Tillståndslösa objekt/statiska klasser minskar komplexiteten.

[6] Dessutom är klassen final innebär att inga subklasser kan skapas, återkommer...

7.9 Representation

Hur ett konkret eller abstrakt koncept avbildas i programmet. Exempel;

- Ett hisssystem kan representeras med en array (med lika många platser som det är våningar i huset).
- En karta kan avbildas som en graf av nodobjekt där varje nod kan kopplas till andra noder (eller som en Map).

Vanligen finns ingen given representation, man *väljer* en representation, ett designbeslut.

- Representationen har ett tillstånd.

7.9.1 Byte av representation

Byte av representation räknas som en acceptabel (tillåten) desingförändring (transformation), d.v.s. bytet skall inte påverka resten av programmet.

Att kunna byta representation utan att övriga delar påverkas är en mycket viktig egenskap.

Representationen skall vara dold för användare av klassen/modulen. Infomation hiding/dataabstraktion)!

7.9.2 Implicit representation

I en del fall kan representationen vara implicit.

Exempel: Antag att vi arbetar med stora glesa matriser. Då kan t.ex. alla nollvärden representeras implicit. D.v.s. bara element skilda från noll sparas verkligen.

7.10 Representationexponering

(Representation exposure) Om en kontruktor får en referens eller om en metod returnerar en referens till representationen uppkommer s.k. representationexponering. Innebär att en annan klass kommer åt och kan ändra i representationen (tillståndet). Bryter mot inkapsling, ökar komplexiteten, förstör möjlighet att resonera utifrån representationsinvarianter, se nedan.

- Lösning: Defensiva kopior (eller icke-muterbara objekt, återkommer...)
- Av effektivitetsskäl tillåter man ibland representationsexponering t.ex. Iteratorer (slipper kopiering).

7.11 Defensiv kopia

Defensiv kopia Man kopierar (djupt) referensvärden, parametrar till konstruktor och/eller returvärden och använder dessa d.v.s. man ändrar från referenssematik till värdesemantik. Kod: defcopy

7.12 Representations-invarianter

En representationsinvariant (RI) är en specifikation för utvecklare som beskriver vilka tillstånd den *valda representationens* får befinna sig i[7]. Då man arbetar med RI känner man till alla implementationsdetaljer.

- De tillåtna tillstånden beskrivs med ett eller flera predikat (som alltid måste vara uppfyllda (sanna)). Predikaten skrivs som kommentarer i klassen.
- Exempel från en Trie-klassen (se laboration 1). Vad som än händer så måste föräldern till roten vara null (annars har vi inget stopp då vi vandrar uppåt)

```
/**
* RI
* this.root.parent == null  //Must always be true
*
```

7.12.1 Verifikation av RI

Utifrån RI kan man föra induktiva resonemang (bevis)[8].

- Om RI etableras av konstruktor (eller init-metod (som anropas))...
  - ...och ∀muterande metoder: RI upprätthålls...
  - ...så kommer representationen alltid att befinner sig i ett tillåtet tillstånd.
  - Anm: RI kan kan vara falsk under den tid metoden körs med efter måste den vara sann (problem kan uppstå vi call-backs, m.m.)

Någorlunda hanterligt i den praktiska programmerinsprocessen. Bara initiering och muterande metoder behöver granskas. Kod:ri

7.13 Failure atomicity

Innebär att gör alla operationer som kan leda till undantag innan man ändrar i representationen. Representationen skall alltså vara intakt även då ett undantag har uppstått.

- Tillståndsförändringar görs alltid så sent som möjligt!

```
// Failure atomicity
public void doIt(int i )throws SomeException{

    doSomething(i);  // Possible exception
    doSomethingElse(i);// Possible exception
    // Now ok
    repr = ....  // Change representation
```

[7] Bereppet kommer från verifikation av datastrukturer.
[8] Jmf. Induktionsbevis, basfall, induktiva fall

```
}
```

Ett vanligt scenario är också kontroll av inparametrar;

```
// First check params
public void doIt(int i )throws SomeException{
    if( i < 0 ){
        throw new IllegalArgumentException(...);
    }
    repr = ....  // Change representation
}
```

7.14 Designmönster: State

Om man har ett antal tillstånd där utfallet av olika operationer beror på det aktuella tillståndet t.ex. Kod: state

```
if( state == a ){
    if( action == x ){
    ...
    }else if (action == y ){
    ...
    }else if( action == z ){
    ...
} else if( state == b ){
    if( action == x ){
    ...
    }else if (action == y ){
    ...
    }else if( action == z ){
    ...
}
...
```

Så kan detta ersättas med designmönstret state.

8 Initiering

Eftersom vi hanterar tillstånd är det naturligtvis mycket viktigt att vi sätter ett väldefinierat starttillstånd för allt. Konstruera alltid objekt så att de har ett väldefinierat tillstånd[1]! Se även initiering vid arv.

8.1 Final

- Om en instansvariabel är final måste den initieras (i konstruktor), annars kompileringsfel.

8.2 Konstruktorer

Konstruktorer är den naturliga mekaniskmen för initiering. Se upp med trådar och arv, se dessa.

8.3 Initieringsblock

8.3.1 Statiska

Rent statiska (tillståndslösa) klasser kan behöva initieras (dock inte tillståndet)

```
// Static initializer
static {
   // init things here
   // code...! loops...[2]
}
```

8.3.2 Instans

Anonyma klasser saknar konstruktor. Kan initieras med ett initieringblock.

```
// Instance initializer
{
   // init things here[3]
}
```

[1] Tyvärr är detta i praktiken inte alltid möjligt eftersom Java i vissa fall kräver en default-konstruktor.
[2] Except return-statement.
[3] Implicit kopierat in i varje konstruktor.

8.4 Initieringsordning

Initering sker inligt följande

1. Initiering av klassvariabler och exekvering av static initializers (d.v.s saker gemensamma för alla objekt initieras). Sker i "textually apperant order".

2. Då objekt skapas initerieras instansvariabler och instance initializers exekveras. Också i textually apperant order. Därefter körs konstruktorn[4].

- Delobjekt instansieras och initieras allra först. Exempel;

```
class A {
  //...then B
  private B = new B();
  //First...initialize C
  private static C = new C();
  :
}
```

Kod: init

Problem kan uppstå om vi vill ge en statisk variabel ett värde som produceras av en instansiniterare. Finns en hel del restriktioner, se JLS 8.3.

8.5 Andra initieringsmetoder

Tyvärr kräver en hel del i Java tekniker att det finns en parameterlös kontruktor. Vi kan då få problem med att ge objektet ett väldefinierat starttillstånd. Verkar inte finnas något riktigt bra sätt att lösa detta på. En variant är att skapa en init metod. Mindre bra, undvik.

Kod: init.method

8.6 Dependecy injection

Ett modernare sätt att initiera objekt är dependency injection. Kräver att man använder ett "ramverk". Ett ramverk består oftast av ett bibliotek man lägger till. I biblioteket finns typisk en klass som fungerar som en fabrik. Med hjälp av denna skapar man objekt (new kan inte användas). Objekten initeras av ramverket då de skapas. Var och hur initiering skall ske anges med annotationer t.ex.

```
public class MyClass {
@Inject
IMyInterface i; // Framework will inject an object
...
}
```

[4]Man kan se det som om initiering av instansfält och initierare körs först i konstruktorn, därefter kommer resten av kontruktorn.

Injektionen sker efter allt annat (kan inte använda varibeln i konstruktorn). Två kända DI ramverk är (se vidare reflection);

- Google Juice.
- Context and Dependecy Injection for Java, CDI (en Java standard)

Kod: guice, cdi.* (obs, behöver extra bibliotek, ligger i mapparna).

9 Klasser och gränssnitt

Detta är generella designsynpunkter på klasser och gränssnitt.

9.1 Klass och typ

Det är operationerna (metodsignaturerna) som är det karakteristiska för (objekten ur) en typ. Däremot är inte implementeringen. Man kan tänka sig två olika klasser med exakt samma operationer men med olika implementationer.

Detta leder till att man istället för att introducera en typ m.h.a. en klass (en implementation) introducerar den m.h.a. ett gränssnitt (en specification, ett slags kontrakt).

- En klass introducerar en typ men en typ behöver inte introduceras med en klass, ...kan använda ett gränsnitt i stället (eller enum).
- Ofta är det bättre att introducera typer m.h.a. gränssnitt.
  - Gränssnitt passar mycket bra för s.k. ”mix-in” typer d.v.s. då man vill utöka en bastyp med lite extra funktionalitet (t.ex. Comparable)
  - Existerande klasser kan lätt anpassas så att de implementerar ett gränssnitt (läggt till implements och skapa metoderna). Mycket svårare att anpassa så att den passar in i en arvshieraki, Kanske flera klasser berörs, isf måste superklassen flyttas uppåt så att den ärver en gemensam superklass (enkelt arv) och alla mellanliggande tvingas ärva.

9.1.1 Markerinterface

I Java finns en del konstiga gränssnitt som inte används för att specifiera en typ d.v.s. gränssnitten innehåller inga metoder (de räknas som en typ ändå av Java). Kallas ”marker interface” t.ex. Serializable.

Används för ”metadata”. Klasser som interagerar med klassen som implementerar ”marker interfacet” kan läsa av detta och bete sig därefter (t.ex. skriva till en ObjectOutputStream om klassen implementerar Serializable, annars exception).

- Mycket dålig användning, vi skall inte...

9.2 Variabler

Som sagt föredra alltid lokala variabler. Variabler skall aldrig skuggas eller dylikt i subklasser.

9.2.1 Klassvariabler

Klassvariabler får bara vara final (och referera till icke muterbara objekt). Annars delat tillstånd mellan alla objekt! Dåligt! Används till;

- Konstanter, static final (vanligt och ofta ok).
  - Finns ofta Constants-klasser, alla applikationsglobala konstanter i en klass som döps till Constants (innehåller bara public static final ...).
- Se Grundläggande kodningsteknik.

9.2.2 Instansvariabler

Instansvariabler skall vara privata (i undantagsfall protected, se Arv).

- Instansvariabler sätta i konstruktor eller sätts/avläses med set/get metoder.

Repetition Variabler är inte polymorfa, deklarerad typ gäller!

> "The following distinction between invoking instance methods on an object and accessing fields of an object must be noted. When an instance method is invoked on an object using a reference, it is the class of the current object denoted by the reference, not the type of the reference, that determines which method implementation will be executed. When a field of an object is accessed using a reference, it is the type of the reference, not the class of the current object denoted by the reference, that determines which field will actually be accessed."

9.3 Metoder

Dela upp metoder i grupper:

- accessor: Avläser eller beräknar ett resultat.
- mutator: Förändrar tillståndet.

Blanda inte dessa, låt aldrig mutatorer returnera domändata!

9.3.1 Klassmetoder

Används till;

- Funktionssamlingar t.ex. Math. och utility-klasser. Tillståndslösa klasser.
- Vissa design mönster, t.ex. Factory method, Singleton.
- Överanvändning kan vara en code smell, ev. för mycket procedurell programmering.

Kod: _static

- Ingen polymofism.
- Klassmetoder kan inte override:a instansmetoder.

Att mixa anrop mellan klass och instans-variabler kan leda till kompileringsfel. Kod: _static.mix

9.3.2 Instansmetoder

Det normala. Används och ger polymoft beteende. Se mer vid Arv.

9.3.2.1 Parametrar som utvärden

Skall unvikas men behändiga i vissa fall. Om parametrar används som utdata skall detta dokumenteras. Kod: recurser

9.3.2.2 Listor som returvärden

Skickar aldrig null som resultat för ”tom” lista. Skicka en tom lista (objekt)

10 Arv och design

10.1 Metodsignatur

(JLS 8.4.2) Metodsignatur = signatur (signature) = metodnamn + parameterlista (antal parametrar, ordningen på dessa och typ för respektive). *Inget annat ingår* (speciellt inte returtypen).

- Två metoder med samma signatur inom samma synlighetsområde (t.ex. i samma klass) är aldrig tillåtet (compile time error). Kan aldrig ha t.ex

```
// Same signature, bad
class A {
  public int doIt( String s ){...}
  public String doIt( String s ){...}
}
```

- Vid override gäller: Samma signatur *och returtyp*. Kod: override
- Vid overload: Bara samma metodnamn. Kod: overload

10.2 Gränssnittsarv

(implements) ingen kod delas. *Grundläggande princip i OO!*

```
class A implements IA {

      :

}
```

- Vi skiljer på ”kontraktet” och implementationen.
- Gränssnittsarv reducerar beroenden (som vi sett tidigare).
- Finns inga kända nackdelar med denna typ av arv.

En liten anmärkning är att gränssnitten är väldigt svåra (omöjliga?!) att förändra. Många applikationer kanske är beroende av gränssnittet, se vidare Abstrakta klasser.

- Ett interface kan ärva ett annat (med extends), blir ändå gränssnittsarv. Kod: inherit.iface

10.3 Implementationsarv

(extends) innebär alltså att kod delas mellan super och subklass (kodåtervinning). En finess men inte något absolut krav för OO.

```
class B extends A {

      :

}
```

- Att B ärver (extends, derives, refines) A innebär att i varje *objekt* av typ B ingår ett delobjekt av typ A.
  - Alla B är A men inte tvärt om d.v.s. B är en delmängd av A (A $\supset$B). Kan kännas lite bakfram, B är ju en "större" klass än A (innehåller lika mycket eller mer än A).
- Arv skapar en partiell ordning mellan super och subklasser, skrivs (ofta) A :> B (A super, B subtyp). Arvsrelationen är reflexiv ($\forall$A; A :> A) och transitiv (A :> B $\wedge$ B :> C => A :> C).

10.4 Fragil base class problem

"The fundamental problem of inheritance"

Implementationsarv skapar mycket starka beroenden mellan super och subklass. Ändringar eller implementationsdetaljer i superklasssen kan lätt förstöra eller få oförutsedda konsekvenser i subklasser. Några exempel;

- Om en superklass tillämpar "self-use" d.v.s metoder (och/eller konstruktor) använder internt egna metoder som kan override:as (en implementationsdetalj) kan detta leda till problem i subklassen (som kanske overrid:ar precis en sådan metod).
  - Kan dessutom skilja mellan olika versioner av superklassen.
- Antag att vi har infört en massa kontroller i subklassen (med override) som gör att man bara kan lägga till objekt som uppfyller vissa kriteria. I nästa version finns en ny metod i superklassen som också lägger till. Den metoden finns nu som ärvd metod i subklassen men utan kontroller! Subklassen blir korrupt!
- Antag att vi bara lägger till nya metoder i subklassen (ingen override). Vad händer om superklassen i nästa version;
  - har fått en metod med samma signatur men med annan returtyp (...subklassen kompilerar inte)?

  - har fått en metod med samma signatur och returtyp (...plötsligt har vi override som troligen inte beter sig som vi vill eftersom metoden kom till efter det att vi skapat subklassmetoden).

- Representationsbyte är som sagt en tillåten operation...(dataabstraktion)
  - ... men om subklassen känner till basklassens implementation.
  - ...och implementionen ändar i basklassen! Stora risker.

OBS! Att allt ovan sker utan att vi rört subklassen (som kanske fungerade tidigare)! I praktiken måste super och subklasser utvecklas parallellt. För oss är detta inget problem eftersom vi har tillgång till alla kod (labbarna).

Kod: inherit.fbc

10.4.1 Designa för arv...

Om vi skall tillåta arv måste klassen designas med mycket stor noggranhet (alltså klasser som skall användas publikt av många utvecklare).

- Klassen måste dokumentera effekterna av override (alltså avslöja hur saker är implementerade, dåligt). Detta binder oss till en implementering (eftersom subklasserna har använt infomationen). Standard är att i Javadoc inleda med "This implementation...", se t.ex. AbstractCollection.remove(Object o).
- Konstruktorer (eller metoder som fungerar som sådana, clone(), readObject()) får inte använda metoder som kan override:as. Metoden körs innan objektet är färdigkonstruerat. Se även kanonisk form.
- Finns ännu fler detaljer...

10.4.2 ...eller förbjud

Om man inte avser att använda implementationsarv (inte vill uppfylla allt ovan) så, förbjud. Använd final

"A class can take these general policies with respect to subclassing, in order of increasing liberality :

- disallow it completely , declare the class as final.
- allow it, but disallow all overrides, declare all methods as final.
- allow it, and permit some overrides, declare some methods as non-final.
- require it, declare some methods as abstract" (återkommer med abstrakta klasser).

Som sagt detta gäller inte med samma tyngd om vi har full kontroll över all kod (gäller främst bibliotek).

10.4.3 När använda implementationsarv

Implementationsarv är acceptabelt när vi har kontroll över hela hierarkin. Eliminera duplicerad kod (flytta till superklass) i små arvshierakier i samma modul. Se även abstrakta klasser. Vissa saker skall aldrig användas arv

- Normal ärver man aldrig behållare.
- Att ärva roller (person, student, lärare...) är fel (dessa kan växla).

10.5 Kompositionsom kontra arv

"Favor object composition over class inheritance." //Design Patterns

Problemen ovan samt ytterligar saker nedan gör att man helt enkelt rekommenderar komposition istf arv. Dessutom viktig skillnad;

- Arv är statiskt, vi måste koda om för att ändra.
- Komposition är dynamisk, kan ske runtime (bättre).

Komposition vs delegering: Vid delegering skickas this med då man forward:ar.

10.6 Designmönster: Decorator

Decorator bygger på komposition.

```
class A {

    B b = ...
    public A( B b ){
        this.b = b;
    }
    public void doIt(){
        ...  // possible something extra that B lacks
        b.doIt();  // forward to wrapped class
    }

}
```

Om alla inblandade klasser har samma gränssnitt kan man sätta samman dessa (se t.ex. javas io klasser)

```
// Decorated C with some extra features from A and B
IMyInterfacs i = new A( new B (new C ));
i.doIt();  // All methods in C present
```

10.7 Subklass och subtyp

Java betraktar en subklass som en subtyp, d.v.s. står det extends så är det en subtyp (en rent syntaktisk konstruktion). *Men subklass och subtyp är inte samma sak!*

- Subtypen måste alltid i alla sammanhang kunna ersätta supertypen utan att något i programmet ändras/bryter ihop.
- Subklass säger bara att kod delas. Det är lätt att åstadkomma en subklass som inte är utbytbar mot sin superklass, *trots att de har exakt samma publika gränssnitt* (metoder). Subklassen är alltså inte en riktig subtyp !

För att en subklass (extends) skall fungera som en subtyp måste den uppfylla/respektera superklassens kontrakt.

10.8 Liskov substitution principle[1]

Säger vad som krävs för att en subklass skall fungera som en subtyp. Principen är en semantisk relation;

> "Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T."

Principen ställer ett antal krav på metoder i subklasser.

- Kovarians för returtyper och kontravarians för parametrar. Kovarians för nya undantag.
- Måste upprätthåll superklassens invarianter.
- Får inte ha striktare förvillkor än metoden i superklassens. Får inte kräva mer än superklassen.
  - Om super-metoden krävar att parameterna ligger mellan 0-24 får inte metoden i sub-klassen kräva att den ligger mellan 0-12 (ganska uppenbart)
- Får inte ha svagare eftervillkor än superklassens metod. Får inte lova mindre.
  - Om superklassmetoden t.ex. tar bort alla element med en viss egenskap i en lista så får inte subklassen bara ta bort det först påträffade (också gansk uppenbart).

Kod: inherit.liskov

10.9 Representationsexponering

Inget hindrar en subklass att skapa en public-metod som exponerar reepresentationen (en protected variabel) i superklassen.

- En icke-muterbar klass kan alltså bli muterbar om man slarvar med implementationen

Kod: inherit.expose

[1] Barbara Liskov, berömd mjukvaruexpert på MIT.

10.10 Instanceof kontra getClass()

Konstruktionerna instanceof och getClass() används för att få tag i Runtim Type Information (RTTI).

instanceof avgör typkompabilitet för tilldelning och typomvandling, d.v.s. om true så är typomvandlingen säker. OBS! *Ger alltid true för subklasser till aktuell klass.* Ger false för null. Returtyp boolean.

getClass() Ger runtimetypen för klassen, inget annat (returtyp Class).

10.11 Polymorfism och RTTI

Man skall undvika RTTI, ersätt med polymorfism. Exempel;

```
// Tedious, hard to change.
// Have a tendecy to crop upp all over the application
if( o instanceof A ){
   ((A)o).doIt();
}else if( o instanceof B ){
   ((B)o).doOther();
}else...
```

Det ovan kallas bl.a. switch-code smell. Vi riskerar att behöva ändra i (potetiellt många) selectionssatser då nya klasser tillkommer (eller gamla försvinner). Bättre, låt B ha en metod doIt som gör det doOther skulle ha gjort[2];

```
// Much simpler
o.doIt()
```

10.12 Tekniska aspekter på implementationsarv

10.12.1 Initering

Det vi tidigare har sagt gäller men dessutom... När en subklass skapas anropas alltid, implicit, åtminstoner en superklass konstruktor (Object). Sker i omvänd ordning d.v.s. om A :> B :> C så;

- Då C instansieras: C's kontruktor anropar B's som anropar A's. När denna är färdig körs B's färdigt och därefter C's.

[2]Effective C++, by Scott Meyers :
"Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself.

- Explict anrop av superklasskonstruktor görs super(...). *Måste göras först i kontruktorn*. Kan behövas då konstruktorn tar argument.
- Se upp med default-kontruktor! Om konstruktorn är overloaded i superklassen finns ingen implicit default-kontruktor (subklassen kan då inte ha default-konstruktor). Kod:inherit.init

10.12.2 Instansvariabler

Protected skall minimeras, ger ökat synlighetsområde!!

En subklass skall aldrig ha en variabel med samma namn som en protected variabel i någon superklass (finns två variabler med samma namn används olika beroende på override, förvirrande, felkälla...)! FIXA KODEXEMPEL

10.12.3 Instansmetoder

Overriding skall användas då vill ha ett beteende baserat på typ av objekt (runtime typen). T.ex. olika saker händer för olika objekt som implementera samma interface. Kan alltså ersätta selektion (if).

Använd alltid @Override!

Overloading skall användas då;

- Samma generella funktionalitet gäller för samtliga metoder (samma sak skall göras). Exempel; max av två saker, t.ex. int, long, ...(elegantare att kunna ha samma namn på samma funktionalitet)
- Samma metod kan ta olika antal parametrar. Den med flest parametrar är basmetoden övriga har förbestämda (default) värden för vissa parametrar.
  - Overloading av konstruktorer mycket vanligt (kan anropa en konstruktor från en annan. Anropas this(...))
- Alla overload-metoder i samma klass, aldrig utspridda i arvshierarkin.

10.12.4 Varians

Ko- och kontravarians (co- contravariance) gäller bl.a. parametrar och returtyper i samband med overriding (referenser inte primitiva)[3].

- Vid overriding i subklasser accepterar Java kovarianta returtyper (returtyperna har samma arvsförhållande (samma riktning, co-) som klasserna. Antag A :> B och C :> D . I A finns metod; public C get() och i B overridden till public D get() .

[3]En typ regel eller typ konverteringsoperator är kovariant om typförhållandena bibehålls (:>). Om de vänds blir det kontravarians i övrigt invarians.

```
A a1 = new A();
A a2 = new B();
C c1 = a1.get();
//c2 runtime är D
C c2 = a2.get(); //Säkert
```

- Parametertyper i Java är invarianta (måste vara exakt samma typ, inte sub eller superklass)[4].
- Användning av kovarianta returtyper eliminerar explicita typomvandlingar. Kod: inherit.covariance

10.12.5 this

Om vi anropar en metod på ett object så syftar this alltid på runtime typen (polymofism!), även om kod från superklassen exekveras, *this byter aldrig typ*! Superobjektet är en del av subobjektet. Kod: inherit._this

10.12.6 Arrayer

- För arrayer av primitiv typer gäller invarians d.v.s. int[] är inte en subtyp till long[].
- För arrayer med referenstyper gäller om S :> T så S[] :> T[]. Exempel Object[] :> String[] (kovarians).
  - OBS! Detta bryter mot typsäkerheten (the array loophole)! Intressant att reda ut varför de som skapade språket valde detta, frivillig uppgift...[5].
    Tvingas införa runtime kontroll för ArrayStoreException, d.v.s. varje gång man stoppar in något i en array kostar det.

    "...an assignment to an element of an array whose type is A[], where A is a reference type, is checked at run-time to ensure that the value assigned can be assigned to the actual element type of the array, where the actual element type may be any reference type that is assignable to A."//JLS 10.10
- Förutom att ha Object som supertyp har alla arrayer med primitiva element två supertyper till (i form av två interface) Cloneable och Serializble
  - Om p primitiv typ, then: Object :> p[], Cloneable :> p[], Serializable :> p[] (se Kanonisk form) Kod: inherit.array

[4] Tänkbart med kontravarianta parametrar d.v.s. parameter i subklass är basklass till parameter i superklass, d.v.s. kräver mindre, ofarligt.
[5] Generiska typer löser problemet, återkommer...

11 Abstrakta och inre klasser

11.1 Anonyma klasser

Används t.ex. till lyssnarklasser i Swing eller inre (singleton)klasser eller som parametrar. Att skapa tillfälliga klasser som "function object's" förekommer, dock skall man inte skapa för många (effektivitet).

Kod: anonymous

11.2 Abstrakta klasser

(JLS 8.1.1.1) Om en klass har någon abstrakt metod måste klassen deklareras som abstract t.ex.

```
public abstract class Point {
  int x = 1, y = 1;
  void move(int dx, int dy) {
    x += dx;
    y += dy;
    alert();
  }
  protected abstract void alert();
}
```

En klass har en abstrakt metod om;

- den explicit deklarerar en sådan.
- någon superklass är abstrakt och klassen varken implementerar eller ärver en implementation av metoden.
- klassen implementerar ett interface men inte implementerar alla metoder.
- Kan inte instansieras, jmf gränssnitt.
- En abstrakt basklass designad för arv kan kräva att subklasser override (annars går den inte att instansiera).

11.2.1 Abstrakta klasser kontra gränssnitt

- Vi programmerar alltid mot gränssnitt, men...

- ...om flera klasser implementerar ett gränssnitt och *dessutom* har duplicerad kod kan man skjuta upp den duplicerade koden till en abstrakt basklass.

Det ovan är vanligt i Javas standarbibliotek. Antag gränssnittet NNN, den abstrakta klassen heter då alltid AbstractNNN t.ex. AbstractAction (gränssnittet Action). Kod: abstractclass

11.2.1.1 Abstrakt klass istället för gränssnitt

Typer specificeras bäst m.h.a. ett interface. Ett stort problem med gränssnitt är att de måste bli "rätt". Om man misslyckas med gränssnittet kommer man att få leva med problement för evigt...

Antag att vi utvecklet ett bibliotek och publicerat detta. Efter ett tag anmärker användarna att något är fel eller konstigt. Vi kan då inte bara ändra gränssnittet eftersom det finns en massa kod som bygger på "kontraktet". Tar vi bort något kommer viss kod bryta ihop (break) lägger vi till saknas implementering i klasser som använder gränssnittet. Om vi ändra riskerar vi att alla applikationer som använder biblioteket måste gås igenom.

Observation Det är mycket svårt att ändra publicerade gränssnitt.

Om det är mycket viktigt att kunna bygga på en typ skulle man i dessa fall kunna specificera typen m.h.a. en abstrakt klass, lätt att lägga till (skapa en default impl. i klassen, alla andra ärver).

11.3 Designmönster: Template

Template är ett mönster som bygger på att man har mycket gemensam kod i en metod utom på ett specifikt ställe. Man gör då metoden abstract (och därmed klassen) och låter sedan varje konkret subklass implementera det som är specifikt i den ärvda metoden. Kod: template

11.4 Inre klasser

(JLS 8.1.3) Tyvärr ett väldigt rörigt område...

- Vanliga klasser kallas "top level", deklarerade på paketnivån.
- Inre klasser (inner classes), klasser deklarerade inuti en annan, ger ytterligare en nivå (nästlade klasser, nested classes).
  - Kan nästlas i flera nivåer, verkar mycket esoteriskt, undvik!
- Java tillhandahåller 4 olika sorter av inre klasser (för samtliga gäller olika specialfall och restriktioner t.ex. om de får vara abstract, deklarera statiska fält, statiska initeringblock, om de kan vara public, private o.s.v., o.s.v.).

Inner class, klass deklarerad i en omslutande yttre klass. Klassen är medlem i den yttre klassen. Klassen har en implicit referens till den omslutande klassen, kan alltså direkt referera fält m.m. i denna (även private fält). Kan ha private, public eller package (defaut), o.s.v. Måste instansieras i den omslutande klassen.

Kod: innerclasses.inner

Local inner class, en klass deklarerad i en metod. Som ovan men kan dessutom använda lokala variabler och metodparametrar *om dessa är final deklarerade*. Variablerna i instansen av den lokala klassen "försvinner" alltså inte efter att metodanropet avslutats! Om klassen skapas i en static-metoder blir den anonyma klassen static, se nedan.

Kod: innerclasses.swing

Anonymous inner class, som local inner class men klassen saknar namn (kommer att vara subtyp till en nämnd klass eller gränssnitt (kan göra new på ett gränssnitt här!!!).[1][2]

Kod: innerclasses.closure

Nested top-level classes = static inner classes, se nedan.

- Vid kompilering skapas "$"-klasser av de inre klasserna som helt fristående (filer med $-teckn i). För att dessa fristående (inre) klasser skall komma åt fält i den yttre klassen ändras synligheten till package!!!

11.4.1 Arv

Inre klasser kan ärva eller ärvas, skall undvikas.

11.4.2 this

Man får se upp när man använder this! Kan bero på var det står, i inre eller yttre klass (förutom ev. polymorfism). *Finns flera this i koden!*

Antag att vi har en yttre klass Outer och en inre Inner. För att komma åt den omslutande instansen, Outer-instansen (enclosing instance) från den inre klassen Outer.this.

Kod: innerclasses._this

11.4.3 Statiska (inre) klasser

Om en klass är deklarerad med static räknas den inte som en inre klass, den är en "top-level class", även om den rent syntaktiskt är skriven i en annan klass (den kallas dock nästlad) [3].

- static betyder alltså här något helt annat!!
- Static inner class, innehåller inte någon referens till omslutande klass (lite effektivare). Används då man inte behöver denna referens.

[1] Anonyma inre klasser är det närmaste man kan komma s.k. "closures". Closures kan i vissa fall förenkla programmeringen, ...have also been proposed as a new feature for Java SE 7..(behöver full tillgång till lokala variabler.

[2] Språket Scala verkar intressant, se http://www.scala-lang.org/

[3] Många författere säger "static inner class"...

11.5 Designmönster: Iterator

Designmönstret iterator gör att vi kan traversera (genomlöpa) en struktur utan att veta hur den är konstruerad (representationen). Ger typiskt ett element i taget i "någon ordning". Ofta är iteratorerna generiska, kommer senare. Se även Container-klasser.

Kod: iterator

11.6 Design

Användning av anonyma och inre klassser är specialfall, undvik...(blir lätt rörig kod). Några tillfällen då de kan användas;

- There is no reason for an object of the local class to exist in the absence of an object of the enclosing class (jmf TrieNode och Trie).
- There is no reason for an object of the local class to exist outside a method of the enclosing class.
- Methods of the object of the local class need access to members of the object of the enclosing class.
- Methods of the object of the local class need access to final local variables and method parameters belonging to the method in which the local class is defined

12 Containerklasser

Detta bygger på boken kapitel 17. Jag säger nog behållare istf container klass (kortare).

12.1 Återanvändning

(reuse) Ingår som en övrig strategi. Om vi kan återanvända tidigare utvecklad (förhoppningsvis) korrekt kod *utan att behöva förstå detaljerna*[1] så minskar vi komplexiteten.

Observation Återanvändning minskar komplexiteten. Vi får funktionalitet på högre nivå.

- Ytterligare fördel: Betydligt snabbare utvecklingstid.
- Bättre kvalité, buggar bör försvinna med tiden.

12.1.1 Vad är återanvändbart

Klasser eller design som har med domänen att göra är sällan återanvändbart. Däremot har man lagt märke till att vissa programkonstruktioner återkommer i applikation efter applikation. T.ex. är det väldigt vanligt med klasser som fungerar som behållare för andra klasser (container/collection classes). En tabell över procentuella antalet klasser i några större applikationer som *använder* behållare [2].

Applikation	Antal klass-er	Som använder behållare
Jedit 4.1	644	123 (19.1%)
Xalan J 2.5.2	2,395	398 (16.6%)
Jakarta Velocity 1.1b1	2,610	780 (29.9%)
Apache Tomcat 5.0.19	3,959	1,084 (27.4%)
Eclipse 3.0-M7	21,774	4,757 (21.8%)
JBoss 4.0.0DR2	32,820	7,888 (24.0%)

- Ca en femtedel av klasserna använder behållare, d.v.s. behållare är lämpliga kandidater får återanvändning.

[1] Kräver högklassig dokumentation. Dokumentation är oerhört viktig!
[2] Discovering and Debugging Algebraic Specifications for Java Classes, Johannes Henkel.

12.2 Typer av behållare

Behållarna kan klassificeras och namnges utifrån sina operationer och sitt beteende. Några exempel (typiska operationer efter);

- Mängd - add, remove, inga dubbletter (som matematisk mängd). Smidigt att slippa kontrollera dubbletter.
- Lista - add(index), remove(index), find(index), ... (en ordnad sekvens).
- Stack - Det finns en speciall position "stacktoppen". Operationerna utförs på denna. Pop (ta bort), push (lägga till), top (avläsa). Kallas LIFO, (last in first out). Används ofta för att "komma ihåg" var man var.
- Kö - enqueue (ta bort från början), dequeue (lägga till sist). Kallas FIFO, (first in first out). Används för att hantera saker i tur och ordning (spara tillfälligt om maninte hinner med).
- (Avbildnings)tabell - lookup(key), som telefonkatalog. OBS! Att det är lättare att gå från sökvärde (key) till värde (value). Om vi vet namn så hittar vi telefonnummer, om vi vet telefonnummer..öh? Inga dubbletter för key's är tillåtna. Mycket vanlig för att spara saker under ett namn. Används vid kvalificerade associationer.
- Finns dessutom många specialiserade som vi inte går in på här (en variant är trådsäkra).

Alla imperativa (OO) språk har normalt stöd (bibliotek) för de flesta typerna av behållare, så ock Java.

Behållare kallas i Java för Collections. Alla behållarna ovan m.fl. finns implementerade i The Java Collection Framework. Se speciellt Design FAQ, intressant...!

12.2.1 Konstruktion av behållare

Att konstruera behållare är en komplex uppgift. I kursen datastrukturer tas detta upp. Ni behöver inte kunna något om implementationen i denna kurs.

- Att kunna förstå API:er är dock viktigt.

12.2.2 Traversering av behållare

Med traversering menas genomlöpning av elementen i en behållare (t.ex. avläsa alla värden i tur och ordning).

12.2.3 Behållare och Arrayer

Behållare ligger på en högra abstraktionsnivå än arrayer. Föredra behållare framför arrayer (array ev i lågnivå klasser, moduler).

12.3 Generiska behållare

Behållarna måste kunna innehålla "vilka typer av objekt som helst". Man vill inte implementera en behållare för varje typ (Integer lista, String lista, o.s.v., vill kunna använda samma kod, reuse...).

- Lösning är generiska (generic) behållare. Man anger för varje behållare (klass) m.h.a. en typparameter vilken sorts objekt behållaren kan hantera.
- Hur man konstruerar enkla generiska behållare (klasser) undersöker vi senare, se generiska programenheter.

12.4 Samlingar-Gränssnittet Collection

Se bok.

OBS! Vi programmerar konsekvent mot gränssnitt. Så här skall det se ut (variabeln är alltid av gränssnittstypen);

```
// Typeparameter String
Set<String> s = new HashSet<String>();

// Typeparameter Integer
List<Integer> l = new ArrayList<Interger>();

// Possible also
Collection<Integer> c = ArrayList<Interger>();
```

Kod: container

12.5 Klassen Collections

Se bok.

En hjälpklasser som kan utföra divers saker med Collections.

> "This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends."// API Collections

Finns också motsvarande för arrays

> "This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists."// API Arrays

För kopiering av arrayer finns dessutom: System.arrayCopy(...)

12.6 Iteratorer

Se bok och inre klasser.

12.6.1 Iterator

Iteratorer i Java implementeras som inre klasser som implementerar gränssnittet Iterable.

```
// Interface Iterator<T>
public abstract boolean hasNext();
public abstract T next();
public abstract void remove();
```

12.6.2 Iterable

Iterable<T> är ett annat gränssnitt som specificerar att klassen skall ha en iterator (som man skall kunna komma åt).

```
// Interface Iterable<E>
public Iterator<T> iterator();
```

- Map implementerar inte Collection och därmed inte Iterable. Man kan plocka ut elementen (type Map.Entry) eller nycklarna som mängder. Värdena kan man får som en Collection.

12.6.3 ListIterator

Iterator kan bara stega framåt. Gränssnittet ListIterator kan stega åt båda håll. Se vidare java.util.ListIterator (Javadoc).

12.6.4 Kort for-loop

Allt som implementerar Iterable kan traverseras m.h.a. den förkortade for-loopen (bl.a. allt som inplementerar Collection och därmed List, Queue, Set, m.fl.).

- Vanliga arrayer implementerar inte Iterable men kan traverseras med den korta for-loopen ändå (JLS 14.14.2)
- En kopia av loop-variabel skapas (men har vi referenser så spelar det ingen roll!

Kod: short-for

12.6.5 ConcurrentModification Exception

Att "oväntat" förändra en Collection (t.ex. ta bort) samtidigt som man genomlöper leder till ConcurrentModificationException.

> "Thrown by iterators and list iterators if the backing collection is modified unexpectedly while the iteration is in progress."

- Undviks genom att använda iterator.remove().
- Kan t.ex. inte ta bort något i en kort for-loop.

Kod: container.traverse

12.7 Listor

Se bok.

12.8 Mängder

Se bok.

12.9 Avbildningstabeller

Se bok.

12.10 Varianter

Finns olika aspekter på behållare.

Kod:variations

12.10.1 Checked collection

> "...if in your project you use either third party code or legacy code that you suspect of not using generics consider wrapping your collections with Collection.checked* methods!"

Checked collections gör en runtime-kontroll på objekt man försöker lägga till. Om objektet inte uppfyller kraven slängs ett undantag direkt.

12.10.2 Unmodifiable collection

Gör att vi inte kan lägga till eller ta bort element (dock: elementen kan förändras).

12.10.3 Trådsäkerhet

Se Aktiva objekt.

12.10.4 Weak

Om man stoppar in saket i en Map så kan man ta bort value-objektet. Key-objektet kan dock ligga kvar och dessutom kommer det inte att skräpsamlas eftersom det finns en referens från tabellen. Har man otur (och kör programmet länge) kan man drabbas av OutOfMemoryException.

- Det ovan kallas ofta memory leak (minnesläcka).

Ett sätt är att använda en WeakHashMap. I denna sparas key-objektet som en s.k. weak reference[3]. Om det bara finns weak references till ett objekt kommer det att skräpsamlas (trots att det finns referenser).

12.11 Byte mellan behållare

Ganska ofta vill man byta från t.ex Set till List eller array. Kan vara lite svårt att lista ut ibland.

Kod: container.change

12.12 Implementation av egna ADT:er

Normalt skall detta inte behövas men om så...kan man utnyttja Javas Collection framework. Sker genom att subklassa någon av;

- AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList, AbstractQueue eller AbstractMap (se vidare abstrakta klasser)
- Man skall aldrig subklassa List, Map, Set,...

Man behöver då bara implementera visa metoder resten sköts av basklassen.

[3]Finns 4 olika sorters referenser: strong, soft, weak, and phantom

13 MVC

Model-view-control är en speciell design för applikationer med grafiskt användargränsnitt.

- Det gäller att inte röra ihop GUI-kod och övrig kod.

13.1 Callbacks

> "In computer programming, a callback is a reference to executable code, or a piece of executable code, that is *passed as an argument* to other code."//Wikipedia

Typiskt t.ex. i Swing där vi ger en JButton en referens till en klass som implementerar ActionListener. Då man senare klickar knappen körs koden som implementerar actionPerformed.

Kod: callback

13.2 Designmönster: Observer

Observer är ett mönster som använder callbacks. En klass, med en callback-metod, registrerar sig som observatör av en annan klass. Då den senare ändrar tillstånd notifieras alla observatörer genom att callback-metoden anropas.

- Detta kallas en push-design, ett objekt trycker ut förändringar till ett antal andra objekt (istf att dessa skall anropa, en pull-design).
- Alla observatörer måste implementera ett gränssnitt som garanterar att de har en callback-metod.

Kod: observer

13.3 Model-view-controller

Som sagt detta gäller bara GUI-program. Programmet delas i tre delar (skikt);

- Model, designmodellen (lägsta skiktet). Modellen signalerar tillståndsförändringar m.h.a. observer-mönstret. Inga refernser till GUI:et eller kontroll i modellen. Samma som domänlagret i en skiktad design.
- GUI de "synliga" delarna, Swing i vårt fall (översta skiktet). Gör anrop på kontroll och modell. Uppdateras före/efter anropen eller m.h.a. Observer. Valda delar av GUI:et fungerar alltså som observatörer av modellen. Samma som presentationsskiktet i en skiktad design.
  - Ingen domänlogik får förekomma i GUI:et

- Control (mellanskiktet). Grundtanken är att kontrollskiktet skall föra information mellan GUI och modell. En viktig punkt är att control-skiktet inte får ha direkta referenser till GUI:et, isf kan vi inte testa kontrollklasserna. Samma som applikationslagret i en skiktad design.
  - Om man har enkla (snabba) kontroller (one-liners) kan man ersätta dessa med direkta anrop i GUI:ets lyssnare (i actionListener:n).
  - Har man komplicerade saker skapar man ett kontrollobjekt i lyssnaren och låter detta ta över (ev i egen tråd).

Kod: mvc

14 Kanonisk form

(JLS 4.3.2) (Canonical form[1]) Klassen Object innehåller ett antal mycket grundläggande metoder. Vid implementation av klasser får man se upp med; Kod: object

- Override av metoderna: toString(), hashCode(), equals() och clone() [2][3].

Motivering;

> "Why is it important to implement these methods correctly? In a small application written, used, and maintained by one individual, it may not be important. However, in large applications, in applications maintained by many people and in libraries intended for use by other people, failing to implement these methods correctly can result in classes that cannot be subclassed easily and that do not work as expected.
>
> It is, for example, possible to write the clone method so that no child classes can be cloned. This will be a problem for users who want to extend the class with the improperly written clone method. For in-house development this mistake can result in excess debug time and rework when the problem is finally discovered. If the class is provided as part of a class library you sell to other programmers, you may find yourself rereleasing your library, handling excess technical support calls, and possibly losing sales as customers discover that your classes can't be extended. "

Förutom detta så är det ur designperspektiv intressant att se hur olika val påverkar

14.1 toString()

Rekommenderat att alltid override denna. Anropas automatiskt i många fall bl.a. +-operator (om andra operanden är en sträng). Mycket bra vid avlusning. Se till att användbar information skrivs ut.

[1] Basic, canonic, canonical: reduced to the simplest and most significant form possible without loss of generality, e.g., "a basic story line"; "a canonical syllable pattern."//Wikipedia
[2] Eclipse kan generera alla utom clone.
[3] Finns också en metod finalize i Object. Kommer att köras precis innan objektet skräpsamlas. Inte garanterad att köras. Används inte av oss (strider mot best practices).

14.2 hashCode

Objects hashCode genererar ett stort någorlunda unikt heltal (m.h.a. en formel). HashCode används t.ex. då man sparar objektet i en Map. Platsen ges av resultatet av hashCode.

Om en klass overrides equals-metoden skall den alltid overrid:a hashCode(). Motiveringen är att objekt som är lika (equals) skall hamna på samma plats i behållaren. Följande kan t.ex. uppstå om Id implementerar equals() men saknar hashCode(), m är ett behållarobjekt;

```
Person p = new Person();
// Spara person under Id...
m.put( new Id( 6905165058 ), person );
// Hämta samma person...men
p = m.get(new Id( 6905165058 ));
// ...p blir null (ej funnen)!
```

De två instanserna av Id kommer att ha olika hash-kod, så när vi söker efter id (som är equals), så blir det ändå fel. Sökning sker på fel plats!

,

14.3 Equals

Objects equals ger likhet utifrån på identitet (==). Om vi vill ha likhet baserat på värde måste vi override:a denna.

- Värdeobjekt override:ar vanligen equals, för dessa gäller ju samma "innehåll" (tillstånd) för likhet. Entitetsobjekt har inget behov av metoden (fast i praktiken se vidare nedan).
- Equals är mycket viktigt i samband med behållare, nämns ofta i dokumentationen.

14.3.1 Equals och arv...

Kontraktet för equals;

- Skall vara en eqvivalensrelation d.v.s.
  - reflexiv, $\forall$o; o.equals(o) == true.
  - symmetrisk, x.equals(y) == true => y.equals(x) == true.
  - transitiv, x.equals(y) $\wedge$ y.equals(z) => x.equals(z).
  - konsistent, d.v.s. spelar ingern roll hur många gånger vi jämför svaret skall alltid vara detsamma (inga sidoeffekter).
- $\forall$o; o.equals(null) == false.

Hur skall equals fungera vid arv? Vi måste bestämma om vi skall tillåta jämförelser mellan super och subklass d.v.s.

1. Skall super.equals(sub) alltid vara falskt (same type only comparison)...?
   - Spelar ingern roll om Person eller Student är super/sub-typ till varann, Person("Sven") och Studen("Sven") är aldrig lika.
   - Kan aldrig jämföra objekt i en hieraki, m.m.
   - Enkelt att implementera, använder; this.getClass() != other.getClass().
2. ... eller skall super.equals(sub) kunna ge true (mixed type comparison)?...
   - D.v.s. om Person :> Student så är Person("Sven") och Student("Sven") lika.
   - Innebär att vi kan jämföra olika objekt i en hierarki, kan vara praktiskt.
   - Använder !(obj instanceof MyClass)

14.3.2 ...ger problem[4].

- Om vi inte har en final klass.
- "Same type" equals (getClass) ger problem med symmetri och transitivitet. Problem med key-värden i tabeller. Spelar ingen roll om man tar hänsyn eller inte till signifikanta fält i subklasser (fält som skall ingå i jämförelsen).

Kod:
equals.sametype
equals.sametype

- "Mixed types" equals (instanceof). Ger problem med symmetri och transitivitet om vi har signifikanta fält i subklasser, annars ok. Symmetri med signifikanta fält kan fixas men då bryter transitiviteten ihop.

Kod:
equals.mixedtyp
equals.mixedtyp

14.3.3 Lösning

1. Använd delegering. Låt ett annat objekt wrappa "subklassen". Ger samma lösning som same type only (tyvärr en del extra kodning).

Kod:
equals.delegatio

2. Deklarera unikt id attribut och equals-metod i basklassen låt subklasser ärva allt (gör metod final).

KOD
equals.mixedtyp

14.4 Djup och grund kopia

Har vi berört förut (men kanske inte namnen).

- Grund kopiering (shallow copy) innebär att vi bara kopierar variabler från en klass till en annan rakt av. Fungerar för primitiva typer (kopier skapas) och ick-muterbara värden (dessa kommer att vara delade med det gör inget, de kan inte ändras).

[4]Se t.ex. webben, Angelica Langer, Secrets of equals() - Part 1, Secrets of equals() - Part 2

- Djup kopiering (deep copy), Om man har muterbara referenser räcker det inte att göra kopior enligt ovan man måste kopiera de refererade objekten också (och så vidare...), annars kommer orginal och kopia att ha delade objekt.

  "Note the following disclaimer: What exactly constitutes a deep clone is debatable. Even though two objects can safely share the same String reference viewed as data, they cannot if the same field is used as an instance-scoped object monitor (such as when calling Object.wait()/notify() on it) [se trådar] or if field instance identity (such as when using the == operator) is significant to the design. In the end, whether or not a field is shareable depends on the class design."//Java World

14.5 Clone

Ofta behöver man en kopia av ett objekt. Clone-metoden var tänkt att användas till detta. Tyvärr är Javas implementation av clone "broken" och bör undvikas. Trots detta kan det vara bra att känna till en del om metoden (används i standarbiblioteken).

Object's clone-metoded (är protected) gör en grund kopia av objektet. För att göra en subklass "klonbar" måste klassen override:a clone() (och göra den public) och implementera gränssnittet Cloneable.

```
// This is the basic, standard implementation
class A implements Cloneable {

    public Object clone() {
        try {

            // First of all: Must clone all superobjects
            // (part of this). Get at shallow copy.
            // Possible to cast to A (runtimetype A)!
            // If this class would have been B then
            // possible to cast to B (some hack in background)!!!
            A a = (A) super.clone();
            // If references, must handle here (deep copy)
            ...
        } catch (CloneNotSupportedException e) {
            ...
        }
    }

}
```

Gränssnittet Cloneable är ett markerinterface (saknar metoder). Det som istället händer är att Object's clone-method ändrar beteende (justerare runtime typen). Ett mycket underligt sätt att använda ett gränssnitt och ett "mystiskt" sätt att skapa nya objekt (extra-linguistic, utomspråkligt, ... ett hack).

Räcker det med en grund kopia kan man direkt i clone anropa Objects (eller någon superklass) clone i den override:ena metoden (super.clone()). Vid djup kopiering får man se till att clone gör just detta (skapa kopior på referensvariabler).

I praktiken fungerar clone som en slags kontruktor, måste uppfylla allt som gäller för kontruktor (inte anropa overriden metod, kan förstöra invarianter).

Om man anropar clone på ett objekt som inte implementerar interfacet genereras ett undantag; CloneNotSupportedException (eller InternalError om man gör som vi, se kodexempel). Om en klass implementera Cloneable så är det inga problem med subklasser, de ärver.

- Kontraktet för clone (kan vara situationsberoende)
  - o.clone() != o
  - o.clone().getClass()== o.getClass()
  - o.clone().equals(o) == true (inget absolut krav)
  - Ingen konstruktor får vara inblandad (gäller alltså för aktuellt objekt inte delar av)! Om konstruktor används så uppstår en cast-exception i subklassen (final klasser ok).

Kod: clone.basic, clone.constructo

14.5.1 Undantag

Signaturen för clone i Object har throws CloneNotSupportedException (checked exception).

- Antag att någon superklass (som är direkt subklas till Object) implementerar Cloneable och clone och alltså kan klonas. Den behöver då aldrig kasta CloneNotSupportedException
- Subklassen ärver Clonable. Om den också implementerar clone så behövs inte heller här något undantag.

Att ange att clone kastar CloneNotSupportedException är helt ologiskt!

- Enda chansen att det går fel skulle vara att Objects clone inte fungerer!
- Typiskt deklarerar men inte något undantag för clone (se kodexempel).

Kod: clone.basic

14.5.2 Kovarians

Ev kan man använda kovarians i subtyper. Måste vara konsekvent i alla klasser (eller inte implementera metoden alls).

Kod: clone.covariant

14.5.3 Förhindra Kloning

Man kan uttryckligen förbjuda cloning, override clone och låt metoden kasta CloneNotSupportedException då den anropas (det enda den gör).

- Icke-muterande typer och entity typer behöver ingen clone.

14.5.4 Kritik

Kritiken är massiv:

- Ologisk undantagshantering
- Cloneable: Saknar metod clone.

  > "making something Cloneable [typomvandla] doesn't say anything about what you can do with it....which means that you can't do a polymorphic clone operation. If I have an array of Cloneable, you would think that I could run down that array and clone every element to make a deep copy of the array, but I can't. You cannot cast something to Cloneable and call the clone method, because Cloneable doesn't have a public clone method and neither does Object. If you try to cast to Cloneable and call the clone method, the compiler will say you are trying to call the protected clone method on object."

- Det förväntas att en klass som implementerar interfacet har en fungerande clone-metod. För stt det skall fungera krävs att;
  - alla superklasser har konsistent implementation av clone()
  - att alla referenser går att klona (dessa har ev. superklasser).

  Finns inget sätt att tvinga fram detta.
- Final: Clone är inte kompatible med final (ev men reflection).
- Generiska klasser och samlinga skapar speciella problem (måste använda reflection).
- Alternativ finns nedan.

14.6 Kopieringskontruktorer

Ett annat (bättre) sätt att kopiera objekt är att skapa en s.k. kopieringskonstruktor!

- Fungar bra för final-klasser.
- Kan ge problem om variabeln är en subklass.

```
// A :> B
// Runtimeclass is B
A a1 = new B();
//Runtimeclass is A :-(
a2 = new A( a1 );
```

Kod: copyctor

- En variant med interface med metod som i sin tur anropar kopieringskonstruktorn. Ger rätt runtimetyp.

Kod: copyctor.uno

14.7 Kopiering med Serializable

> "A common solution to the deep copy problem is to use Java Object Serialization (JOS). The idea is simple: Write the object to an array using JOS's ObjectOutputStream and then use ObjectInputStream to reconsistute a copy of the object. The result will be a completely distinct object, with completely distinct referenced objects. JOS takes care of all of the details: superclass fields, following object graphs, and handling repeated references to the same object within the graph."

Eventuellt ineffektivt (stora objekt).

14.8 Jämförelse

Förutom likhet mellan objekt behövs ofta jämförelse. Är objekt a större/mindre än b? Användbart t.ex. vid sortering.

14.8.1 Comparable

> "This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.[5]"// API

Genom att låta en klass implementera (det generiska) gränssnittet Comparable visar man att klassen har en *naturlig ordning*. Enda metoden är;

```
// Comparable<T>
public int compareTo( T o );
```

- Sortering sker lätt m.h.a. färdiga metoder i Collections och Arrays om klassen implementerar Comparable (många Java-standardklasser implementerar Comparable t.ex. String). Dessutom kan man söka min och max värden i Collections. Exempel;

[5] The relation that defines the natural ordering on a given class C is:{(x, y) such that x.compareTo(y) <= 0}.

```
// Customer-object in list
// implements Comparable
Collections.sort (customerList);
```

- Kontraktet för Comparable (sgn-funktionen ger -1, 0, 1)
  - $\forall$x,y, sgn(x.compareTo(y))== -sgn(y.compareTo(x)).
  - x.compareTo(y) $>$ 0 $\wedge$ y.compareTo(z) $>$ 0 $\Rightarrow$ x.compareTo(z) $>$ 0.
  - x.compareTo(y) == 0 $\Rightarrow$ sgn(x.compareTo(z)) == sgn(y.compareTo(z)).
  - Rekommenderas startkt att; (x.compareTo(y) == 0) == (x.equals(y)). Kallas att det är *consistent* med equals. Om ej skall detta anges mycket tydligt i specificationen.

    > "Curiously, BigDecimal violates this. Look at the Java API documentation for an explanation of the difference"

    Har att göra med den naturliga ordning. Generellt är likhet och jämförelse skilda saker (kan jämföra på area men likhet ges av omkrets).
- Collections och Arrays har metoder för sökning (binärsökning), kräver att man först sorterar.
- Nackdelen med Comparable är att man i förväg måste bestämma objektens naturliga ordning (vilken kanske inte är helt uppenbar). Se comparator.

Kod: comparable

14.8.2 Comparator

En mer flexibel lösning är gränssnittet Comparator som inte implementeras av klassen, utan av en speciell comparator-klass som sedan används vid sorteringen. Gränssnittet för Comparator .

```
// Comparator<T>
int compare(T o1, T o2);
boolean equals( Object o );
```

Exempel;

```
// Using a comparator class
Comparator<Customer> comparator = ...
Collections.sort (customerList,
           comparator);
```

- Collections och Arrays sortering kan också göras med hjälp av Comparator.
- Vissa implementationer av behållare, t.ex. TreeSet, kan ta en Comparator som argument till konstruktorn, d.v.s. behållaren ges en total ordning då den skapas.

Kod: comparator

14.8.3 Design

Överväg noga om klassen skall inplementera Comparable! Hur flexibelt ändra sortering (vad skall det sorteras efter).

15 Aktiva objekt

Se även boken kapitel 12.

15.1 Processer

En process är ett program under körning (programmet självt är bara en samling bytes). En process behöver olika resurser t.ex.;

- Minne (heap, anropsstack, m.m.)
- Processorkraft.
- IO resurser, in och ut strömmar.

En process har också vissa säkerhetsattribut och ett tillstånd (kan t.ex. flyttas ut från arbetsminnet till virtuellt minne (disk)).

På ett system som kan köra flera processer samtidigt (multitasking) måste processerna fördelas antingen på flera processorer eller så får processerna dela på processorns tid (time sharing).

Hur tiden fördelas mellan processer sköts av en schemaläggare (scheduler). Finns många olika strategier (processer kan ha olika prioritet). Schemaläggare är helt plattformsberoende. För att kunna flytta program får de inte vara beroende av schemaläggaren funktion (timing).

Olika processer är normalt helt separerade. Om två processer vill kommunicera sker detta med speciella tekniker, kallas "inter process communication". I Unix/Linux finns t. ex. pipes som möjliggör detta.

Att starta en process kräver en del arbete, en process är "tung" att starta.

15.1.1 Blockerande anrop

En process följer en sekvens av instruktioner. Om sekvensen stöter på ett anrop till en metod innebär detta normalt att resten av programmet (processen) får vänta tills metoden exekverat klart, kallas blockerande anrop.

- Om metoden tar lång tid måste "allt annat" vänta.

Kan vara irriterande t.ex. då man använder GUI:n. Om metoden tar lång tid (ladda ner fil) kommer hela gränssnittet att låsa sig tills filen är nerladdad. Processen kan fortsätta med instruktionerna för att uppdatera GUI:et först efter att nedladdningen är klar.[1]

Kod: blocking

[1] Finns även non-blocking IO i paketet nio (new IO). Används t.ex. om man skall skriva högkapacitetsservrar.

15.2 Trådar

En lösning på bl.a. blockerande anrop är trådar. Trådar gör att en process kan göra flera saker samtidigt (eller time sharing). Man kan t.ex. låta det som tar tid köras i en egen tråd. Trådade program kan upplevas som mer responsiva[2].

Kod: blocking

- En tråd är en fristående "thread of execution" *inom en process*[3].
- En tråd är en lättviktsprocess det går snabbt att byta mellan trådar (jämfört med processer).
- Processen har flera (samtidigt) exekverande förlopp (parallell, concurrent-progam).

Trådar påminner om processer ovan men;

- Olika trådar kommer åt den gemensamma processens minne (variabler), dessa delas alltså mellan trådarna (shared resource). Ingen interprocesskommunikation behövs.
- Varje tråd har en egen anropsstack, lokala variabler delas inte!
- Programmets instruktioner (objektens metoder) kan köras av olika trådar, i ena stunden exekveras en metod av en tråd, i nästa körs de av en annan tråd (multithreading). Hur detta växlar (context switch) kan vi som programmerare inte styra! Schemaläggaren (inbyggd i JVM:en) byter trådar utifrån någon strategi (vi uppfattar det som slumpmässigt, vi vet aldrig vilken tråd som kommer att köras efter nästa byte).

Observation Trådade program ger en kraftigt ökad komplexitet. Flera parallella förlopp som kan påverka tillståndet.

D.v.s. undvik om möjligt trådade program.

15.3 Trådar i Java

(JLS 17)

- Alla Javaprogram har minst en tråd, den som börjar exekvera main-metoden(), kallas "main thread"[4].
- Använder man Swing skapas automatiskt ett antal trådar t.ex. händelsetråden.
- I Lab 3 kommer vi att se att inkommande nätverksanrop skapar trådar.
- Man kan skapa egna trådar m.h.a. färdiga Java klasser, se nedan.

[2] Trådade program exekverar inte snabbare, det tar tid att byta mellan trådar.
[3] Finns typer, native/green-threads, daemon-threads,..
[4] Finns trådar i bakgrunden t.ex. minnehanteringen men dessa räknar vi inte med.

- Ett Java-program avslutas då samtliga trådar exekverat klart (main kan vara klar innan andra trådar! Ibland dyker utskrifter upp fast man tror man avslutat programmet...:-))[5].

15.3.1 Gränssnittet Runnable

Alla klasser som är tänkta att exekveras i en egen tråd måste implementera interfacet Runnable med enda metod;

```
public void run();
```

All kod tråden skall exekvera måste finnas i implementering av run (vanligen en loop). Vissa standarklasser implementerar interfacet t.ex ...

15.3.2 Klassen Thread

Klassen Thread representerar en tråd (dock är en tråd inte ett objekt! ..det är en thread of execution, blanda inte ihop...).

- Klassen implementerar gränssnittet Runnable.
- Ett antal metoder t.ex. start() som startar tråden.
- Metoden stop() skall aldrig användas. För att stoppa tråden (döda den) används boolesk variabel som bryter loopen i run.metoden. Därmed avslutas run och tråden dör, se vidare nedan.
- Thread ärver några grundläggande metoder från Object t.ex. wait, och notify (tas inte upp i denna kurs).

15.3.2.1 Statiska metoder i Thread

Finns ett antal användbara;

- Thread.currentThread() ger (namnet) på den tråd som exekverar koden.
- Thread.sleep(), söver tråden en viss tid (d.v.s. tråden exekveras inte under denna tid).

Obs! Att dessa inte påverkar ett objekt "lås", se vidare nedan.

[5]Förenklat, finns även s.k. daemon threads m.m.

15.3.3 Egna trådar

Kan skapas på flera sätt (vi föredrar dock att använda trådar på en högre abstraktionsnivå, se nedan).

1. Skapa en klass som implementerar Runnable. Skapa en instans av Thread och skicka klassen som argument till Threads kontruktor. Medför att koden i vår run-metod kommer att exekveras i tråden.
2. Låt klassen ärva Thread (mindre bra, undvik arv)

Kod: thread.basic

Att använda Eclipse debugger för att se hur trådar fungerar är mycket givande.

15.4 Race Conditions

Eftersom trådar delar tillstånd får de inte skriva/läsa hur som helst! Betrakta följande;

```
// Not thread safe
int x = 10;
x = x + 1;
```

Tilldelningen skall utföras två gånger d.v.s. efter detta skall x vara 12. Inget problem med ett otrådat program (single threaded). Antag nu att samma skall ske m.h.a. trådar.

1. Tråd A läser x värde från minnet och blir avbruten...
2. Tråd B läser x och blir avbruten...
3. Tråd A ökar x med 1 och skriver sedan till x (x = 11 alltså);
4. Tråd B gör som A d.v.s. x = 11, ...!

Det ovan kallas "race-conditions", d.v.s. resultatet är beroende på hur schemaläggaren hanterar trådarna (vilket vi inte kan påverka)[6].

KOD threads.raceconc

15.5 Atomära operationer

Vissa operationer är garanterade att inte bli avbrutna (är trådsäkra). Dock får man se upp med cachening, se nedan;

- en enstaka läsning eller skrivning av en variabel, frånsett typerna long och double, är en atomär operation[7].

Följande är inte trådsäkert (inte atomärt);

```
// Not thread safe (3 operations)
x++;
```

[6] Ett vanlig fel kallas "check and act" d.v.s. man kontrollerar ett villkor och skall därefter utföra något. Precis när man kontrollerat villkoret bli tråden avbruten och en annan tråd ändrar villkoret.

[7] Finns klass AtomicLong.

15.6 Synkronisering

Operationer (metoder) som *inte* leder till race conditions kallas trådsäkra (thread safe).

- Synkronisering innebär att man "bakar ihop" ett kodstycke till en atomär operation. Nödvändigt då flera trådar skall anropa koden och denna använder delad resurser. Åstadkomms i Java med det reserverade ordet synchronized, finns två varianter;

```
class A {
  // Synchronized method
  synchronized void doIt(){
     // Critical section
     // Shard resource used
  }
}

// Synchronized statement
synchronized(exp){
  // Critical section
  // Shard resource used
}
```

  Exp måste vara av referenstyp (vanligen ett enkelt objekt).

- Området mellan { och } kallas en kritisk sektion[8]. För metoden är alltså hela metoden en kritisk region.
- Om en tråd påbörjer en kritisk sektion är den garanterad att kunna exekvera klart sektionen. Byte mellan tråd sker före eller efter.
- Konstruktorer och initeringblock kan inte vara synchronized.
- Synchronized ingår inte i metodsignaturen (det är en implementeringsdetalj).
- Fler saker tillkommer t.ex. vid inre klasser...
- OBS! Sammansättning av synchronized-metoder inte blir atomär.

15.7 Objektlås

- Alla Java-objekt är associerade med en monitor[9]. En tråd kan låsa eller låsa upp monitorn. Vi kan förenklat säga att alla objekt i Java har ett implicit "lås" (lock = intrinsic locks = built in locks[10]).

[8] In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
[9] Monitor en språkmekanism (kompilatorm, JVM) som automatisk hanterar kritiska regioner m.h.a. lås (mutex).
[10] Finns andra varianter med bibliotek.

- En tråd måste "exklusivt äga" objektets lås innan den kan exekvera den kritisk regionen (d.v.s. om tråden lyckats så låser den objektet (och därmed regionen för andra trådar). Vem som äger ett lås bestämms av systemet [11].
  - För synkroniserade metoder är det låset för this tråden äger. Alla *synkroniserade* metoder i this kommer att vara låsta för andra trådar. Icke-synkroniserade kan köras.
  - En synkroniserad metod kan anropa en annan synkroniserad metod på objektet (det är ju samma tråd, samma lås).
  - För synkroniserade satser är det låset för objeket i "parentesen" man äger.
- Låset släpps då tråden lämnar den kritiska regionen (bara den tråd som har låset kan släppa det). Låset kan även släppas på andra sätt (som vi inte behöver i denna kurs).
- Om en region är låst av en tråd och en annan tråd försöker komma åt låset tvingas den att vänta tills låset släpps (läggs i ett s.k. wait set, eventuellt tillsammans med flera andra trådar,
  - OBS! Inte säkert att just den tråden får låset först när det släpps, någon annan tråd som väntar kan få det... Finns ingen turordning.
- En synkroniserad statisk metod innebär att man låser alla statiska klassmetoder.
- Synkronisering tar viss tid!
  - Många klasser i Java finns i två versioner, en trådsäker lite långsammare och en snabbare icke-trådsäker.

KOD
threads._synchr

15.7.1 Synlighet

Ingen annan tråd kan se tillståndsförändringarna då en tråd exekverar en kritisk region (tråden har en cache).

- Först då tråden lämnar regionen blir resultatet synliga för andra[12]!
- Innebär att man ibland måste använda synchronized-metoder trots att man bara har en atomär skriv/läs operation.

[11] JLS 17.1 "The synchronized statement computes a reference to an object; it then attempts to perform a lock action on that object's monitor"

[12] "When an object acquires a lock, it first invalidates its cache, so that it is guaranteed to load variables directly from main memory. Similarly, before an object releases a lock, it flushes its cache, forcing any changes made to appear in main memory."

15.8 Deadlock

> "Deadlock refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain"

För mycket (eller felaktig användning av) synchronized ökar risken för deadlock.

- Synkroniserade metoder skall aldrig anropa metoder som kan override:as (kanske skapas en annan tråd som i sin tur anropar aktuellt objekt, -> deadlock).

Kod: threads.deadlock

15.9 Samverkande trådar

Kod: threads.deadlock

Finns exempel i boken men tas inte upp i denna kurs (typiskt producent/konsument).

- Vi går alltså inte igenom wait och notify, notifyAll.

15.10 Schemalagda händelser

Ibland har man behov av att vissa saker utförs med jämna mellanrum (i bakgrunden). Man kan m.h.a klasserna Timer och TimerTask schemalägga körningar.

15.10.1 util.Timer och TimerTask

(java.util.Timer, inte swing.Timer)

- Timer, A facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals (Timer är trådsäker).
- TimerTask, A task that can be scheduled for one-time or repeated execution by a Timer (implements Runnable).

Schemalagda saker bör gå relativt snabbt, annars tar den övriga schemalagdas task's tid.

Kod: timer-task

15.11 Trådsäkra behållare

Metoder som lägger till/tar bort saker ut behållare behöver inte vara trådsäkra om behållaren är det. Följande gäller;

- Om man vill skapa en trådsäker variant av den behållara man har kan man använda;

```
Collection mySynchCollection =
  Collections.
   synchronizedCollection(myCollection);
```

- Finns ett helt paket med trådade versioner av behållare plus massor av andra saker se java.util.concurrent.

15.12 Swing

Swing är inte trådsäkert!

> "The Swing API was designed to be powerful, flexible, and easy to use. In particular, we wanted to make it easy for programmers to build new Swing components, whether from scratch or by extending components that we provide.
>
> For this reason, we do not require Swing components to support access from multiple threads.[13]"// Muller Walrath (from Sun)

15.12.1 The Single thread rule

> "Once a Swing component has been realized (painted or to be painted on screen), all code that might affect or depend on the state of that component should be executed in the event-dispatching thread"

Alltså allt som påverkar GUI:et skall exekveras i "event-dispatching tråden" (EDT, som startas automatisk då man har en Swing application).

Vi går ett steg längre och konstruerar dessutom hela GUI:et i EDT (se laborationer).

15.12.2 SwingUtilities

För att hantera interaktionen mellan EDT och andra trådar kan man använda klassen SwingUtilities.

Om vi har en tråd som skall påverka GUI:et kan vi låta denna tråd "lämna över" till EDT genom att använda metoderna.

- SwingUtilities.invokeLater() (icke-blockerade anrop)
- SwingUtilities.invokeAndWait() (blockerade anrop)

Kod: swingutilities

15.12.3 SwingWorker

Problemet ovan med att GUI:et låster sig vid tidsödande operationer kan lösas m.h.a. klassen SwingWorker (som kommer att starta en egen tråd i bakgrunden). Klassen är generisk och lite "mystisk"

```
public abstract class SwingWorker<T,V>
  extends Object implements RunnableFuture<T>
```

[13]Single-threaded GUI frameworks are not unique to Java; Qt, NextStep, MacOS Cocoa, X Windows, and many others are also single-threaded. This is not for lack of trying; there have been many attempts to write multithreaded GUI frameworks, but because of persistent problems with race conditions and deadlock, they all eventually arrived at the single-threaded event queue model in which a dedicated thread fetches events off a queue and dispatches them to applicationdefined event handlers. //Java Concurrency in Practice

Typparametrarna står för:

- T returtypen för metoden doInBackground(), som körs i bakgrundstråden.
- V står för typen på mellanliggande resultat (innan doInBackground är klar, kan t.ex. ha en progress indicator)

Metoden done() kommer att exekveras i EDT. I denna kan man alltså uppdatera GUI:et. De värden man kan behöva (resultatet av doInBackground) får man genom att anropa metoden get()

Kod: swing-worker

15.12.4 swing.Timer

Det finns en timer i Swing, används för animationer (sänder en ActionEvent enlig schemaläggningen). Eventen kan fångas i en vanlig Swing lyssnare. OBS! Att denna kod kommer att köras i Swing-tråden.

15.13 Design

- Undvik trådar om möjligt
- Trådsäkerhet åstadkomms genom;
  - Icke muterbara klasser.
  - Tillståndslösa klasser.
  - Användning av synkronisering;
  - Trådsäkerhet garanteras inte bara för att man använder synchronized, måste ske med insikt (variabler kan bero på varann, m.m.)
  - Synkronisering kostar i effektivitet.
- Deadlock
  - Man får se upp med synchronized, begränsa användning så mycket som möjligt. Eftersom vi låser objekt kan överdriven användning leda till deadlock.
  - Anrop "ut" från en synkroniserad metod måste ske med stor försiktighet, eventuellt kan det leda till en callback, som i sin tur leder till deadlock (objektet är ju redan låst av oss).
- Kan delegera trådsäkerhet, om man t.ex. använder trådsäkra behållare kanske inte metoden behöver vara trådsäker.
- Ett problem är konstruktorn, ett icke-muterbart objekt är muterbart under tiden kontruktorn exekverar (med något undantag). Kan ställa till problem vid trådade programmen.[14]

[14]Se t.ex. Immutable objects in Java , C. Haack et alt.

16 Generisk programenheter

En del finns i boken, se kapitel 17. En mycket bra referens till generiska typer är http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ. html

16.1 Generiska klasser

Då klassen deklareras anger man typvariabler (typparametrar) inom vinkelparenteser (traditionellt används en enda stor bokstav, T (type), E (element) m.fl). Enum kan inte vara generiska.

Kod: generic.basic

```
public class A<T> {

    private T t;
        ...
    }
```

16.1.1 Arv

Generiska klasser kan ingå i arvshierakier (behållare skall aldrig ärvas!). Inte blanda råa och generiska typer m.m., se nedan.

Kod: generics.inherit

16.1.2 Inre klasser

Går bra. Dock ej inre anonyma.

Kod: generics.inner

16.1.3 Kanonisk form

Hur implementeras equal, clone, ... för generiska klasser, left as excercise to the reader, clone måste använda reflection, se Angelica Langer...(kommer ej på tenta)

16.2 Instansiering av generiska klasser

> "A generic class declaration defines a set of parameterized types, one for each possible invocation of the type parameter section."

```
// Instantiating the
// generic type A<T>,
// results in a
// parameterized type
A<String> as = new A<String>();
```

16.3 Generiska gränssnitt

Generiska gränssnitt går också bra (som vi sett). Exempel;

```
public interface Comparator<T> {

      public int compare(T o1, T o2);
      public boolean equals(Object obj);

}
```

16.4 Användning av generiska gränssnitt

Finns två varianter beroende på om klassen skall vara generisk eller ej.

KOD generic.iface

- Icke generisk klass; implementerar det parametriserade gränssnittet

```
public class A implements IA<String> {
   private String s;
}
```

- Generisk klass (måste ange <A> för både klass och gränssnitt)

```
public class <A> implements IA<A> {
   private A s;
}
```

16.5 Begränsade typparametrar

Genom att vid deklarationen ange begränsingar för typparametern får kompilatorn mer typinformation. Exempel; T är en subtyp till Number!

```
// Always extend (also for interfaces)
public class A <T extends Number> {
  T data;
  :
  // Compiler knows this is ok
  data.doubleValue();
```

Förhindrar felaktiga instansieringar;

```
//String doesn't extend number!
A<String> a = new A<String>();
```

Generall iakttagelse: Man försöker "stämma av" typsystemet så att man kan göra så mycket som möjligt men ändå bibehålla typsäkerheten.

Kod: boundtype

16.5.1 Multipla typparametrar och multipelt begränsade typparametrar

Kan man ha. Exempel;

```
class Pair<A extends Comparable<A> & Cloneable,
           B extends Comparable<B> & Cloneable>
              implements Comparable<Pair<A,B>>,
    Cloneable {...}
```

- Samtliga begränsningar måste gälla (unionen av...) .

16.6 Generiska typer

Klasser och gränssnitt introducerar typer, så också generiska klasser och gränssnitt. Vad gäller?

- Om S :> T och C är någon klass så är *inte* C<S> :> C<T>, t.ex. en behållare med element S är inte supertyp till en behållare med element T!

  ```
  List<String> ls = ...
  List<Object> lo = ls; //FEL!
  ```

  Motivering: Om så skulle vi kunna göra;

  ```
  lo.add(new Object());
  // Assign Object to String
  String s = ls.get(0);
  ```

- Invariant subtypning gäller för generiska typer (jämför arrayer och kovarians)!

- En parametriserad typ är alltid subtyp till motsvarande råa typ t.ex.

  ```
  List :> List<String>

  List l;
  List<String> ls = ...;
  l = ls; // Ok.
  ls = l; // Warning!
  ```

16.7 Jokrar (Wildcards)

- Används bara vid instansieringar d.v.s. inte vid typdeklarationer (wildcard instantiations -> wildcard parameterized type).
- Hur skriva en metod som kan ta vilken sorts lista som helst? Vilken är typen för "vad som helst"? Följande går inte;

```
List<String> ls = ...

public myMethod(
        List<Object> lo ){
   :
}
// Bad call ls not subtype
o.myMethod( ls );
```

- Lösning: Supertypen för alla behållare är ;

```
// Wildcard instantiations
List<?> lu;
```

  ? kallas joker (wildcard) och typen "List of unknown". Löser problemet!

```
// Print any Collection
void printCollection(Collection<?> c){
   for (Object e : c) {
       System.out.println(e);
   }
}
```

- Dock, det enda vi vet om elementen i List<?> är att de åtminstonde är av typen Object! Detta får följder för vad man kan göra med List<?>! *Allt som kräver mer typinformation än så är otillåtet*[1]*!*
  - Kan inte instansiera "unknown"-behållare (new ArrayList<?>() FEL).
  - Kan aldrig stoppa in något i en unknown-behållare. Vet inte om det vi stoppar in är subtyp (eftersom typen är okänd).
- Naturligtvis är; List<?> != List<Object>
- ? kallas obegränsad joker (unbounded wildcard)
- Begränsade jokrar ger typsäker kovarians, se nedan.

Kod: wildcard

[1] Finns även något som kallas "wildcard capture" ett specialfall då man kan använda t.ex. en typ av Set<?> där en typ av Set<T> krävs (supertypen istf. subtypen).

16.7.1 Begränsade jokrar (bounded wildcards)

Generiska typer är invarianta, begränsade jokrar ger ökad flexibilitet. Jokrar kan bara ha en begränsning, alltså inte multipla som ovan vid .

- Detta gäller bara parametertyper![2]

16.7.2 Uppåt begränsade jokrar

(wildcards with upper bounds)

- Antag att vi har A :> B :> C samtliga med metoden doIt() (overridden). Vi vill ha en metod som kan ta listor av typ A, B eller C *och anropa doIt()* . Följande går inte;

```
// Works for List<A> only invariance
void doItAny(List<A> la) {
   for( .... ){
    la.doIt();
   }
}
```

  List<?> som parameter går inte heller eftersom vi då bara har typen Object att arbeta med.

- Lösning: Begränsade jokrar (bounded wildcards).

```
// Works for subclasses
void doItAny(
      List<? extends A> la){
   for( .... ){
    la.doIt(); //Ok!!
   }
}

List<B> lb = ...
doIt( lb ); // Ok
```

- I exemplet ovan är A övre *gräns för typparametern (upper bound).*

- <T extends Number> går inte, alltså T istf ?, ger ingen vettig infomation, se nedan... Kod: wildcard.upperbound

[2] "No wildcard type for return value! Wouldn't make the API any more flexible. Would force user to deal with wildcard types explicitly. User should not have to think about wildcards to use your API" //Josh Bloch

16.7.3 Nedåt begränsade jokrar

Antag A :> B och att vi vill ge en mängd en total ordning t.ex. MySet<B>. Kan göras genom att skicka in en komparator till konstruktorn (konstruktorn använder denna för att på något sätt sortera elementen). Hur skall konstruktorn se ut? Antag följande;

```
public class MySet<E> {...
    public MySet(Comparator<E> c){...
```

Konstruktorn kräver nu att att vi skickar in en Comparator<B> för MySet<B>, samma typparameter på båda ställena!!, ...men det skulle kunna gå lika bra med en Comparator<A> där A är superklass till B (kan mindra men spelar ingen roll, ev. en annan ordning men fungerar). Lösning;

```
public class TreeSet<E> {...
    public TreeSet(Comparator<? super E> c){...
```

- *E är lower bound.*

KOD generics.wildcard.lowe

16.7.4 Typer med jokrar och begränsade parametrar

Vilka super/sub-typs relationer finns det mellan instansierade generiska typer? Kan bli mycket komplicerat (exempel, se Angelica Langer). Vilken typ är super respektive subtyp av ...(kommer ej på tentan)?!?

```
Collection<
    ? extends Serializable>
List<? extends Number>
```

eller

```
Pair<? extends Serializable,? super Long>
Pair<? extends Number,Object>
```

16.8 Typradering

Generiska typer har av kompabilitetskäl implementerats m.h.a. s.k. typradering (type erasure).

- Typinformationen används vid kompileringen efter detta stryks den (statisk typinformation)! Motsatsen kallas "reifiable" types (typinformationen finns tillgänglig runtime).
- Generiska typer: Vid typradering ersätts typvariabeln med sin längst till vänster angivna begränsning (den första) eller Object om sådan saknas.

- Parametriserade typer; Ersätts med den icke-paramereriserade typen List<String> blir List, o.s.v.
- Generiska metoder; Alla förekomster av typvariabeln ersätt som vid typer.
- Kompilatorn lägger in explicita typomvandlingar för att få korrekta typer.
- Typraderingen kan ändra signaturtypen, kompilatorn genererar då specialla metoder för att återställa detta (bridge methods[3]).

KOD erasure

16.8.1 Konsekvenser

Alla generiska typer (List<T>) använder samma kompilerade klass (en klass per generisk typ, List<String> och List<Integer> är identiskt samma klass).

- Ingen typinformation finns tillgänglig runtime (till skillnad från t.ex. C++)! Följande är tillåtet

```
... = new T();
```

- Att cast:a är tillåtet med ger en varning (kan undertryckas). Innebär som vanligt en fara.

```
Object o = ...;
@SuppressWarnings("unchecked")
public E get(){
  return (E) o;// Really shure??
}
```

- Instanceof fungerar bara med reifiable typer (typer som har kvar runtime information). Exempel;

```
Object o = new LinkedList<Long>();
o instanceof List      //Ok
o instanceof List<?>   //Ok
o instanceof List<Long> //Error
o instanceof List<
     ? extends Number> //Error
o instanceof List<
     ? super Number> //Error
```

16.8.2 Typradering och reflection

Problemen ovan kan lösas m.h.a. reflection. Se kodexempel vid reflection.

Kod: reflection, erasure.solution

[3] Vilka normalt inte kan anropas direkt, kollas av kompilator.

16.9 Generiska metoder

- Förutom typer kan också metoder (och konstruktorer) vara generiska (både instans och statiska[4]).
- Anges med en typvariabel i vinkelparenteser innan returtypen, typvariabeln används i parametrar och/eller returtyper och i metodkropp.
- Behöver vanligen inte instansieras, kompilatorn härleder typen (typerna), se nedan.
- Kan inte använda jokrar som typvariabler (parametrar kan använda jokrar (returtyper ska inte använda jokrar)).

KOD methods

16.9.0.1 Overload

Följande är ok.

```
class SomeClass {
   // Object param atfer erasure
   public <T> void method(T arg){...}
   // Number after
   public <T extends Number> void method( T arg){...}
   // No erasure
   public void method( Long arg){...}
}
```

Metod ett och två kommer att ha olika signatur eftersom generiska metoder inkluderar parametrarnas "bounds" (Object och Number). Följande blir fel (får samma signatur efter radering (Number));

```
class SomeClass {
  public <T extends Number> void method( T arg){...}
  public void method( Number arg){...}
}
```

Kod: generics.overload

16.9.0.2 Override

Fungerar.

Kod: generics.override

16.9.1 Typhärledningar

(Type inference) Extremt komplicerat se JLS 15.12.2.7. Mycket förenklat finns...

[4] Typvariabeln har ju inget med klassen att göra!

16.9.1.1 Typhärledning från parametertyper

- Hur blir det med overloading av generiska metoder? Metoderna kan få flera olika signaturer!
- Kompilatorn härleder typargumenten baserat på de aktuella argumentens typer (vilka typer skall stoppas in istället för typargumenten).
  - Normalt härleds de mest specifika typer som möjliggör ett typriktigt anrop.

Kod: generic.methods.infer

- Kan leda till konstigheter då kompilatorn härleder en gemensam supertyp. Skydda mot detta genom begränsningar.

Kod: generic.methods.coun

16.9.1.2 Typhärledningar vid tilldelningar

(assignment contexts) Om kompilatorn inte kan få fram typerna m.h.a. parametrarna kan den ändå i vissa fall härleda typen;

- Om resultattypen finns i returtypen och det sker en tilldelning till en parametriserad typ. Exempel;

```
// No params!
public <T> Trap<T> makeTrap(){
  return new Trap<T>();
}

// Can infer T to Mouse
Trap<Mouse> mouseTrap =
        makeTrap();
// Can infer T to Bear
Trap<Bear> bearTrap =
        makeTrap();
```

- I icke-generisk kod är följande ekvivalent;

```
g( f(x) );

// Same as above (if non-generic)
tmp = f(x);
g( tmp );
```

  Men i generisk är det inte säkert g(f(x)) kompilerar. Ingen tilldelning finns, kan ev. inte härleda typer!

16.9.2 Explicit instansiering

I vissa fall kan inte kompilatorn härleda instansieringen av typparametrarna alls. Man måste då explicit ange den aktuella typen. Sker enlig;

```
// Explicit
o.<String>myMethod(...);
```

Kod: generic.methods.expli

16.10 Static

Typparametrar får inte förekomma i samband med static variabler. Alla parametriserade typer delar samma klassdeklaration, d.v.s. List<String> och List<Integer> skulle dela samma klassvariabel??!!

```
public final class X <T> {
  private static T field; // error
  public static T getField() {
    return field; } // error
  public static void setField( T t) {
    field = t; } // error
}
```

En generisk klass kan ha *vanliga statiska medlemmar*. Parameteriserade typen med jokrar är också tillåtet;

```
private static List<T> field; // No!
private static List<?> field; // Ok!
```

Kod: generic.static

16.11 Arrayer

Kan inte ha typvariabler för arrayer (finns ingen runtime information om vad T är).

```
// T gone after compiling
<T> T[] makeArray(T t) {
    return new T[100]; // error
}
```

- Elementtypen för ett array-objekt får inte vara en typvariabel eller en parametriserad typ. Runtime kontrollen av att man sparar rätt typ i arrayen kräver typinformation men denna typinformationen har ju raderats vid typraderingen.

  JLS 10.10 "If the element type of an array were not reifiable (§4.7), the virtual machine could not perform the store check described in the preceding paragraph. This is why creation of arrays of non-reifiable types is forbidden. One may declare variables of array types whose element type is not reifiable, but any attempt to assign them a value will give rise to an unchecked warning (§5.1.9)."

  Kod: array

16.12 Generiska undantag

Javas catch-gren fungerar inte med generiska klasser, d.v.s. vi kan int ha generiska undantagsklasser.

> "It is a compile-time error if a generic class is a direct or indirect subclass of Throwable. This restriction is needed since the catch mechanism of the Java virtual machine works only with non-generic classes."

17 Reflection

Wikipedias definition;

> "In computer science, reflection is the process by which a computer program can observe and modify its own structure and behavior. The programming paradigm driven by reflection is called reflective programming."

Mer konkret: Vi kan ta fram data om objekt (inte objektets data utan data om objektet = data om datan = metadata).

Reflection är inget man tar till direkt. Det används vid avancerad (speciell) utveckling t.ex. ramverk[1].

Fördelar Ger helt nya möjligheter.

Nackdelar Långsamt, ej typsäkert (ofta mycket casting), kan bryta mot inkapsling, måste matcha strängar exakt (vad händer då vi flyttar mellan paket o.dyl), m.m. Svårt att refaktorera kod.

> "There is no easy way (other than File Search) even in modern IDE's to know which attribute is referenced and where. This makes Refactorings much more complex (tiresome!) and error prone."

17.1 Reflection i Java

I Java har alla objekt möjlighet att komma åt sitt klassobjekt m.h.a. getClass() eller bara .class (får ett objekt av typen Class<T>). Objektet innehåller infomation om klassen t.ex. konstruktorer, metoder, fält (vi kan även komma åt private!!).

- För att använda .class måste man veta klassnamnet
- getClass() räcker med ett objekt.

Kod: reflection.basic

17.2 Instansiering med reflection

Man kan utifrån en sträng (klassens kvalificerade namn) instansiera nya objekt. Man kan även skapa objekt som implementerar ett känt interface.

Kod: reflection.creation

[1] Ett ramverk är en halvfärdig applikation. Tanken är att användare (programmerare) skall kunna anpassa ramverket till olika behov.

Att man kan skapa implementationer av interface gör att man t.ex. dynamiskt kan ladda ner kod och köra klassen behöver inte vara känd bara interfacet (jmf RMI). Dessutom säkerhetsproblem.

Kod: reflection.plugin

17.3 Reflection och generiska typer

Fungerar.

Kod: reflection.generics

17.3.1 Instansiering av generiska typer

Man kan inte direkt instansiera typer som berörs av typradering. Men med refletion och ett klass-objekt kan många situationer lösas.

Kod: reflection.erasure

17.4 Annotations typer

JSR175 till Java 5.0

> "An annotation type declaration is a special kind of interface declaration. To distinguish an annotation type declaration from an ordinary interface declaration, the keyword interface is preceded by an at sign (@). " /JLS 9.6

17.4.1 Default annotations

- @Override - indicates that the method should override a method in the superclass.
- @SuppressWarning – directs the compiler to suppress the specified warning.

In the package java.lang.annotation:

- @Documented – directs tools to automatically generate Javadoc for the annotated element (e.g. a method or variable).
- @Inherited – this indicates that the associated annotation is inherited by subclass of the current class.
- @Retention – indicates how long annotations with the annotated type are to be retained. For example, a retention type of RUNTIME indicates...
- @Target – This indicates the Java element to which associated annotations apply.

17.5 Egna annoteringar

1. Annotation declaration should start with an 'at' sign like @, following with an interface keyword, following with the annotation name.
2. Method declarations should not have any parameters.
3. Method declarations should not have any throws clauses.
4. Return types of the method should be one of the following: * primitives * String * Class * enum * array of the above types

Begränsningar

- Kan inte vara generiska
- extends inte tillåtet
- alla metoder parameterlösa
- ingen throws i metoddeklarationerna

Annotationstyper och namnrymnder fungerar som för klasser och vanliga interface. Det direkta super-interfacet för alla annotationstyper är annotation.Annotation.

17.6 Annoteringar i ramverk

I modern programutveckling används ofta s.k. ramverk t.ex. för dependecy injection. Många ramverk använder annoteringar för

Kod: framework

18 Programmering "by contract"

Det finns något som kallas klassinvariant. Viss förvirring råder. En del likställer den med RI. Se vidare nedan.

Användare En programmerare som använder kod utvecklad av någon annan. Har inte har tillgång till koden (endast .class-filerna)

Utvecklare En programmerare som utvecklar kod som någon annan ev. kommer att använda. Har full insyn i koden (modulens kod).

18.1 Notation

Vi vill på ett kortfattat och precist sätt uttrycka olika villkor. Finns tyvärr ingen allmänt accepterad standard. Vi använder naturligt språk och (frivilligt) en mycker förenklad form av JML (Java Modeling Language, se http://www.eecs.ucf.edu/~leavens/JML/) för att försöka uttrycka sanna påståenden (predikat).

- "Förkortad Java", utelämnar set/get/is/has som förled till metoder, parenteser m.m. se exempel. Använder ==, equals, relations och logiska operatorer (<=, &&, ||, ...).
- Mängdnotation (vanligen för tillståndet), {....} (en mängd), contains ($=\in$), empty ($=\emptyset$), + (union), \ (mängdsubtraktion),
- \old(...), betecknar klassen tllstånd precis *innan* denna metod exekverade. Parentesen avgränsar vad som skall betraktas i det gamla tillståndet.
- \result, betecknar resultatet av en metod (direkt efter metoden har exekverats).
- | ... | (belopp, längd, storlek) och * för upprepning (= 0 eller flera gånger).

18.2 Klassinvarianter

En klass är en beskrivning av vad som är gemensamt för alla objekt som tillhör klassen. Dessa objekt har naturliga invarianter, alltså saker som måste gälla för objekten. Alla "Personer" har en ålder, för denna gäller att ålderna är aldrig negativ. Icke-negativ ålder är en klassinvariant.

Klassinvarianter kan vara till nytta för användare av klassen/modulen.

- Ingår i den publika specifikationen.
- Exempel;

```
// Klasskommentar i gränssnitt
// @inv this.size >= 0
```

Aha!..Storleken för detta är >= 0 behöver aldrig kontroller i vår kod!

18.3 Specifikation

Begrepper "korrekt" förutsätter att vi har något att utgå ifrån, korrekt gentemot vad...? Vi måste bestämma vad vi menar med korrekt. Detta görs m.h.a. en specifikation.

- Specifikationen gäller bara klassens/modulens publika gränssnitt (publika metoder)! Specifikationen kan ses som ett *kontrakt* (gränssnitt) mellan utvecklare och klient. Om du använde det så här... så garanterar jag det här...
- Specifikationen skrivs ur två perspektiv:
  - Användarens (kallas ofta klienten). Vad måste användare veta för att kunna använda detta? *Man skall inte behöva förstå implementationen!*
  - Utvecklarens. För att kunna verifiera/testa måste vi bestämma hur det skall fungera! Måste specificera innan vi kan implementera.
- Specifikationen anger *vad* som görs inte *hur* det görs.
- Specifikationen kan skrivas;
  - på naturligt språk (enklare, oprecist, informell specifikation, jmf Javadoc).
  - som ovan. Svårare men mer exakt, se 18.1

Vad kan klienten vilja veta? Varje metod har (förhoppningsvis) ett tydligt namn samt parametrar (också med bra namn) och returtyp. Ofta räcker inte detta. Exempel;

- Påverkas någon klassinvariant?
- Är metodens returvärde by value eller by reference (delade objekt)?
- Modifieras parameterobjekt?
- Modifieras statiska variabler eller objekt?
- Ändras objektets tillstånd (måste anges på ett abstrakt sätt, implementationen skall vara dold).
- Om man tar bort något, förändras något annat t.ex. ordningen...?
- Generaras något undantag (i så fall vilket, återkommer...)?
- Modifieras några externa resurser (filer..)?

Antag att vi skall implementera en behållarklass. Hur hanterar vi t.ex. om man försöker lägga till samma värde som redan finns i en behållare (ett desingval) ?

- Vi tillåter dubbletter, ..ingen åtgärd.
- Det gamla värdet skrivs över.
- Vi använder ett returvärde som "signal" om lyckades eller ej (returtyp boolean t.ex.).
- Vi genererar ett undantag.

På motsvarande sätt, vad händer om vi försöker ta bort ett objekt som inte finns i behållaren?

- Inget händer.
- Kan generer undantag.
- Kan returnera null (om objektet istället fanns får vi en referens till det borttagna objektet).
- Returnera en tom lista.

Saker enligt ovan m.fl. skall vi försöka klargöra i specifikationen.

18.4 Specifikation i praktiken

Vi använder en metod som kallas "design by contract". För varje metod i klassen anger vi ett för och ett eftervillkor (många villkor med && mellan);

Förvillkor (precondition) Är ett predikat som måste vara sant precis innan metoden exekverar.

Eftervillkor (postcondition) Är ett predikat som är sant precis då metoden avslutas.

- Exempel från någon typ av behållare (parametrar får inte vara null, efter anropet är behållaren utökad med de nya värdena;

```
/**
*  @pre   key != null &&
*         value != null
*  @post  this.equals(\old(this) +
*                     {key,value})
*/
public abstract void add(
     String key, String value);
```

OBS! För och eftervillkor har inget med RI att göra, representationen är ju dold

18.5 Verifikation

Hur kan vi övertyga oss om att programmet utifrån specifikationen och invarianterna är korrekt?

Observation Ofta är skrivoperationer som är farliga kan leda till otillåtna tillstånd. Läsoperationer kan aldrig leda till *direkta* fel.

18.6 Verifikation av metoder

Verifiera att för och eftervilkor upprätthålls.

- Brott mot förvillkor ger IllegalArgumentException.
- Brott mot invarianter ger IllegalStateException.

18.7 Resonemang i praktiken

(i denna kurs) Görs alltså mot RI/klassinvariant, för- och eftervillkor.

- Motivera utifrån koden att invarianter etableras/upprätthålls och att för/eftervillkor uppfylls (alternativ hantera fel,...återkommer vis undantag).
  - Skriv korta kommentarer direkt i kod.
- Använd naturligt språk eller JML-liknande (logiska härledningar mycket svårt...).

18.8 Abstrakta datatyper

En abstrakt datatyp (abstract datatype, ADT[1]) är en typ tillsammans med operationer på denna (i vår fall ett gränssnitt och implementationen av detta (ev flera klasser)).

- Abstrakta datatyper döljer alltid representationen och implementering av operationer enl. ovan (därav abstrakta).
- Exempel på ADT:er är behållarna i The Collections Framework.
- Trie:en i Lab 1 kan också betraktas som en ADT.

18.8.1 Specifikation, Implementation och verifiering av en ADT

Vi skall titta på en Stack ADT. Kod: adt

[1] ADT är ett äldre begrepp än OO, det fanns ADT:er innan OO. Ofta avser man behållartyper.

18.9 Defensiv programmering

Ett annat begrepp, påminner mycket om det ovan.

"Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software in spite of unforeseeable usage of said software. "//Wikipedia