UNIVERSITETET
I OSLO

OQL – Object Query Language

ODMGs spørrespråk

Hva OQL er

- OQL er spørrespråket i ODMG-standarden
- OQL utvider objektorienterte programmeringsspråk med SQL-lignende imperativer og databasefunksjonalitet
- Som SQL, er OQL et deklarativt språk
- OQL er laget for å brukes på data beskrevet med ODL, og OQL brukes alltid sammen med et vertsspråk

- **NB!** I ODL kan, og bør, vi gi ekstensjonenene eget navn
  OQL referer *alltid* til ekstensjoner, og *aldri* til klasser

Motivasjon: «The impedance mismatch»

- Relasjonelle språk som SQL er dårlig egnet for bruk sammen med konvensjonelle programmeringsspråk som C, C++ og Java
- Eksempelvis har SQL og C radikalt forskjellige datamodeller; hverken relasjon, mengde eller bag er innebygd datatype i C
- Mens SQL opererer på hele relasjoner, kan programmerings-språkene bare håndtere ett tuppel om gangen
- Amerikanerne har lånt et utrykk fra fysikken og sier at vi har *an impedance mismatch* mellom SQL og vanlige programmeringsspråk
- OQL er laget for å utvide OO-språkene med SQL-aktige imperativer for å unngå «the impedance mismatch»

Objekt- og verdilikhet

- To (*mutable*) objekter av samme type
  (to forekomster i samme klasse) kan ikke være like,
  men de kan ha samme verdi (*state*)

- Eksempel:
  La O_1 og O_2 være to objekter fra samme klasse
  - OQL-uttrykket $O_1 = O_2$ er alltid **`FALSE`**
  - OQL-uttrykket $*O_1 = *O_2$ er **`TRUE`** hvis, og bare hvis,
    de to objektene har samme verdi i alle attributter

Typer

- Basistyper: **`string, integer, float, boolean, character, enumerations,`** m.fl.
- Typekonstruktører:
  - **`Struct`** for strukturer (vi skal kalle dem struct-er)
  - Samlingstyper (Collection types):
    `set, bag, list, array`
    (Merk: **`dictionary`** kan ikke brukes i OQL)
- Typekonstruktører kan nestes i vilkårlig dybde
- **`Set(Struct())`** og **`Bag(Struct())`** er spesielt viktige i forbindelse med bruk av relasjoner

Beregninger og endringer

- Objekter kan utføre endringer og endre verdi ved å utføre metoder som er definert i objektets klasse

- Select i OQL kan ha sideeffekter, dvs. at den kan endre tilstanden i databasen

  I motsetning til SQL har ikke OQL noen egen update-funksjon

- Metoder kalles ved å navigere langs stier;
  det er ingen forskjell på å adressere attributter, assosiasjoner eller metoder

  Sti-navigering er en arv fra nettverksdatabasene

Sti-uttrykk

- Vi aksesserer komponenter ved hjelp av prikk-notasjon
- La *x* være et objekt av klassen *C*
  - Hvis *a* er et attributt i *C*, så er *x.a* verdien til *a* i objektet *x*
  - Hvis *r* er en assosiasjon i *C*, så er *x.r* verdien *x* er koblet til via *r,* dvs. et objekt eller en samling objekter avhengig av typen til *r*
  - Hvis *m* er en metode i *C*, så er *x.m*(···) resultatet av å anvende *m* på *x*
- Vi kan lage uttrykk med mange prikker, men bare det siste leddet kan være en samling
- OQL tillater piler som synonymer for prikker, så *x*–>*a* er det samme som *x.a* (i motsetning til eksempelvis i C)

ODL for Bar-Beer-Sell (BBS) eksemplet

```
class Bar (extent Bars)
{   attribute string name;
    attribute string addr;
    relationship Set<Sell> beersSold inverse Sell::bar;
}

class Beer (extent Beers)
{   attribute string name;
    attribute string manf;
    relationship Set<Sell> soldBy inverse Sell::beer;
}

class Sell (extent Sells)
{   attribute float price;
    relationship Bar bar inverse Bar::beersSold;
    relationship Beer beer inverse Beer::soldBy;
    void raise_price(float price);
}
```

Stiuttrykk i BBS-eksemplet

- La *s* være en variabel med type **`Sell`**
  - **`s.price`** er prisen i objektet *s* (ølmerke solgt i denne baren)
  - **`s.raise_price(x)`** øker prisen på s.beer i s.bar med x
  - **`s.bar`** er en peker til baren nevnt i *s*
  - **`s.bar.addr`** er adressen til denne baren

    Merk:
    Flere prikker er OK fordi **`s.bar`** er et *objekt*, og ikke en samling
- La *b* være en variabel med type **`Bar`**
  - **`b.name`** er navnet på baren
  - **`b.beersSold`** er en mengde ølmerker som denne baren selger (en mengde pekere til **`Sell`**)

    Merk:
    Stiuttrykket **`b.beersSold.price`** ville ha vært ulovlig fordi **`b.beersSold`** er en *mengde* objekter, og ikke et enkelt objekt

To regler for stiuttrykk

- Hvis x er et objekt, kan stiuttrykket x forlenges slik s forlenges til **`s.beer`** og **`s.beer.name`** ovenfor

- Hvis x er en samling (som **`b.beersSold`** ovenfor), kan x brukes overalt hvor en samling er korrekt (f.eks. etter **`FROM`**)

Select-From-Where (SFW)

- Syntaksen er nær identisk med SQL-syntaks:

  SELECT <liste med verdier>
  FROM <liste med samlinger>
  WHERE <betingelse>

- Samlinger i FROM kan være:
  - ODL extents
  - Uttrykk som evalueres til en samling

  Vi kan gi navn til et «typisk objekt» i en slik samling, et navn vi kan bruke i spørringen

  Objektnavnet plasseres etter samlingen, eventuelt etter nøkkelordet AS
- Merk: Som i SQL kan det være mange spørringer som gir samme svar

Select-From-Where i BBS-eksemplet

- Finn menyen i “Joe’s” med fokus på **`Sells`** objekter:

```
SELECT s.beer.name, s.price
FROM Sells s
WHERE s.bar.name = "Joe's"
```

- Merk at OQL bruker «dobbel-fnutter» på strenger
  (SQL bruker «enkel-fnutter»)

- Finn menyen i “Joe’s”, denne gangen med fokus på **`Bar`** objekter:

```
SELECT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's"
```

- Merk at det typiske objektet *b* i den første samlingen i **`FROM`** brukes i definisjonen av den andre samlingen

Sammenligningsoperatorer

- Verdier kan sammenlignes med disse operatorene:
  - `=` : likhet
  - `!=` : forskjellig fra
  - `<` : mindre enn
  - `>` : større enn
  - `<=` : mindre eller lik
  - `>=` : større eller lik
- Tekster kan i tillegg sammenlignes med disse to operatorene:
  - **`IN`** sjekker om et tegn er i en tekststreng: <c> **`IN`** <text>
  - **`LIKE`** sjekker om to tekster er like: $\langle text_1 \rangle$ **`LIKE`** $\langle text_2 \rangle$
    $\langle text_2 \rangle$ kan inneholde jokertegn:
    - `_` eller `?` : ett vilkårlig tegn
    - `*` eller `%` : en vilkårlig tekststreng

Sammenligningsoperatorer i BBS-eksemplet

- Oppgave: Finn navn og pris på alle ølmerker hos "Joe's" som begynner med "B" og inneholder tekststrengen "ud"

```
SELECT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE   b.name = "Joe's" AND
        s.beer.name LIKE "B*" AND
        s.beer.name LIKE "*ud*" AND
```

Merknad 1:
Barens navn er lik "Joe's"

Merknad 2:
Ølets navn begynner med "B" fulgt av vilkårlig mange vilkårlige tegn

Merknad 3:
Ølets navn inneholder "ud" med et vilkårlig antall vilkårlige tegn foran og bak

Kvantorer

- Vi kan teste om *alle* forkomster, *minst en* forkomst, *noen* forkomster, osv. tilfredsstiller en betingelse
- Dette er boolske uttrykk som brukes i **`WHERE`**-klausuler

  Alle: **`FOR ALL`** *x* **`IN`** <samling> **`:`** <betingelse>

  Minst en: **`EXISTS`** *x* **`IN`** <samling> **`:`** <betingelse>

  `EXISTS` *x*

  Bare en: **`UNIQUE`** *x*

  Noen: <samling> <θ> **`SOME`**/**`ANY`** <betingelse>
  hvor <θ> $\in$ {=, !=, <, >, <=, >=}

  `SOME` og **`ANY`** er to navn på samme kvantor

  `NOT` reverserer den boolske verdien

BBS-eksemplet: Kvantorer - I

Oppgave:
Finn alle barer hvor minst en øltype koster mer enn €5

```
SELECT b.name
FROM Bars b
WHERE EXISTS s IN b.beersSold : s.price > 5.00
```

Oppgave:
Finn alle barer som bare selger øl som koster mer enn €5

```
SELECT b.name
FROM Bars b
WHERE FOR ALL s IN b.beersSold : s.price > 5.00
```

BBS-eksemplet: Kvantorer - II

- Oppgave:
  Finn de barene hvor alt øl som koster mer enn €5 er produsert av “Pete’s”

```
SELECT b.name
FROM Bars b
WHERE FOR ALL be IN
                    be.manf = "Pete's"
( SELECT s.beer
  FROM b.beersSold s
  WHERE s.price > 5.00 ) :
```

Merknad 2:
alle disse «dyre» ølmerkene må være produsert av “Pete’s”

Merknad 1:
finn alle ølmerker i en bar som koster mer enn €5

Resultatets type

- Default: bag av struct-er
  Feltnavnene i struct-en blir siste navn i stiuttrykkene i SELECT-klausulen
- Eksempel: Menyen hos “Joe’s”:

```
SELECT s.beer.name, s.price
FROM Sells s
WHERE s.bar.name = "Joe's"
```

  har resultattypen:

```
Bag(Struct(name: string, price: real))
```

Renavning av felt

- Det er ikke alltid resultattypen

```
Bag(Struct(name: string, price: real))
```

  har de «riktige» attributtnavnene
- Renavn ved å prefikse stien med ønsket navn og et kolon
- Eksempel: Menyen hos “Joe’s”:

```
SELECT beername: s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's"
```

  har typen:

```
Bag(Struct(beername: string, price: real))
```

Endring av samlingstype til mengde

- Det er ikke alltid ønskelig å få en *bag* med struct-er (default) som resultat av en SFW-spørring
- Bruk **`SELECT DISTINCT`** for å få en *mengde* med struct-er
- Eksempel:

```
SELECT DISTINCT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's"
```

Endring av samlingstype til liste

- Bruk **`ORDER BY`** for å få en *liste* med struct-er
- Eksempel:

```
joeMenu = SELECT s.beer.name, s.price
          FROM Bars b, b.beersSold s
          WHERE b.name = "Joe's"
          ORDER BY s.price ASC
```

- **`ASC`** = ascending (stigende) er default
  `DESC` = descending (synkende)
- Vi kan bruke en liste som om den var en array:

```
cheapest_beer = joeMenu[0].name;
```

Subqueries

- Brukes når resultatet er en samling, det vil hovedsklig si
  - i **`FROM`**-klausuler
  - sammen med kvantorer som **`EXISTS`** og **`FOR ALL`**

- Eksempel: Subquery i **`FROM`**:
  Finn produsentene av øl som serveres hos “Joe's”

```
SELECT DISTINCT b.manf
FROM (  SELECT s.beer
        FROM Sells s
        WHERE s.bar.name = "Joe's"
    ) b
```

Bruk av vertsspråkvariable

- Når vi bruker SQL fra et vertsspråk, må vi flytte data mellom tupler og variable

  Det trenger vi ikke når vi bruker OQL
- Alle variable med riktig type kan settes lik et OQL-uttrykk
- Eksempel (I C++ stil):
  Navn på barer som bare selger øl som koster mer enn €5

```
Set<string> expensive_bars;
expensive_bars =
   SELECT DISTINCT b.name
   FROM Bars b
   WHERE FOR ALL s IN b.beersSold :
                          s.price > 5.00
```

Å hente forekomster fra en samling – I

- Gitt en samling med bare en forekomst
  Bruk **`ELEMENT`** til å hente forekomsten

- Eksempel:
  Finn prisen "Joe's" tar for "Bud" og legg resultatet i variabelen *p*:

```
p = ELEMENT(
      SELECT s.price
      FROM Sells s
      WHERE s.bar.name = "Joe's" AND
            s.beer.name = "Bud")
```

Å hente forekomster fra en samling– II

- Hvordan hente alle forekomstene i en samling, en om gangen:
  - Gjør samlingen om til en liste
  - Hent listeelementene med <list_name>[i]
- Eksempel (i C-stil):
  Skriv ut Joe's meny, sortert etter pris, med øltyper med samme pris ordnet alfabetisk

```
L = SELECT s.beer.name, s.price
    FROM Sells s
    WHERE s.bar.name = "Joe's"
    ORDER BY s.price, s.beer.name;

printf("Beer\tPrice\n\n");

for(i=0; i<=COUNT(L); i++)
    printf("%s\t%f\n", L[i].name, L[i].price);
```

Merknad 1:
Lag en liste

Merknad 2: Element nr `i` i `L` hentes fra `L[i-1]`
Indeksen `i` starter i `0`

Å lage objekter

- En SFW-spørring lar oss lage nye objekter hvor typen er definert av **`SELECT`**-klausulen
- Eksempel:

```
SELECT beername: s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's Bar"
```

Merknad: Definerer et nytt objekt med type `Bag<Struct( beername: string, price: integer)>`

- Constructor-metoder lar oss lage nye persistente objekter (detaljene avhenger av vertsspråket)
- Eksempel: Legg inn en ny øltype

```
newBeer = Beer(name: "XXX", manf: "YYY")
```

Lager et nytt **`Beer`**-objekt, som legges i extent-en **`Beers`**

Vertsspråkvariabelen **`newBeer`** settes lik dette objektet

Aggregatfunksjoner

- De fem funksjonene **`AVG`**, **`SUM`**, **`MIN`**, **`MAX`**, og **`COUNT`** kan brukes på enhver samling så sant funksjonen har mening for elementtypen i samlingen
- Eksempel:
  Finn gjennomsnittsprisen på øl hos Joe's

```
x = AVG( SELECT s.price
         FROM Sells s
         WHERE s.bar.name = "Joe's");
```

- Merk:
  Rent teknisk er resultatet av denne SFW-spørringen en bag med ett-felts struct-er som tolkes som en bag med verdiene i det feltet

Union, snitt og differans

- Vi kan ta union, snitt og differans mellom alle objekter av type **`Set`** eller **`Bag`** ved henholdsvis å bruke nøkkelordene **`UNION`**, **`INTERSECT`** og **`EXCEPT`**
- Resultatet er en **`Bag`** hvis minst ett av objektene er en **`Bag`**
  Hvis begge objektene er et **`Set`**, blir resultatet et **`Set`**
- Eksempel:
  Finn navn på alle ølmerker "Joe's" selger som "Steve's" ikke selger

```
( SELECT s.beer.name
  FROM Sells s
  WHERE s.bar.name = "Joe's" )

 EXCEPT

( SELECT s.beer.name
  FROM Sells s
  WHERE s.bar.name = "Steve's" )
```

Merknad 1:
Finn alle ølmerker "Joe's" selger

Merknad 2:
Finn alle ølmerker "Steve's" selger

Merknad 3:
Fjern alle ølmerker "Steve's" selger fra de "Joe's" selger

Gruppering – I

- Gruppering i OQL ligner den i SQL, men det er noen forskjeller
- SQL-aktig eksempel: Finn gjennomsnittsprisen på øl i alle barer

```
SELECT bar.name, AVG(price)
FROM Sells
GROUP BY bar;
```

- Et betimelig spørsmål: Hva er **`bar`** i eksemplet ovenfor, er det *navnet* på gruppen eller den felles *verdien* av **`bar`** i alle tupler i gruppen?

  I SQL spiller det ingen rolle, men i OQL kan du lage grupper fra verdien av vilkårlige funksjoner (metoder), og ikke bare av attributter
- I OQL bestemmes gruppene av felles verdier, og ikke av navn
- Eksempel:
  Vi kan gruppere på første bokstav i barnavnet
  (Da trenger vi en funksjon (metode) som gir oss denne bokstaven)

Gruppering – II

- Generelt format:

  GROUP BY f_1: e_1, f_2: e_2, ..., f_n: e_n

- Syntaks for grupperingsklausulen i OQL består av
  - nøkkelordene **GROUP BY**
  - en kommaseparert liste av navngitte grupperingsverdier
- Hvert listeelement består av
  - navn
  - kolon
  - grupperingsverdi
- Eksempel:

```
SELECT ...
FROM ...
GROUP BY barName: s.bar.name
```

Oversikt over utføring av gruppering

Startsamlingen
Definert av **FROM**, **WHERE**

Grupperingsverdier
definert av funksjoner

Mellomresultatet: Grupper av funksjonsverdier og partisjon

Termene listet i
SELECT-klausulen

Sluttresultatsamlingen

Merknad 1:
De utvagte objektene (**WHERE**) fra samlingen av objekter i **FROM**
Rent teknisk er det en **Bag** av struct-er

Merknad 2:
Verdiene som returneres fra *startsamlingen* som resultat av **GROUP BY** uttrykk:
Struct(f$_1$:v$_1$, ..., partition:P)
De første feltene definerer gruppen, **P** er en bag av verdier som tilhører denne gruppen

Merknad 3:
SELECT-klausulen kan velge verdier fra *mellomresultatet*, dvs. fra **f$_1$, f$_2$, .., f$_n$** og **partition**
Verdier i den siste kan bare aksesseres via aggregatfunksjoner på medlemmene i bagen **P**

BBS-eksemplet: Gruppering – I

- Grupperingsoppgave 1:
  Finn gjennomsnittsprisen på øl for hver bar

```
SELECT    barName, avgPrice:
          AVG(SELECT p.s.price
              FROM   partition p)

FROM      Sells s

GROUP BY  barName: s.bar.name
```

BBS-eksemplet: Gruppering – II

```
SELECT    barName, avgPrice:
          AVG(SELECT p.s.price
              FROM   partition p)
FROM      Sells s
GROUP BY  barName: s.bar.name
```

1. *Startsamlingen*: **`Sells`**
   - Teknisk er dette en bag av struct-er på formen **`Struct`**(**`s:`***s*1) hvor *s*1 er et **`Sell`** -object
   - **`s`** kalles «et typisk objekt» og fungerer som navn på det eneste feltet i denne struct-en
   - Generelt bør vi (men må ikke) gi navn til typiske objekter for alle samlinger i **`FROM`**-klausulen

BBS-eksemplet: Gruppering – III

```
SELECT   barName, avgPrice:
         AVG(SELECT p.s.price
             FROM   partition p)
FROM     Sells s
GROUP BY barName: s.bar.name
```

2. Mellomresultatet

- Vi har én funksjon, **`s.bar.name`**, som gitt et **`Sell`**-objekt *s* gir oss navnet på baren som *s* refererer til
- Mellomresultatet blir en mengde struct-er av type:
  `Struct{barName:string,`
  `partition:Bag<Struct{s:Sell}>}`
- For eksempel: **`Struct(barName =`**
  `"Joe's",partition = {`s_1,…,s_n**`})`**
  hvor s_1,…,s_n er alle struct-er med ett felt, kalt **`s`**, hvis verdi er ett av **`Sell`**-objektene som sier at et ølmerke selges i Joe's Bar

BBS-eksemplet: Gruppering – IV

```
SELECT   barName, avgPrice:
         AVG(SELECT p.s.price
             FROM   partition p)
FROM     Sells s
GROUP BY barName: s.bar.name
```

3. Sluttresultatsamlingen

Denne består av barnavn-gjennomsnittspris-par, ett par for hver struct i mellomresultatsamlingen

- Typen er: **`Struct{barName: string, avgPrice: real}`**
- Merk at i subqueryet i **`SELECT`**-klausulen aksesseres variablene i partisjonen via aggregatfunksjonen **`AVG`**
- *p* varierer over alle struct-ene i **`partition`**, og disse struct-ene består av et enkelt felt kalt **`s`** med et **`Sell`**-objekt som verdi
- Følgelig henter **`p.s.price`** frem prisen fra et av **`Sell`**-objektene som tilhører akkurat denne baren
- Eksempel på typisk output:
  `Struct(barName = "Joe's", avgPrice = 2.83)`

BBS-eksemplet: Gruppering – V

- Grupperingsoppgave 2:
  For hvert ølmerke, finn antall barer som tar en lav pris (≤ 2.00) og en høy pris (≥ 4.00) for det ølmerket
- Strategi: Grupper etter tre kriterier:
  - ølmerket
  - en boolsk funksjon som er sann hvis prisen er lav
  - en boolsk funksjon som er sann hvis prisen er høy

```
SELECT   beerName, low, high,
         count: COUNT(partition)
FROM     Beers b, b.soldBy s
GROUP BY beerName: b.name,
         low: s.price <= 2.00,
         high: s.price >= 4.00
```

BBS-eksemplet: Gruppering – VI

```
SELECT    bName, low, high, count: COUNT(partition)
FROM      Beers b, b.soldBy s
GROUP BY  bName: b.name,
          low: s.price <= 2.00,
          high: s.price >= 4.00
```

1. *Startsamlingen*: Par (*b*, *s*), hvor *b* er et **`Beer`**-objekt, og *s* et **`Sell`** (**`b.soldBy`**) objekt som representerer salg av dette ølmerket i en bar
   - Medlemmene i startsamlingen har type: **`Struct{b: Beer, s: Sell}`**

BBS-eksemplet: Gruppering – VII

```
SELECT    bName, low, high, count: COUNT(partition)
FROM      Beers b, b.soldBy s
GROUP BY  bName: b.name,
          low: s.price <= 2.00,
          high: s.price >= 4.00
```

2. *Mellomresultatet*:
   Kvadrupler som består av ett ølmerke, to boolske verdier som sier om gruppen gjelder høye og/eller lave priser og gruppens partisjon

   Partisjonen er en mengde struct-er med type:

   `Struct{b: Beer, s: Sell}`

   Eksempel på en typisk forekomst i partisjonen:

   `Struct(b:"Bud"` object**`,s:`**et **`Sell`**-object som gjelder Bud**`)`**

BBS-eksemplet: Gruppering – VIII

2. *Mellomresultatet* (forts.):
   - Kvadruplene i mellomresultatet er av type:

```
Struct{bName: string,
       low:   boolean,
       high:  boolean,
       partition: Set<Struct{b: Beer, s:Sell}>}
```

   - Typiske struct-er i mellomresultatet:

bName	low	high	partition
Bud	TRUE	FALSE	S_{low}
Bud	FALSE	TRUE	S_{high}
Bud	FALSE	FALSE	S_{mid}
…	…	…	…

Merknad 1:
S_X er mengder av beer-sell-par (b, s)

Merknad 2:
S_{low} : prisen er lav (≤ 2)

Merknad 3:
S_{high} : prisen er høy (≥ 4)

Merknad 4:
S_{mid} : medium pris (mellom 2 og 4)

Merknad 5:
Partisjoner med `low = high = TRUE` er tomme og blir ikke med i mellomresultatet

BBS-eksemplet: Gruppering – IX

```
SELECT   bName, low, high, count: COUNT(partition)
FROM     Beers b, b.soldBy s
GROUP BY bName: b.name,
         low: s.price <= 2.00,
         high: s.price >= 4.00
```

3. *Sluttresultatsamlingen*:
   - De tre første komponentene i hver gruppes struct kopieres til sluttresultatet
   - Den siste komponenten (**`partition`**) telles opp
   - Et eksempel på et resultat for ett ølmerke:

bName	low	high	count
Bud	TRUE	FALSE	27
Bud	FALSE	TRUE	14
Bud	FALSE	FALSE	36

Having

- **`GROUP BY`** kan bli fulgt av en **`HAVING`**-klausul for å eliminere noen av gruppene generert av **`GROUP BY`**
- **`HAVING`**-betingelsen testes på **`partition`**-feltet i hver struct (gruppe) i mellomresultatet
- Hvis **`HAVING`**-betingelsen er **`FALSE`**, vil gruppen ikke bidra til resultatsamlingen

BBS-eksemplet: Having

- Oppgave:
  Finn gjennomsnittsprisen på øl i hver bar, men bare i barer hvor det dyreste ølet koster mer enn €10

```
SELECT    barName, avgPrice:
          AVG(SELECT p.s.price
              FROM   partition p)
FROM      Sells s
GROUP BY barName: s.bar.name

HAVING MAX( SELECT p.s.price
            FROM partition p) > 10
```

Merknad 1:
Samme som før:
Finn gjennomsnittspris på øl i en bar

Merknad 2:
Velg bare de gruppene hvor maksimumsprisen er større enn 10

Oppsummering av OQL

- Queries/subqueries – **`Select-From-Where`** (SFW)
- Resultattyper – bager, mengder eller lister
- Kvantorer – **`for all`**, **`exists`**, m.fl.
- Generering av objekter –
  både ved bruk av new og som svar på queries
- Aggregering – **`count`**, **`max`**, **`min`**, **`avg`**, **`sum`**
- Bruk av vertsspråk – OQL passer naturlig inn
- Operatorer på mengde- eller bagobjekter –
  `union`, **`intersect`**, **`except`**
- Gruppering med egenskaper – **`group by`** with **`having`**