Programmering Fortsättning

Föreläsning 6

Joachim von Hacht

Innehåll

1 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).

1.1 Abstrakta klasser kontra gränssnitt

- Vi programmerar alltid mot gränssnitt, men...
- ...om flera klasser implementerar ett gränssnitt och *dessutom* har duplicerad kod gör vi enlig följande;

BILD UML

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

KOD abstractclass

1.1.1 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).

2 Designmönster

2.1 Template

Idé; Lägg huvudalgoritm i abstrakt basklass, låt subklasser definiera egna delar av algoritmen.

BILD UML

KOD template

2.2 State

Mönstret används då ett program har ett antal tyliga övergripande tillstånd t.ex. uppkopplad/nedkopplad, öppen/stängd o.dyl.

(kan vara många tilllstånd, mycket komplext). Idén är att låta ett programs övergripande tillstånd representeras av klasser/objekt (en klass/tillstånd). Eventullet kan en abstrakt klass användas. Ersätter kod såsom (double switch idiom);

```
switch( state ){
case CONNECTED:
   switch( command ){
   case CONNECT:...
   case SEND: ...
   case LISTEN:
   default: ...
   }
case DISCONNECTED:
   switch( command ){
   case CONNECT:...
   case SEND: ...
   case: LISTEN:
   default: ...
   }
default:...
}
```

State kan använda en abstrakt klass. OBS! Att det finns två kategoriera av metoder; sådana som byter tillstånd och sådana som utför något i det aktuella tillståndet.

BILD UML

KOD state

3 Adapterklasser

Adapterklasser påminner om Abstrakta klasser (men de är inte abstrakta), förekommer bl.a. i Swing t.ex. MouseAdapter.

- Syftet med en adapterklass är att tillhandahålla färdiga (tomma) implementationer för något gränssnitt. Användaren subklassar och overrides de metoder som behövs (spar arbete).

KOD adapter

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.

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

KOD innerclasses.swing

BILD

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!!!
- Vissa saker fungerar inte med inre klasser t.ex. clone, återkommer...

4.1 this

Vad this är i koden beror på var det står, i inre eller yttre klass (alltså inte som vid arv där this alltid syftar på runtime objektet). *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

4.2 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.

4.3 Arv

Inre klasser kan ärva eller ärvas, skall nog undvikas...(dock ett exempel, bizzare)

KOD innerclasses._static

4.4 Design

Användning av inre klassser är specialfall, troligen undvik...

- 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).

[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"...

- 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

5 Introduktion till generisk klasser

Innan Java 1.5 fanns inga typsäkra behållare.

- I en sådan kan man stoppa in vilka typer som helst (typomvandlas till Object), det är upp till programmeraren att hålla rätt på saker och ting (heterogena objektsamlingar).

KOD preonefive

- Kan koda på detta sätt även i Java 1.5 eller senare (bakåtkompabilitet). Kallas råa types (raw), kompilator varnat. Undvik!
- Generiska klasser (generic/parameterized klasser) introducerades bl.a. som ett sätt att få typsäkra behållare (homogena objektsamlingar);
- En generisk klass är en klass med en eller flera typparametrar (ibland kallat typvariabler) av referenstyp, primitiva typer är *inte* tillåtna.
- Typsäkerheten kan nu kontrolleras vid kompileringen

```
List<String> s =
    new ArrayList<String>();
// Won't compile
s.add(new Integer(3));
```

- Samtliga klasser i Java Collection FrameWork är generiska (skrevs om för Java 1.5).

5.1 Deklaration av generiska klasser

- Då klassen deklareras anger man typvariabler (typparametrar) inom vinkelparenteser (traditionellt används en enda stor bokstav, T (type), E (element) m.fl)[4].

```
public class A<T> {...
```

5.2 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>();
```

KOD generics.basic

[4] Enum och inre anonyma klasser kan inte vara generiska.

5.3 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);
}
```

KOD generic._iface

5.3.1 Användning av generiska gränssnitt

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

- 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;
}
```

5.4 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).
- För långtgående 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++)!
  - Inget av följande är tillåtet

```
... = new T();
... = new T[...];
if( ref instanceof 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??
}
```

- Mer senare...

6 Undantag

(JLS 11) Undantag (Exceptions), undantagshantering (exception handling).

- Viktigt, stort, svårt och omtvistat område!
- Grundproblem: Hur skall programmet reagera på oväntade, extrema, felaktiga, konstiga händelser (utanför programmets ansvarsområde)??
- Om programmet klarar dessa påfrestningar är det "robust".

Några orsaker till undantag;

- Programmeringsfel (NullPointerException, IllegalArgumentException,...).
- Användaren av vår kod har inte läst dokumentationen, anropar med fel...
- Resursfel; DVD:n var inte isatt, nätverket gick ner, hårddisken är full, minnet tog
- slut, databasen har kraschat, ...

6.1 Manuell undantags hantering

Äldre (sämre) metod: Använd returvärden e.t.c.. ger mycket grötig kod (bryter mot separation of concerns) exempel;

```
// 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 {
  :
}
```

Om man plötslig vill ha felkontroll i en void-metod måste ev. hela anropskedjor skrivas om. Returvärdena är upptagna (kan lösas med s.k. flaggor). Kan det vara svårt att hitta vettiga "fel"-värden. Olika programmerare gör på olika sätt ingen standard...

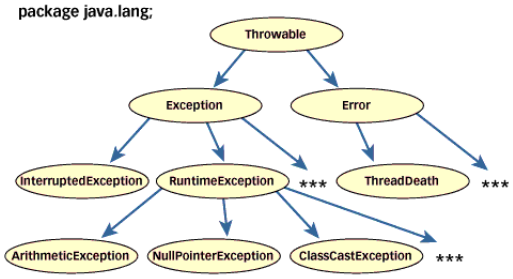
6.2 Javas undantagshantering

Java har en standardiserad felhanteringsmekanism (exception handling)

> "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
>
> "Programs can also throw exceptions explicitly, using throw statements" //JLS 11

6.3 Undantagsklasser

Ett undantag representeras i Java som ett objekt ur följande hierarki (basklass Throwable).



- Error-grenen är reserverade för Java och JVM:en används inte av oss.
- Exception representerar fel som vi eventuellt skall kunna hantera.
- Vi kan även skapa egna undantagsklasser som subklasser till Exception. Ofta gör man detta för att få undantaget till rätt abstraktionsnivå, se nedan.

6.4 Checked och unchecked exceptions

6.4.1 Checked exceptions

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 har varit att tvinga programmeraren att ta ställning till tänkbara fel.

6.4.2 Unchecked exceptions

Klassen RuntimeException och alla underklasser tilldenna.

- Hantering kontrolleras inte av kompilatorn.

KOD exception

6.4.3 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,se nedan),
  - Instabila metodsignaturer se 6.6.
  - 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.
  - Exeptionhandling är (mycket) resurskrävande.

Se vidare 6.8.

6.5 Att hantera undantag

6.5.1 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[5].

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

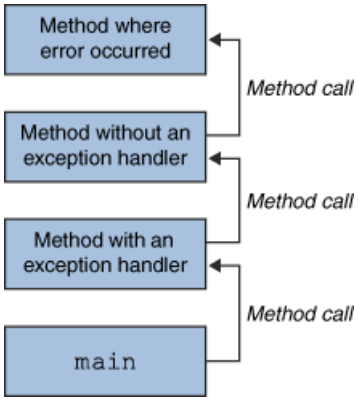
```
- a pushas på stacken anropar b
- b pushas anropar c
- c pushas.
- c klar popas, programmet fortsätter med b
- b klar popas, fortsätt med a
- a klar popas.
```

6.5.2 Exekveringsförlopp

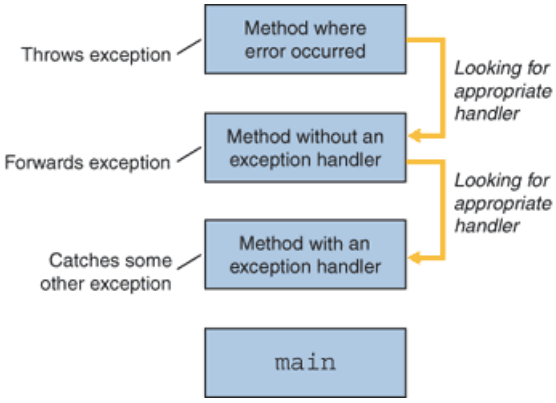
- Då ett undantag uppstår skapas en instans av Throwable. Det normala flödet avbryts och programmet vandrar upp genom anropsstacken (unwind the call stack). Hanteras inte undantaget någonstans kommer programmet att avbrytas (med en felutskrift). Antag metodanrop enlig

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

bild;



Flödet i felhanteringen blir;



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).

```
try {
   //Perhaps exception here...
   a.doSomeCall();
   :    // This is NOT executed
catch( ExceptionClass1 e ){
  //...if so jump directly
  // to here (the handler)
  :
catch( ExceptionClass2 e )
  // ..or here
  :
  // more catch as needed
} finally{
  // always executed
  // (use for clean up)
  :
}
```

Variant utan finally

```
try {
   //Perhaps exception here..
   a.doSomeCall();
   :
catch( ExceptionClass1 e ){
  //..if so jump directly
  // to here (the handler)
  :
} ...
```

Variant utan catch (OBS! undantaget är inte hanterat)

```
try {
   //Perhaps exception here..
```

```
    a.doSomeCall();
    :
finally{
  :
} ...
```

KOD exception._finally

- finally delen är mycket användbar eftersom den är garanterad att köras. Kan användas för att frigöra resurser (annars resource leak) och återställa objekt till väldefinierade tillstånd.

6.6 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[6].

KOD f5_2.streams.ReadWriteTextFile.getContents()

KOD exception.inherit

6.5.3 throws

- Om man inte hantrar en checked exception direkt kan man skicka det vidare till anropande metod m.h.a. throws, i signaturen se 6.6.

```
public void myMethod()
            throws IOException {
    :
    // Somewhere a call that
    //can cause an IOException
    // (no try..catch here)
    :
}
```

KOD f5_2.streams.ReadBinary

- Då en felhanterare har körts klart räknas felet som "hanterat" och det normala programflödet fortsätter (direkt efter felhanteraren).

6.7 Att kasta egna undantag

Vi kan låta våra program kasta ett undantag (checked och unchecked) programmatiskt (i kod) t.ex. då någon metod anropas och programmet inte vet hur det skall fortsätta. Görs med reserverade ordet throw (i metoden som inte vet vad den skall göra...). Därefter skapas en instans av en Throwable-klass (vilket undantag som skall genererar). I konstruktorn skall man skriva en så tydlig och förklarande text som möjligt, vad har hänt, var (kan generera egna eller Javas undantag, dessutom kan de vara checked eller unchecked, se nedan)?

```
throw
  new IOException(
```

[6] 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

```
        "File not found");
```

I anropande metod;

```
try{
 : // Call metod here
} catch(IOException e){
 : // Handle exception here
}
```

KOD f3.adt

6.7.1 Failure atomicity

Vid kontroll av t.ex. pre för metoder kastar man alltid undantaget direkt, innan man gjort något. Innebär att objektet behåller sitt tillstånd även då ett fel uppstår (den tillståndsförändrande koden körs inte).

```
// @pre i >= 0
// ...
public void doIt( int i ){
   if( i < 0 ){
     throw new
       IllegalArgumentException(
             "Negative value...");
   }
   // Chang state here
```

- Tillståndsförändringar görs alltid så sent som möjligt!

6.8 Exception tunneling

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

Kombinationen try ..finally och exception tunneling verkar lovande: Undantaget hanteras inte (ingen catch) utan vandrar uppåt. På varja abstraktionsnivå kan (vid behov) try..finally användas för att köra "återställningskod" (kod som kommer efter undantaget och normalt inte skulle ha exekverats). Slutligen fångar man undantaget på lämplig plats (stoppar vandringen).

BILD

KOD exception.tunneling

6.9 Design

Mycket att tänka på...!

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

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

- 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)! Se 6.8.

- Man får försöka separera ut undantagshanteringen så mycket det går. Eventuellt (troligen) skapa ExceptionHandler-klasser. Centraliserad visning av t.ex. dialogrutor är också önskvärt, man skall inte behöva gå in i GUI koden för att ändra ett felmeddelande.

- 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) se även 6.8

- Undantaget skall presenteras på rätt abstraktionsnivå, ett lågnivåundantag, t.ex. nätverket fungerar inte, bör översättast till något relevant för den nivå som hanterar undantaget. Innebär att man får paketera om undantag genom att fånga dem och sedan kasta nya (ev. egna, applikationsspecifika undantag objekt, re-throw).

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

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

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

- Om man anger en checked exception i ett gränssnitt tvingar man eventuellt användaren (programmera mot gränssnitt) att hantera undantag i onödan, implementationen kanske inte kastar något undantag..! Undvik! Undantag kontrolleras vi kompilering, runtimetyper har inget med detta att göra.

```
// 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 hantera 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.

- Exceptions skall användas för undantag inte för att styra normalt flöde.

7 Sammanfattning

- Abstrakta klasser kan komma till användning för att undvika kodduplicering.

- Inre klasser är praktiska vid vissa tillfällen.

- Javas felhantering löser problemen ovan med manuell hantering. Vi separerar fel-kod och logik (delvis i alla fall), sökningen efter felhanterare sker automatiskt och undantagsobjektet kan innehålla mer information om felet (mer än bara true, -1 eller dylikt).
- Alltid failure atomicity vid exceptions.
- Checked vs unchecket är en öppen fråga..? Använd ev exception tunnerling.

Entity	Declaration Context	Access-ibility Modifiers	Outer Instance	Direct Access to Enclosing Context	Defines Static or Non-static Members
Package-level class	As package member	`public` or default	No	N/A	Both static and non-static
Top-level nested class (static)	As static class member	all	No	Static members in enclosing context	Both static and non-static
Non-static inner class	As non-static class member	all	Yes	All members in enclosing context	Only non-static
Local class (non-static)	In block with non-static context	none	Yes	All members in enclosing context + local final variables	Only non-static
Local class (static)	In block with static context	none	No	Static members in enclosing context + local final variables	Only non-static
Anonymous class (non-static)	As expression in non-static context	none	Yes	All members in enclosing context + local final variables	Only non-static
Anonymous class (static)	As expression in static context	none	No	Static members in enclosing context + local final variables	Only non-static
Interface	As package member or static class member	`public` only	N/A	N/A	Static variables + non-static method prototypes

Figur 1: Sammanfattning deklarationer.