**Programmering Fortsättning**

# Föreläsning 3

Joachim von Hacht

## Innehåll


**1 Final** 2

**2 Static** 2
- 2.1 Klassmetoder . . . 2
- 2.2 Klassvariabler . . . 3
  - 2.2.1 Klasskonstanter . . . 3
- 2.3 Mixa klass och instans . . . 3
- 2.4 Static och gränssnitt . . . 3
  - 2.4.1 Static override av instansmetod . . . 3
- 2.5 Static och overload . . . 3
- 2.6 Design . . . 3
  - 2.6.1 Strängkonstanter och matchning . . . 4

**3 Initiering** 4
- 3.1 Init-metoder . . . 5
- 3.2 Design . . . 5

**4 Återanvändning** 5
- 4.1 Bibliotek . . . 5
  - 4.1.1 jar-filer . . . 5
- 4.2 Behållarklasser . . . 5
- 4.3 Behållare i Java . . . 6
  - 4.3.1 Generiska behållare . . . 6
- 4.4 Byte mellan behållare . . . 6
  - 4.4.1 Arrays och Collections 7
- 4.5 Traversering av behållare . . . 7
  - 4.5.1 Iterator . . . 7
  - 4.5.2 Iterable . . . 7
  - 4.5.3 ListIterator . . . 7
  - 4.5.4 ConcurrentModification Exception . . . 7

**5 Semiformella resonemang** 8
- 5.1 Notation . . . 8
- 5.2 Representation . . . 8
- 5.3 Invarianter . . . 8
  - 5.3.1 Representationsinvarianter . . . 8
  - 5.3.2 Klassinvarianter . . . 9
- 5.4 Specifikation . . . 9
  - 5.4.1 Specifikation i praktiken . . . 10
- 5.5 Verifikation . . . 10
  - 5.5.1 Klassifikation av metoder . . . 10
  - 5.5.2 Verifikation av invarianter . . . 10
  - 5.5.3 Verifikation av metoder 11
  - 5.5.4 Resonemang i praktiken . . . 11

**6 Representation-exponering** 11
- 6.1 Defensiv kopia . . . 11

**7 Abstrakta datatyper** 11

7.1 Specifikation, Implementation och verifiering av en ADT . . . . . 12

**8 Defensiv programmering 12**

**9 Sammanfattning 12**

# 1 Final

Final *i samband med variabeldeklaration* innebär att värdet inte kan ändras (en konstant, ofta i kombination med static, se nedan).

- 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!
- En intressant egenskap med final; Om en variabel är final måste den initieras, annars kompileringsfel. Vi är alltså garanterade att referensen är initierad (eller undantag). T.ex.

```
public class Car {
  // Always has an engine,
  // else exception
  private final Engine engine;
  public Car(){
    engine = new Engine();
  }
}
```

- Vissa använder final till parametrar (kan inte ändra av misstag). Omdebatterat, kan kladda till koden med en massa final.

```
public void doIt(final Object x,
                 final Object y){
  // Can't assign x, y
}
```

- Final återkommer i en annan betydelse i samband med arv...

**Observation** Final minskar komplexiteten, vi kan inte ändra denna del av tillståndet. Överväg alltid möjligheten.

# 2 Static

Normalt innebär static att det handlar om en klass inte ett objekt (själva ordet static kommer från C/C++, mindre lyckat..ointuitivt).

## 2.1 Klassmetoder

(kontra: instanmetoder).

"A method that is declared static is called a class method. A class method is always invoked without reference to a particular object. An attempt to reference the current object using the keyword **this** or the keyword **super** or to reference the type parameters of any surrounding declaration in the body of a class method results in a compile-time error. It is a compile-time error for a static method to be declared abstract".//JLS 8.4.3.2

- Statiska metoder är inte polymorfa (ingen overriding) Vi kan säga att static metoder bestäms compile time[1][2].

[1] Under kompileringen "markeras" att anropen skall ske static, d.v.s. runtime miljön skall inte leta bland objekt för att hitta metoden.

[2] Om det finns ett objekt i samband med anropet, beräknas vilket objekt det är men resultatet kastas bort.

```
// Polymorfism means behaviour depending
// on runtimetyp of object
// But static has no runtime-type (type is known)
Math.srqt(77);  //  Type is Math !!
```

## 2.2 Klassvariabler

Klassvariabler (kontra: instansvariabler). Static fält finns i metodarean i JVM:en de hör inte ihop med instanserna på heap:en.

### 2.2.1 Klasskonstanter

Inget problem de är ju final. Använd ofta och gärna! (Code smell; Värden i koden = hårdkodat, ersätt med konstanter).

KOD _static

## 2.3 Mixa klass och instans

- Static-metoder kan inte använda/anropa instans- variabler/metoder (vilket objekt avsees? Finns inget!).
- Instansmetoder kan använda/anropa static (alla instancer känner till sin klass).

KOD _static.mix

## 2.4 Static och gränssnitt

- Metoder: static inte tillåtet (static är inte polymorfa, kan inte override:a).
- Variabler; static är tillåtet (alla variabler är default static final). Använd ej, bad practises.

KOD _static.iface

### 2.4.1 Static override av instansmetod

Kan vi override:a en instansmetod (från ett gränssnitt) och göra den static i implementationen? ...nej! static kan inte override en instance.

KOD _static.override

## 2.5 Static och overload

- Static metoder kan overload:as.
- Instance-metod kan inte ha samma signatur som klass metod (klassmetoden gäller ju alla instanser, d.v.s. static räknas inte till signaturen).

KOD _static.overload

## 2.6 Design

static används till;

- Gemensam funktionalitet eller data som delas mellan alla instanser. En static variabel påverkar alla objekt tillstånd, mycket dåligt, ökar komplexiteten! Undvik!
- Konstanter, static final (vanligt och bra).
- Constants-klasser, vanligt att samla alla applikationsglobala konstanter i en klass som döps till Constants (innehåller bara public static final deklarerade variabler).
- Funktionssamlingar t.ex. Math. och utility-klasser, vissa design mönster, Factory/Factory method m.fl.
- Ofta lägger man "parametervärden" till metoder i klassen som konstanter (så slipper användaren hålla reda på dessa

man bara väjer bland givna...troligen bättre med enum).

- Överanvändning kan vara en code smell, ev. för mycket procedurell programmering.

### 2.6.1 Strängkonstanter och matchning

Ofta har man behov av ett namn (en sträng) på flera olika ställen i ett program (t.ex. hämta ur en Hashtabell...). Att hårdkoda strängen (skriva direkt i koden) är ett dåligt alternativ, felstavning leder till runtime fel. Ett sätt är att skapa konstanter för strängarna m.h.a. static final och en Constants-klass.

Ett annat (bättre) alternativ är att använda enum. Fördelar;

- Att ha strängar till namn (nycklar) kan ersättas med typsäkra namn.
- Då vi verkligen måste ha en sträng kan man använda toString() metoden för enum-värdet (och valueOf() om man vill åt andra hållet).

KOD _static.constants

## 3 Initiering

Eftersom vi hanterar tillstånd är det naturligtvis mycket viktigt att vi sätter ett vädefinierat starttillstånd för "allt". Initiering sker inligt följande

- initiering av klassvaribler och exekvering av static initializers (d.v.s saker gemensamma för alla objekt initieras). Sker i "textually apperant order".

```
// Class variable
private static int i = 99;

// Static initializer
static {
   // init things here
   // code...! loops...[3]
}
```

static har med klassen att göra kan inte ha private, m.m. i static initializers

- Då objekt skapas (senare) gör initering av instansvariabler och instance initializers exekveras. Därefter körs konstruktorn[4]. Också i textually apperant order (konstruktor körs sist).

```
// Instance variable
private int i = 99;

// Instance initializer
{
   // init things here[5]
}
```

- Delobjekt instansieras och initieras först. Exempel;

```
class A {
  //...then
  private B = new B();
  //First...
  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.

[3] Except return-statement.

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

[5] Implicit kopierat in i varje konstruktor.

### 3.1 Init-metoder

Tyvärr kräver en hel del i Java att det finns en parameterlös kontruktor (default) t.ex. reflection, .... Vi kan då få problem med att ge objektet ett väldefinierat starttillstånd. I dylika fall får man skapa en init metod. Att initailisera på detta sättet är ofta mindre bra, undvik.

KOD init.method

### 3.2 Design

Kontruera objekt så att det alltid har ett väldefinierat tillstånd (static/object initializers, kontruktor, init-metoder). Argument måste kontrolleras! Se även defensiv kopia och representationsexponering, senare.

## 4 Återanvändning

(reuse) En stratategi. Om vi kan återanvända tidigare utvecklad (förhoppningsvis) korrekt kod **utan att behöva förstå detaljerna** [6] så minskar vi komplexiteten.

**Observation** Återanvändning minskar komplexiteten.

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

### 4.1 Bibliotek

Ett bibliotek består av ett antal återanvändbara (generella) moduler. En Java implementation (Java SE, standard edition) innehåller ett stort antal standardbibliotek java.lang, m.fl. m.fl ( $>$ 4.000 klasser)

Utöver detta finns en oerhörd massa Java-bibliotek för i princip allt man kan tänka sig att hämta på nätet (ofta gratis). Leta alltid först!

#### 4.1.1 jar-filer

I Java skapas bibliotek som .jar-filer (bin-katalogen med class-filer m.m. packas ihop ungefär som zip).

### 4.2 Behållarklasser

Man har 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 classes). En tabell över procentuella antalet klasser i några större applikationer som *använder* behållare [7].

| Applikation | Antal klasser | 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 femtdel av klasserna använder behållare, d.v.s. de är lämpliga kandidater får återanvändning.

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

- Mängd - add, remove, inga dubbletter (som matematisk mängd).

[6] Kräver högklassig dokumentation. Dokumentation är oerhört viktig!

[7] Discovering and Debugging Algebraic Specifications for Java Classes, Johannes Henkel.

- Lista - add(index), remove(index), find(index), ...(en ordnad sekvens)
- Stack - pop, push, top, (LIFO, last in first out)
- Kö - enqueue, dequeue (FIFO, first in first out).
- 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...Inga dubbletter för key's tillåtna .

BILD

## 4.3 Behållare i Java

Alla behållarna ovan finns implementerade i The Java Collection Framework. http://java.sun.com/j2se/1.5.0/docs/guide/collections/. De grundläggande (översta) gränssnitten är;

- java.util.Collection.
- java.util.Map (tabell)

Följande gränssnitt extends Collection.

- java.util.Set
- java.util.List
- java.util.Queue

Ett antal konkreta klasser implementerar "kontrakten". Några är;

- java.util.Stack (bygger på Vector som inte är ett gränssnitt, dåligt,...).
- java.util.ArrayList och LinkedList implementerar List.
- java.util.HashMap implementerar Map[8].
- java.util.HashSet implementerar Set.

### 4.3.1 Generiska behållare

Behållare för vad? Man vill inte implementera en behållare för varje typ av objekt (Integer lista, String lista, o.s.v., vill kunna använda samma kod, reuse...).

- Lösning i Java[9] är generiska (generic) behållare (klasser). Man anger för varje klass m.h.a. en typparameter vilken sorts objekt behållaren kan hantera. Exempel[10];

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>();
// Typeparametrar String
Map<String, String> m =
       new HashMap<String, String>();
```

KOD container

## 4.4 Byte mellan behållare

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

[8] Hash, hacka sönder

[9] Finns i många andra språk också C++ t.ex.

[10] Innan Java < 1.5 fanns inte generiska klasser, alla behållare använda då typen Object. Användaren fick sedan göra explicita typkonverteringar.

KOD container.change

### 4.4.1 Arrays och Collections

Två hjälpklasser som kan utföra divers saker med arrayer och Collections.

> "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

> "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

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

## 4.5 Traversering av behållare

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

### 4.5.1 Iterator

Iterator är ett designmönster (och ett gränssnitt i Java) som gör det möjligt traversera en behållar-ADT på ett abstrakt sätt (utan att veta hur den är konstruerad). Iteratorer i Java implementeras som inre klasser som implementerar gränssnittet Iterable, återkommer...;

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

### 4.5.2 Iterable

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

```
// 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.
- Vanliga arrayer implementerar inte Iterable men kan traverseras med den korta for-loopen ändå (JLS 14.14.2)

### 4.5.3 ListIterator

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

### 4.5.4 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().

- OBS! Kan t.ex. inte ta bort något i en for-loop.

KOD container.traverse

# 5 Semiformella resonemang

En strategi. Definitioner;

**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).

## 5.1 Notation

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).

## 5.2 Representation

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

- En person kan avbildas som en klass (typ) med fälten String fName, String lName o.s.v.
- En kö kan avbildas som en kö-klass där själva kön representeras som en array. Operationerna enqueue och dequeue innebär att man skall lägga repektive ta ut element ut fältet på ett visst sätt.

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

- Representationen skall vara dold för användare av klassen/modulen. Infomation hiding!
- Byte av representation räknas som en acceptabel (tillåten) desingförändring (transformation), d.v.s. bytet skall inte påverka resten av programmet.

## 5.3 Invarianter

### 5.3.1 Representations-invarianter

- En representationsinvariant (RI) är en specifikation för utvecklare som beskriver vilka tillstånd den *valda representationens* får befinna sig i. Då man arbetar med RI känner man till alla impl. detaljer.
- Ingår inte i publika gränssnittet (representationen är ju dold).

- Måste välja representation först.
- Exempel från en Trie-klassen (se lab1, implicit && mellan raderna)

```
// this.root != null
// this.root.parent == null
```

### 5.3.2 Klassinvarianter

Klassinvarianter är påståenden om klassers tillstånd som 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!

## 5.4 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 5.1

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;

- 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.

### 5.4.1 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

## 5.5 Verifikation

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

### 5.5.1 Klassifikation av metoder

Inte heltäckande finns andra typer (eller blandingar av nedan).

- Konstruktorer (constructors, ctors) Skapar nya objekt.
- Mutator (setter). Ändrar klassen tillstånd.
- Accessor (accessor, getter, observer) En metod som gör det möjlig att avläsa klassen tillstånd (en del av).

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

Undantag, se representationsexponering senare.

### 5.5.2 Verifikation av invarianter

(Induktiva resonemang) Utifrån RI och klassinvarianter kan man föra induktiva resonemang[11].

- Om RI/klassinvariant etableras av konstruktor (eller init-metod)...

[11] Jmf. Induktionsbevis, basfall, induktiva fall

- ...och ∀muterande metoder; RI och klass inv. upprätthålls...
- ...så kommer objektet alltid att befinner sig i ett tillåtet tillstånd.

#### 5.5.3 Verifikation av metoder

Verifiera att för och eftervilkor upprätthålls.

#### 5.5.4 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...).

## 6 Representation-exponering

(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 kompleiteten, förstöra möjlighet att resonera utifrån invarianter. Lösning: Defensiva kopior (eller icke-muterbara objekt, återkommer...)

BILD

- Av effektivitetsskäl tillåter man ibland representationsexponering t.ex. Iterator (slipper kopiera).

### 6.1 Defensiv kopia

- Defensiv kopia = Man kopierat referensvärden, parametrar till konstruktor och/eller returvärden och använder dessa d.v.s. man ändrar från referenssematik till värdesemantik.
- Finns också stöd i Java, unmodifiable....(List, Set, Collection). Ger en skrivskyddad (kopia) version av behållaren (elementens tillstånd kan dock ändras);

```
return Collection.
 unmodifiableCollection(repr);
```

KOD defcopy

## 7 Abstrakta datatyper

En abstrakt datatyp (abstract datatype, ADT[12]) 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 implementring 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.

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

### 7.1 Specifikation, Implementation och verifiering av en ADT

Vi skall titta på en Stack ADT.

KOD adt

## 8 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

## 9 Sammanfattning

- final minskar komplexiteten, värden kan inte ändras.
- static handlar om klasser inte enskilda objekt kan användas till, variabler och metoder som delas av alla instanser och/eller konstanter, Utility/Constants-klasser.
- Alltid försöka att initiera till ett giltigt starttillstånd.
- Återanvändning minskar komplexiteten och snabbar upp utvecklingen, ökar chansen att utveckla högkvalitativa applikationer.
- Att utveckla återanvändbar kod, bibliotek, är komplext;
  - Skall vara lättanvänd.
  - Måste vara korrekt.
  - Kräver specifikation, verifikation och testning.
- Representationsexponering måste hanteras ev. med defensiva kopior.
- En abstrakt datatyp är en typ med tillhörande operationer där den konkreta implementationen är dold.