Överlagring, static, testning, formella metoder och undantag

Förelasning 13

TDA540 – Objektorienterad Programmering

CHALMERS

Gränssnitt igen

För att kunna ändra på olika delar av programmet utan att andra delar påverkas använder man gränssnitt (variabler av gränssnittstyp)

- Vi har talat om List (ArrayList, LinkedList)
- Tänkbart är också att lägga ett gränssnitt mellan modellen och GUI
- GUI:et använder bara metoder från gränssnittet …
- … om vi vill ändra implementationen behöver inget ändras i GUI:et
- Ger dessutom en bra sammanfattning av alla metoder i top-level klassen
- Den generella idén med gränssnitt är att avgränsa programmet i separata delar som inte påverkar varnadra

Namngivning

Ett oväntat problem när man programmerar är att hitta bra namn på allt (som förklarar vad saker står för)

- Har man hittat ett bra namn vill man kunna använda detta på flera ställen (inte hitta på nya hela tiden …)
- Synlighetsområden ger oss den möjligheten
- Kan förekomma att man vill använda samma namn på metoder inom samma synlighetsområde (normalt inte tillåtet)
- Dock, ... tillåtet i Java om metoderna har olika parametrar (antal och/eller typer). Returtyp spelar ingen roll
- Under kompileringen bestäms utifrån parametrarna vilken av metoderna (med samma namn) som skall köras
- Detta kallas: **Överlagring (overloading),** olyckligtvis väldigt likt överskuggning (som ju är något helt annat)

Användning Överlagring

Typiskt metoder som gör i princip samma sak men med olika antal och/eller typer på parametrar (options)

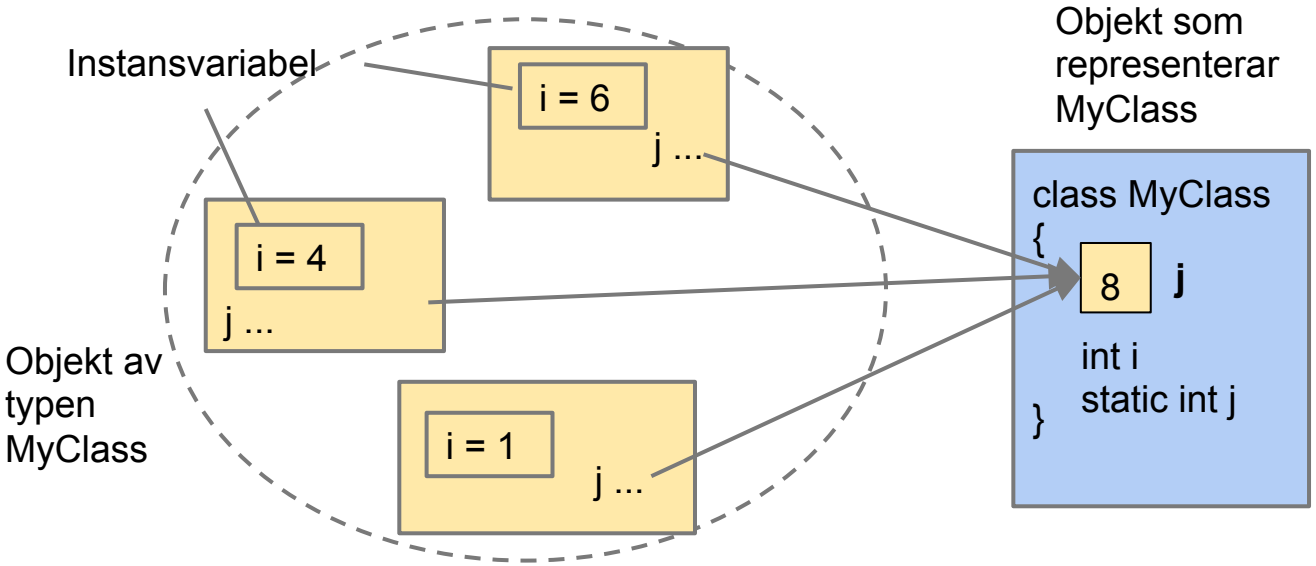
Exempel String … många indexOf-varianter
- int indexOf(int ch)
- int indexOf(int ch, int fromIndex)
- int indexOf(String str)
- int indexOf(String str, int fromIndex)

Exempel List … 2 remove varianter
- E remove(int index) (E är någon typ, List är generisk)
- boolean remove(Object o)

Instans och Klassvariabler

En **klassvariabel** (**static**) är gemensam för alla objekt av typen

- Anges med static vid deklarationen
- Variabeln existerar i objektet som representerar klassen
- Kan refereras från alla instanser



Klassvariabler

Användning av klassvariabler

- Eftersom variabeln delas av alla kan vilket objekt som helst ändra värdet för alla andra (gör ett objekt fel drabbas alla andra)
- En klassvariabel skall vara final….
- … d.v.s. ett fixt värde som kan användas på flera ställen i programmet
- Kallas **konstant** (ett namn på ett värde)
- Alla hårdkodade värden skall normalt ersättas med konstanter
- Skrivs med stora bokstäver ("_" för sammansatta ord)

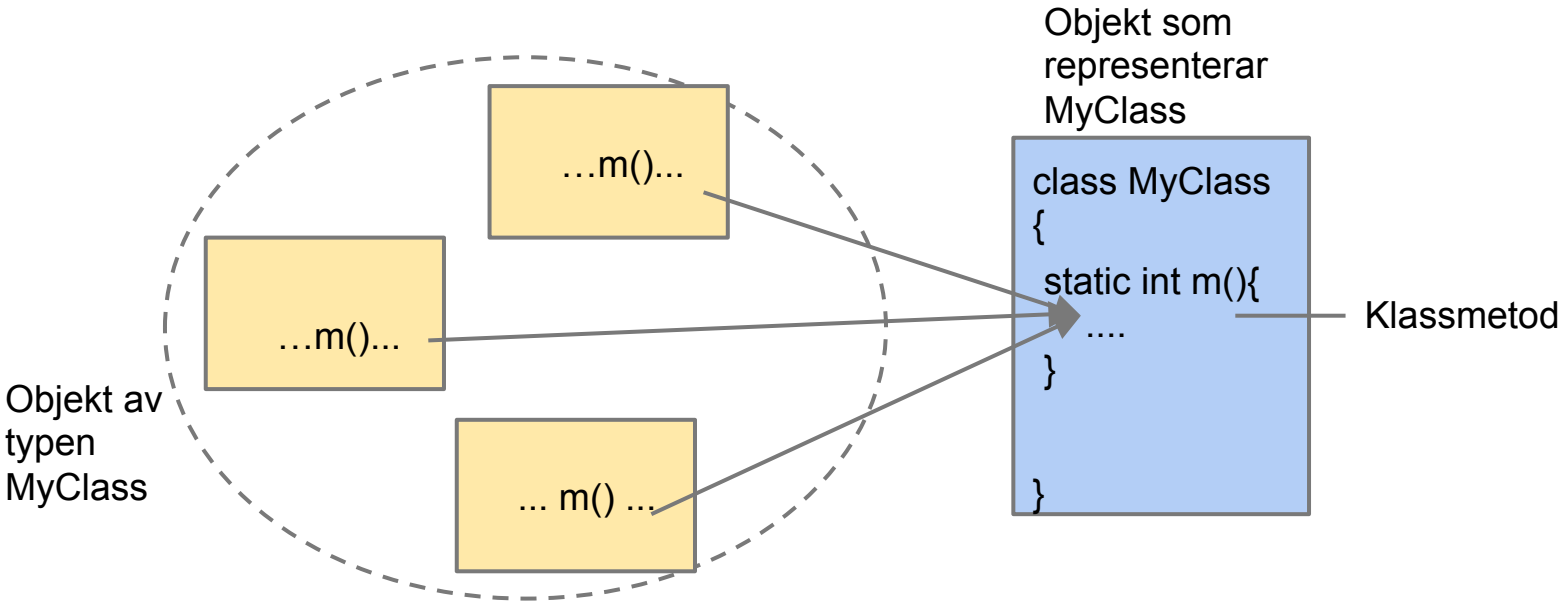
```
// In MyClass, declare constant
public static final int MY_CONSTANT = 42;

// Access, no object needed, just class
… = MyClass.MY_CONSTANT;
```

Klassmetoder

En **klassmetod** (**static**) är gemensam för alla objekt av typen

- Anges med static vid deklarationen
- Metoden existerar i objektet som representerar klassen
- Kan refereras från alla instanser



Instans och Klassmetoder

En instansmetod, kan använda instansvariabler och klassvariabler

- Instansmetod måste anropas på ett objekt, det objekt vars instansvariabler metoden (eventuellt) skall använda
- Kan anropa andra instansmetoder eller klassmetoder
- Kan använda this
- Kan överlagras och överskuggas

En klassmetod (**static**) kan bara använda klassvariabler

- Anropas direkt "på klassen" (som klassvariabler)
- Kan anropa andra klassmetoder (inte instansmetoder)
- Kan inte använda this-referensen (this är ju det aktuella objektet, orimligt...)
- Kan överlagras men inte överskuggas (finns inget objekt)

Användning av Klassmetoder

Används då man inte behöver några instansvariabler för att utföra uppgiften

- Motsvarar rena funktioner (som i matematik), indata -> utdata
- T.ex. Math.sin(...)
- Metoden main (den som anropas först av allt i alla Java program) måste vara static …
- … medför att alla metoder som anropas från main måste vara static
- Generellt om static: Mycket litet behov, undvik
- Vi använder bara i main metoder som anropas från main och i våra testklasser ... (more to come)

Programkvalitet

Hur kan vi säkerställa kvaliteten på våra program?

- Vi har märkt hur lätt det är att göra fel … (och svårt att hitta fel)
- Idealiskt vore om vi kunde “bevisa” att programmet är korrekt (bug-fritt)
- Finns några principer vi kan/bör följa?

Programmeringsprinciper

Principer finns ... men sällan absoluta (som i Fysik t.ex).
Mer av riktlinjer, rekommendationer

- Allt som finns/görs skall finnas/göras på ett ställe (annars problem t.ex. vid ändringar). Ingen duplicerat kod
- Allt skall ansvara för en sak (variabler, metoder klasser, …). Lättare att använda, förstå och förändra
- Innebär: Små klasser med små metoder
- Ingen redundant kod. Ju mer kod desto fler möjligheter till fel (extremt kompakt kod svårbegriplig, undvik)
- Använd konstanter
- Minska tillståndet (mer kommer)

Testning

Testning är ett sätt att höja kvaliteten på våra program

- **Enhetstestning** (**unit tests**), testa enskilda objekt i isolering
- **Integrationstestning** (**integration testing**) testa några samverkande objekt (klasser)

Hur utforma tester?

- Tester bör vara automatiska. Ett musklick skall räcka för att köra alla tester (man skall inte behöva sitta och skriva in …)
- En test skall avslutas med “sant” (testen passerade) eller “falskt” (det gick fel, något är fel)
- Alla tester sparas, om något i programmet ändras kan alla testerna köras igen. Allt skall fungera som förut.
- Testerna fungerar som dokumentation, de som arbetar med programmet förstår lättare hur saker är konstruerade

Testning i Kursen

Finns avancerade metoder för testning

- Vi förenklar …
- Testerna kommer att implementeras som metoder i en speciell testklass (namn: Test). Finns färdig mall.
- I testklassen finns en main-metod, som anropar att antal testmetoder (som vi måste skriva)
- Varje testmetod skapar objekt och anropar metoder på detta (dessa)
- Sist i testmetoden anropar vi en (statisk, överlagrad) boolesk funktion som avgör om resultatet var korrekt (om ej undantag)
- Varje test testar en eller ett par metoder på ett objekt (inte allt)
- Kan behövas "hjälp" objekt (om testobjektet är kopplat)
- Ge metoderna beskrivande (långa) namn, så förstår man vad de gör
- Exempel: PigWithTest

Refaktorisera

Att refaktorisera koden innebär att vi ger koden en bättre (begripligare) struktur

- Allt skall fungera som tidigare
- Exempel: Byta namn, dela upp metoder/klasser, slå ihop klasser, ändra kopplingar, ...
- Programmet blir bättre (för en programmerare, användaren märker inget)
- IntelliJ kan göra en hel del av detta
- Genom att vi har tester “vågar” vi göra ändringar av fungerande kod
- … vi kör alla tester efter varje förändring. Alla skall lyckas!

Formell Verifikation

Formell verifikation används för att “bevisa” att program är korrekta*)

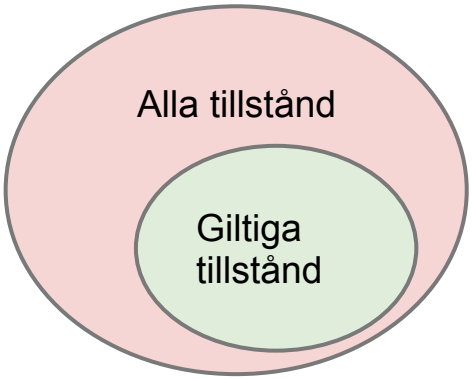
- Mycket komplicerat (logisk bevis)
- Vissa “informella” formella resonemang kan dock hjälpa till
- Vi skall titta på **invarianter** och **förvillkor**
- Vi förenklar och “gör på vårt sätt” (finns andra sätt (åsikter) …)
- Exempel: scrabbleFormalException

*) Vissa program lägger man väldigt stora resurser på att verkligen bevisa korrektheten, t.ex. mjukvara i raketer, flygplan

Tillstånd

Ett objekt har tillstånd

- Tillstånd = Mängden av alla värden för alla instansvariabler
- Vissa tillstånd är inte giltiga, d.v.s då variablerna har dessa värden har något blivit fel
- Ju färre tillstånd ett objekt kan ha desto färre otillåtna tillstånd d.v.s. minimera tillståndet, använd lokala variabler och final



Invariant

En invariant är något som inte förändras

- Invarianter skrivs som booleska uttryck
- Uttrycket skall alltid vara sant (vad som än händer då vi kör programmet)
- Skrivs som kommentar överst i klassen med en Java liknande syntax

```
// Inv: path != null
public class NNN { …
   private ... path = ...
```

Användning av Invarianter

Vi använder invarianter för att ange vilka tillstånd som är giltiga för ett objekt (vilka värden som instansvariablerna inte får ha)

- Vi måste kontrollera att invarianten är sann då objektet precis har skapats och att den är sann vad vi än gör (vilka metoder som än körs)
- … om så är objektet alltid i ett giltigt tillstånd
- Exempel: ScrabbleFormalException

Förvillkor

En del metoder kräver att vissa villkor är uppfylld om de skall exekveras korrekt, ett **förvillkor (precondition)**

- Endast om förvillkoret är uppfyllt “lovar” metoden att utföra sitt uppdrag, om ej är resultatet odefinierat (metoden lovar ingenting)
- Skrivs också som ett boolesk uttryck (i en metodkommentar ovanför metoden)
- Exempel: TileBag, remove-metoden

```
// Pre: tiles.size() > 0
public Tile remove() {
    if (tiles.size() > 0) {
        int i = r.nextInt(tiles.size());
        return tiles.remove(i);
    } else {
        return null;  // Pre cond. failed
    }
}
```

Undantag

Ett typiskt programmeringsproblem är hur ett objekt skall bete sig då det inte kan utföra en uppgift

- Utifrån invarianter och förvillkor kan vi identifiera problematiska situationer
- Kan ev. undvikas …
- … returnera ett värde som är "omöjligt", t. ex. -1 eller null (måste kontrolleras efter anrop). Ibland svårt att hitta omöjligt värde
- … returnera sant eller falskt (boolean) om det gick bra respektive inte.
- … eventuellt generera att undantag

Förvillkor och Undantag

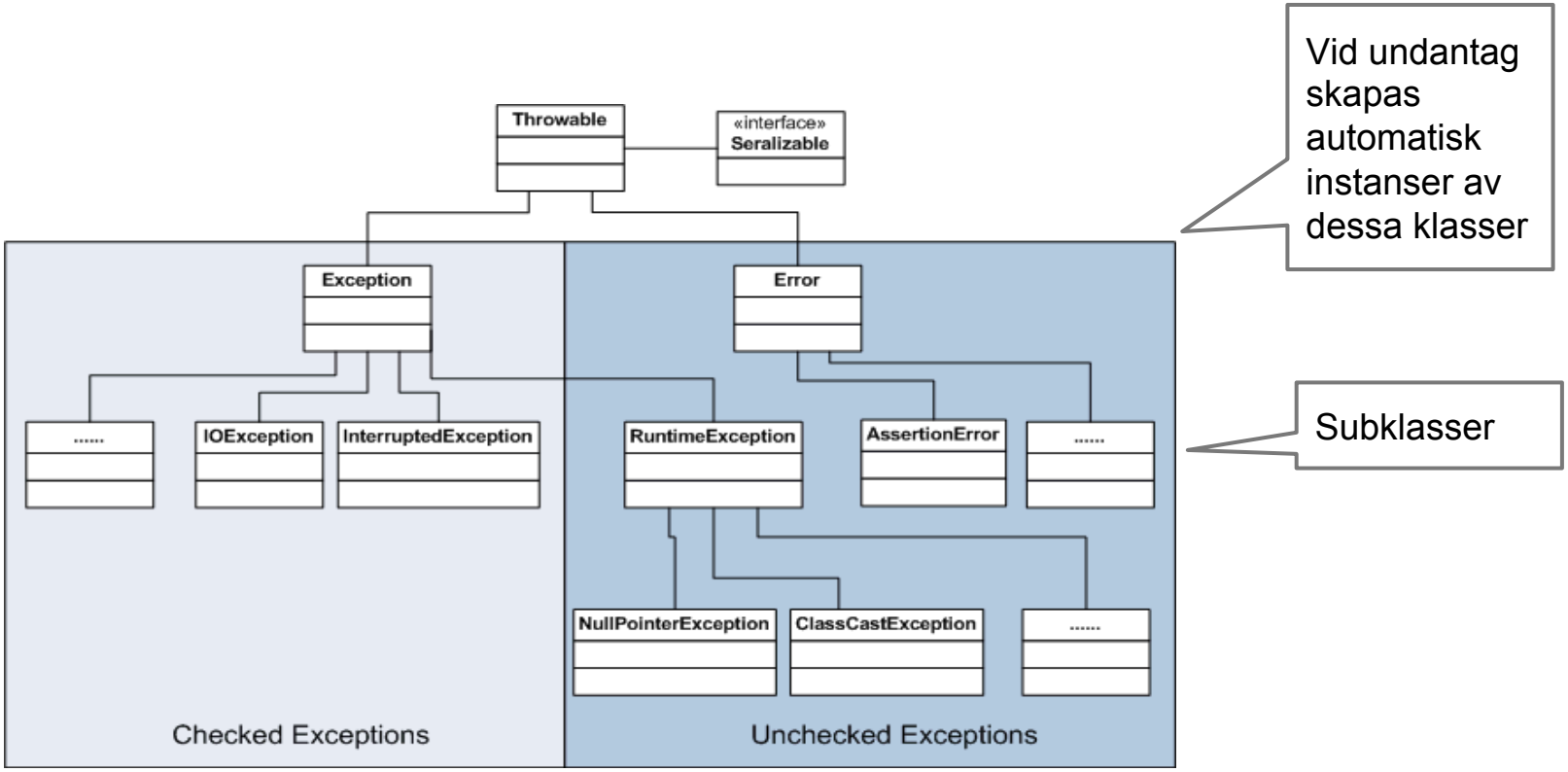
Utifrån förvillkoren lägger man in tester för metodens indata

- Om indata bryter mot något förvillkor kan ett undantag genereras
- Viss indata leder direkt till fel då man anropar något objekt i koden isf behöver man inte själv kasta ett undantag.

```
// Class Holder from Scrabble
// Pre: tile != null
public boolean add(Tile tile) {
    if( tile == null){
        throw new IllegalArgumentException("Tile null");
    }
    for (int i = 0; i < tiles.length; i++) {
        if (tiles[i] == null) {
            tiles[i] = tile;
            actualSize++;
            return true;
        }
    }
```

Rep: Undantag i Java

I Java finns två sorters undantag, unchecked exceptions och checked exceptions



Rep: Undantag i Java, forts

Checked exceptions används för situationer programmet måste hantera (t.ex. “filen saknas”)

- Checked exception måste “fångas” (hanteras), kontrolleras att så sker vid kompilering

Unchecked exceptions används för fel som programmeraren kan göra (NullPointerException)

- Kontrolleras inte vid kompilering
- Unchecked fångas vanligen inte, har vi gjort fel så skall det “smälla” så snabbt och högt som möjligt (programmet skall krascha så att vi märker felet direkt)

Flöde Undantag

Då en anropad metod är klar sker alltid ett återhopp till "den rad" där metoden anropades

- Om vi däremot kastar ett undantag så ändras flödet
- Programmet kommer att "passera" genom alla metodanrop för att slutligen nå main(), där programmet kommer att avbrytas med en felutskrift
- Ett sätt att stoppa detta är att **fånga** undantaget. Kan ske i vilken metod som helst "på vägen"
- Ofta låter man undantaget vandra hela vägen upp till GUI:et där man visar en dialogruta (fångar och visar)

try … catch

Används för att fånga exceptions

- Aldrig lämna catch-grenen tom, då räknas undantaget som fångat men inget händer/syns…