

Hva er rekursjon?

- Enkelt forklart: en metode som kaller på seg selv
- Mer definert: en teknikk som løser et problem ved å løse mindre problemer av samme type
- Kan ses på som en evig løkke

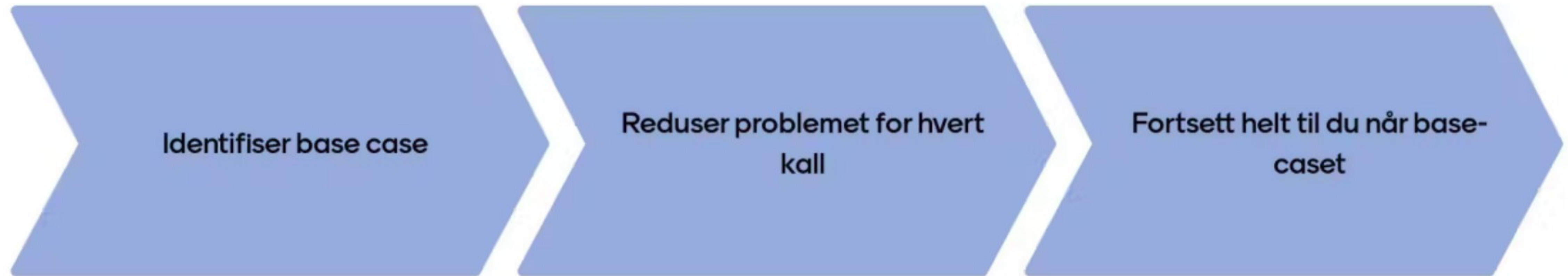
Hva må vi ha med i en rekursiv metode?

- Vi må ha et base tilfelle som stopper rekursjonen
- Noe som forandrer seg i kallene. - slik at vi til slutt når basistilfelle
- Rekursivt kall

```
void rekursjon(int x){  
    if(x == 0) return;  
    rekursjon(x-1);  
}
```

Base tilfelle

Rekursivt kall der noe forandrer seg i kallene



Base case må ikke være en if-setning

→ For eksempel i oblig 7 (om labyrint) så løses det med polymorfi

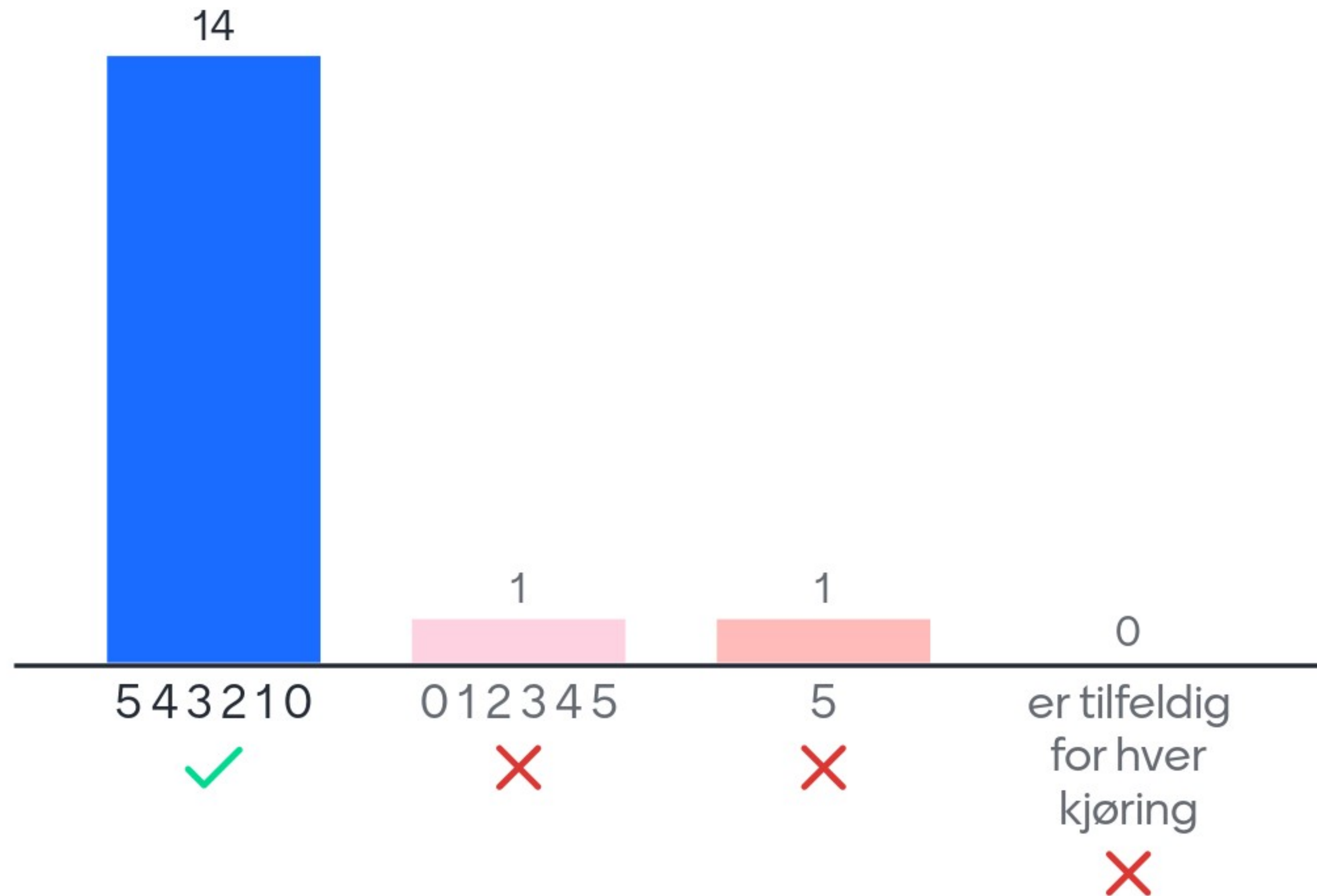
```
class HvitRute extends Rute {  
    @Override  
    public void finn(Rute fra) {  
        // Gå videre til alle naboer  
    }  
}
```

```
class Aapning extends HvitRute {  
    @Override  
    public void finn(Rute fra) {  
        System.out.print("Fant en åpning: (" + rad + ", " + kol + ")");  
    }  
}
```

Hvordan løse rekursive problemer?

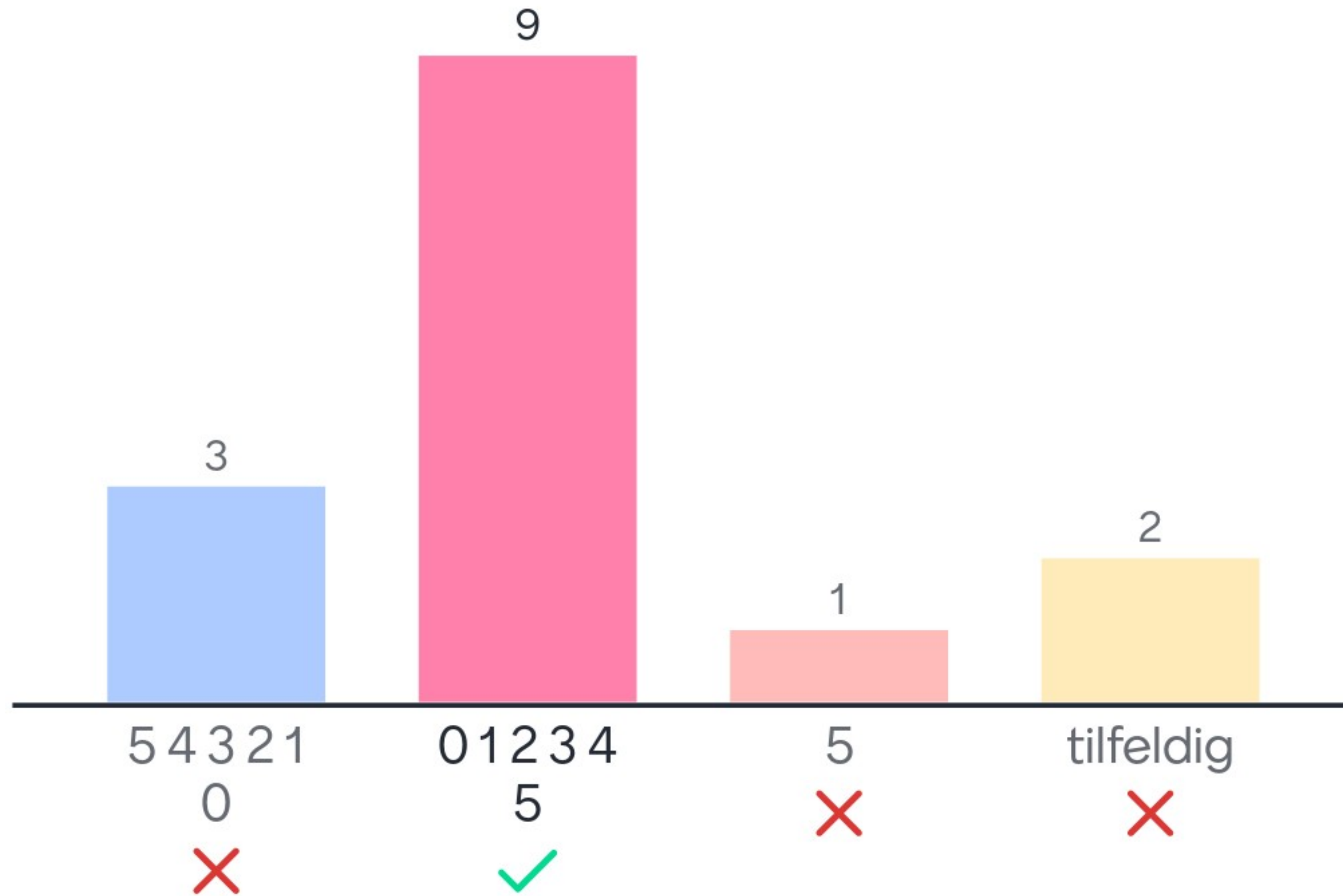
- Hva er det enkleste mulige tilfellet/inputet? Hva skal resultatet være da? Tilsvarende base case
- Se på gradvis større eksempler for å finne ut hva det rekursive steget skal være.
- Hvordan kan man redusere problemet for hvert kall for å nærme seg basistilfellet?

Hva printes ut?



```
class Rekursjon{  
  
    static void skrivTall1(int n){  
        if(n < 0) return;  
        System.out.println(n);  
        skrivTall1(n-1);  
    }  
    Run | Debug  
    public static void main(String[] args) {  
        skrivTall1(n:5);  
    }  
}
```

Hva printes ut?



```
class Rekursjon{  
  
    static void skrivTall2(int n){  
        if(n < 0) return;  
        skrivTall2(n-1);  
        System.out.println(n);  
    }  
    Run | Debug  
    public static void main(String[] args) {  
        skrivTall2(n:5);  
    }  
}
```

Live koding

- Rekursjon i Lenkelister
- Oppgave 2d eksamen 2019 om hunder
- Flere små eksempler

Tips

- Tegne opp rekursjonen
- Se på youtube
- Gjør tidligere ukeoppgaver (uke 14) eller trix

REPETISJONSKURS IN1010 - GUI

Sivert Fjeldstad Madsen

I DAG

- GUI
- Standard oppsett
- MVC
- Livekode





graphic design is my passion

GUI – GRAPHICAL USER INTERFACE

- De fleste programmene vi har skrevet i IN1010 blir brukt gjennom terminalen
- Programmer som er tenkt å bli brukt mye har sjelden terminalen som eneste interaksjonspunkt
- Man programmerer gjerne et eget *interaksjonslag* som lar brukeren påvirke programmet
 - Det er dette vi kaller et brukergrensesnitt
- Her vil en bruker kunne få se og interagere med programmet på ulike måter



GUI - KODESKIKK

- En lur ting å tenke på når man programmerer større programmer som skal ha et GUI, er å holde det adskilt fra resten av koden 
- Tenk at du skal kunne enkelt bytte mellom flere forskjellige GULer ved bare å endre hvilken klasse som blir brukt. F.eks:
 - En klasse som gir et grensesnitt basert på **terminalen** (som i oblig 4)
 - En klasse som gir et grensesnitt basert på Swing og AWT
- På denne måten er det enkelt å gjøre endringer på grensesnittet uten å måtte endre masse underliggende logikk, og vice versa
- Dette er en annen form for innkapsling 

DEKLARATIV PROGRAMMERING

- De fleste moderne rammeverk for GUIer bruker det som kalles **deklarativ programmering**
- I IN1010 har vi i all hovedsak drevet med **imperativ** programmering, med noen få smakebiter av **funksjonell** programmering
- Deklarativ programmering går ut på at programmereren (deg) forteller *hva* de vil ha, men ikke i like stor grad *hvordan*
- Dette gjør at vi slipper å forholde oss til mange ting, sånn som hvordan man oppretter tråder for de ulike GUI-delene og hvordan man faktisk tenger et vindu på en skjerm
- Det gjør også at vi må godta å ikke ha like god kontroll over alt som vi har blitt vant til

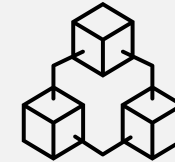


GENERELT OPPSETT

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;
```

Importerer bibliotekene vi
trenger

```
...  
try {  
    UIManager.setLookAndFeel(  
        UIManager.getCrossPlatformLookAndFeelClassName()  
    );  
} catch (Exception e) {  
    System.exit(1);  
}
```



Setter utseendet til å
matche operativsystemet
(hvis mulig)

```
JFrame vindu = new JFrame("Dette er navnet på vinduet!");  
vindu.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
...  
vindu.pack();  
vindu.setLocationRelativeTo(null);  
vindu.setVisible(true);
```

Oppretter vinduet

Gjør så programmet
stopper når vinduet lukkes

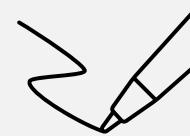
Legger alt innhold i vinduet

Setter vinduet midt på
skjermen

Gjør vinduet synlig

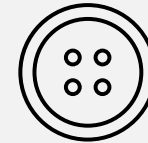
LEGG TIL ELEMENTER I VINDUET

- Et helt tomt vindu er ikke spesielt interessant
- Vi bruker innebygde klasser for å representere ulike ting:
 - **JPanel** – for å lage tegneflater
 - **JLabel** – for tekst
 - **TextField** og **TextPane** – for tekstfelter
 - **Button** – for knapper
- Elementer legges til en tegneflate med **.add()**
- Tegneflater legges til vinduet (JFrame) med **.add()**



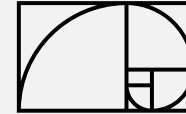
JBUTTON

- Knapper er en av de vanligste måtene for en bruker å interagere med et program
- Vi må definere hva som skal skje når en knapp blir trykket på, og dette gjør vi med indre klasser
 - Ganske likt som arbeiderklassene når vi programmerer med tråder
- «Hendelses»-klassen vår må implementere interfacet **ActionListener**
 - Som krever at vi skriver metoden **public void actionPerformed (ActionEvent e)**
 - Denne metoden definerer hva som skjer når knappen blir trykket på



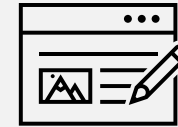
INTERFACE

- Det kan være en god idé å lage et **interface** for view-klassen
 - La alle views tilknyttet programmet implementere dette
- På denne måten blir det enklere å erstatte viewet med et annet dersom man skulle ønske det
- I kontrolleren (og eventuelt modellen) kan man da alltid vite at metodene man bruker finnes uansett hvilket view man bruker
- Dette gjør det enklere for andre å bygge videre på programmet ditt
- Det vil også kunne gjøre testing enklere



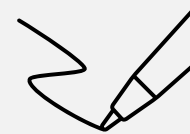
LAYOUT

- Når vi lager GUI vil vi ofte presentere de ulike elementene på bestemte måter
- Vi kan da bruke ulike former for **layout**
 - Vi kan bestemme dette for alle tegneflater (JPanel)
 - Merk også at vi kan ha tegneflater inne i tegneflater
- **BorderLayout** lar oss velge noen forhåndsdefinerte posisjoner
- **GridLayout** lar oss lage et rutenett
- Slide **39** og **40** fra [forelesningen](#) gir gode eksempler på dette



FONTER OG FARGER

- Man kan endre fonten og stilen på teksten i alle elementer som inneholder tekst
 - Dette gjøres med `element.setFont()`
- I tillegg kan man endre størrelse, farge, rammer, og bakgrunner på de fleste elementer
- Igjen gir [forelesnings-pdfen](#) en god gjennomgang av dette, på slide **41-44**



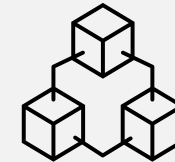
MODEL-VIEW-CONTROLLER

- Når vi skriver større programmer er det lurt å strukturere dem på en fornuftig måte
- Vi trenger ikke finne opp hjulet på nytt hver gang
- Derfor benytter vi oss gjerne av et allerede eksisterende **programmeringsmønster**
- I IN1010 lærer vi om **MVC**
 - I IN2000 lærer man **MVVM**: Model-View-ViewModel
 - Det finnes også andre

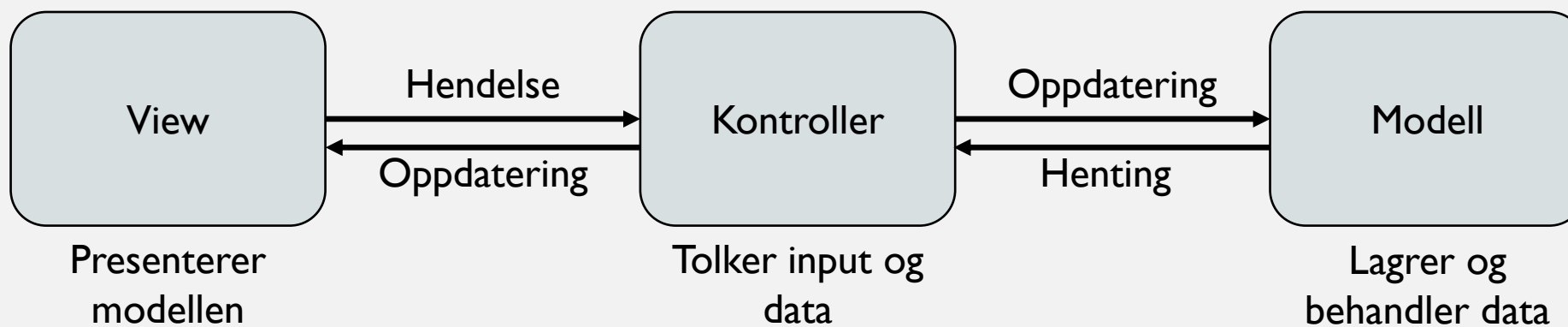


MVC - FORTSETTELSE

- Tanken bak MVC er å dele opp programmet vårt i tre ulike deler som hver har ansvaret for forskjellige ting
- **Modellen** er der all dataen og mesteparten av logikken bak programmet ligger
 - Dette kan f.eks. være en eller flere lister med data
- **Viewet** er det som presenterer dataen fra modellen til brukeren
 - Typisk **GUI**en til programmet
- **Kontrolleren** er det som ligger mellom og kobler til to sammen
 - All kommunikasjon mellom GUIen og modellen bør skje gjennom kontrolleren



PROGRAMFLYT I MVC



OPPSUMMERING

- GUI i Java gir flere muligheter for brukerinteraksjon!
- Lag et ark med de vanligste konstruksjonene i Swing og AWT, siden disse brukes hele tiden
- Ikke tenk på grafisk design på eksamen!
 - Bare pass på å ha med alle delene dere blir bedt om
- MVC handler om en tredeling av ansvar
 - Viewet og modellen kjenner kun til kontrolleren, og kontrolleren kjenner til de to andre



IN1010 Repetisjon

Tråder

Julian Fjeld

julianfj@uio.no

Oversikt

- Tråder
 sleep()
 join()
- Runnable
- Monitor
- Lock
- Condition
- Barrierer
 CountDownLatch

Hvorfor bruker vi tråder?

- De fleste datamaskiner i dag har flere prosessorer eller kjerner
- Kjerner kan jobbe på forskjellige ting samtidig
- Når vi lager programmer med tråder sier vi til datamaskinen at disse arbeidsoppgavene kan gjøres i hvilken rekkefølge som passer den, også samtidig
- Visse arbeidsoppgaver kan med andre ord gjøres mer effektivt

Tråder

- Tenk på tråder som arbeidere
- Effektive, dumme, halvblinde arbeidere. De skal ikke vite hva som skjer rundt seg, men la seg lede av arbeidsleder (monitor)
- En tråd kan få en oppgave å gjøre, en klasse vi skriver som implementerer grensesnittet Runnable
- Mange tråder kan gjøre oppgavene sine samtidig, men jobber de på samme ting må de vente til den er ledig
- Når vi bruker venting med tråder må vi huske å ta høyde for at de kan bli avbrutt (InterruptedException)
- Sleep(x) gjør at en tråd venter i x antall millusekunder
- Om traad er en referanse til en tråd kan vi bruke traad.join() for å vente til traad er ferdig med run-metoden sin

- Når vi oppretter en tråd trenger vi (som oftest) i IN1010 en oppgave (ny instans av en klasse som implementerer Runnable), og en referanse til en monitor

Monitor

- En monitor kan vi tenke på som en arbeidsleder
- Monitoren tar seg av all organisering av tråder og passer på at de ikke snubler i beina på hverandre
- Om to tråder vil aksessere samme data er det monitor som passer på at de havner i køer (lock() og unlock())
- En monitor har som regel en beholder, en lås, og eventuelle conditions til den låsen
- Den implementerer også metoder som gjør at tråder kan interagere med beholderen, men passer på at bare én tråd kan gjøre det av gangen (lock() og unlock())

Lock

- Grensesnitt med 6 metoder, vi bruker bare 2, nemlig `lock()` og `unlock()`
- Vi bruker klassen `ReentrantLock` som implementerer `Lock`
- Når en tråd prøver å låse en lås som er låst, havner den i en kø og venter på at låsen låses opp før den fortsetter
- Denne køen er ikke synlig for oss som bruker låsen, men skjer i bakgrunnen og kan virke litt som magi når man ikke vet hvordan det fungerer

Condition

- Condition kan vi bruke om vi vil at tråder skal vente på en lås ved andre tilfeller enn at den bare er låst
- Disse tilfellene definerer vi selv, og må vi også passe på å opprettholde selv
- Om vi for eksempel venter på at en liste har mer enn 1 element i seg må vi passe på å gi signal til alle som venter ved hvert tilfelle det kan ha blitt satt inn et element
- En Condition opprettes via en lås med `laas.newCondition()` og hører til den låsen

Barrierer

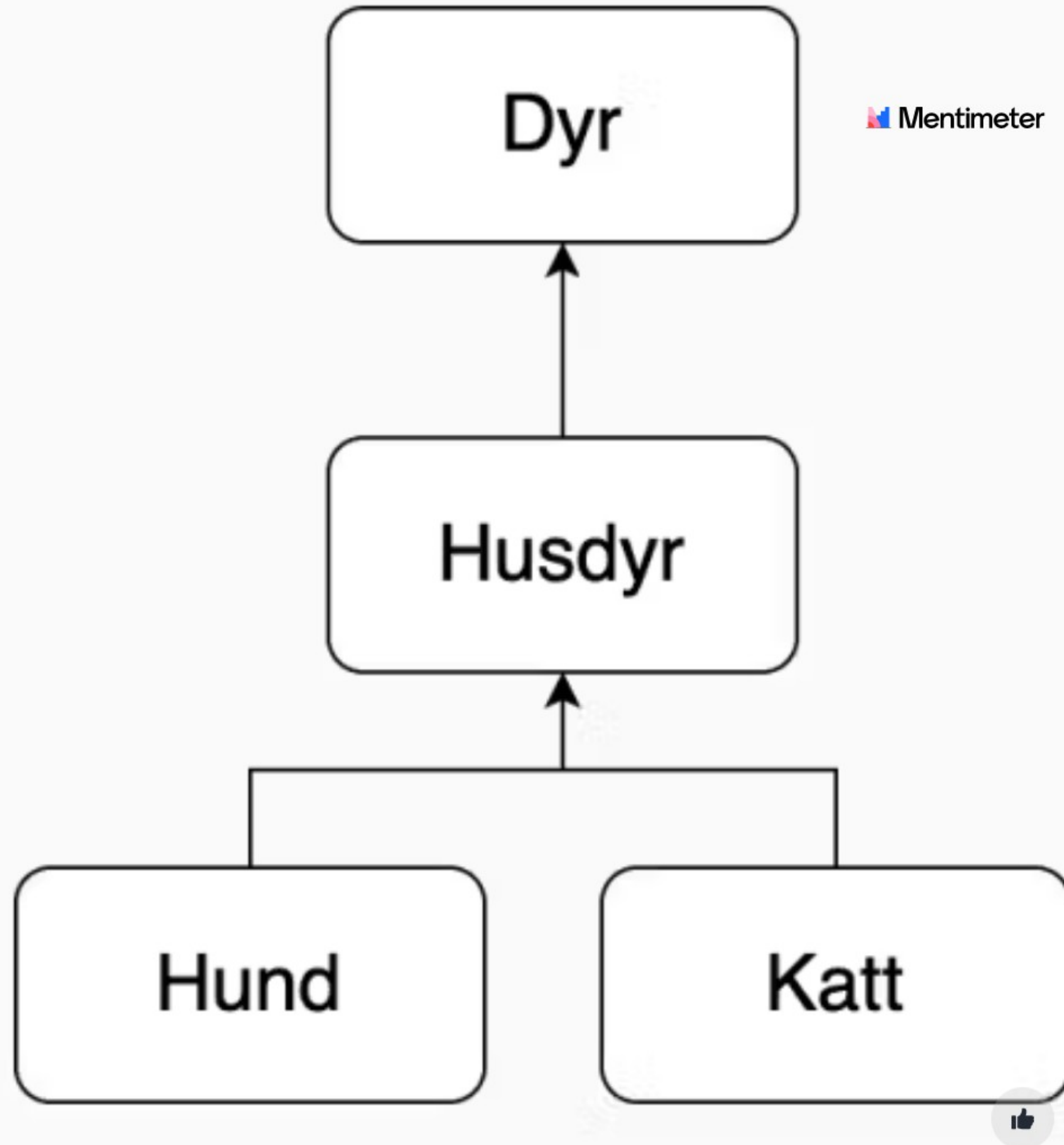
- Barrierer er deler av koden der vi synkroniserer tråder
- Det betyr at én tråd, for eksempel tråden som kjører main(), venter på andre tråder
- Ofte kan man bruke join til dette (om man har en referanse til trådene)
- CountdownLatch er også en mulig løsning

Arv

- Klasser i objektorientert programmering representerer noe som deler et sett med egenskaper
- En subklasse har egenskapene til en klasse + noen nye mer spesialiserte egenskaper
- Subklassen arver egenskapene til superklassen
- Hvorfor subklasser?
 - Ønsker å modellere virkeligheten
 - Gir struktur til systemet vi lager
 - Gjenbruk

Klassehierarki

- Kan ha flere nivåer med arv
- Alle hunder er både dyr og husdyr
- Alle katter er både dyr og husdyr
 - Ingen hunder er katter
 - Ingen katter er hunder




```
class Hund extends Husdyr{...}
```

- extends: for å lage en subklasse

```
protected int alder;
```

- protected: alle av klassens subklasser kan se egenskapen

```
abstract class Dyr {...}
```

- abstract: hvis en klasse er abstrakt kan man ikke opprette en instans av denne klassen, men subklassene arver egenskaper

```
public abstract void spis();
```

- i en abstrakt klasse så kan man også ha abstrakte metoder som man ikke trenger å implementere
- alle subklassene må da implementere metoden

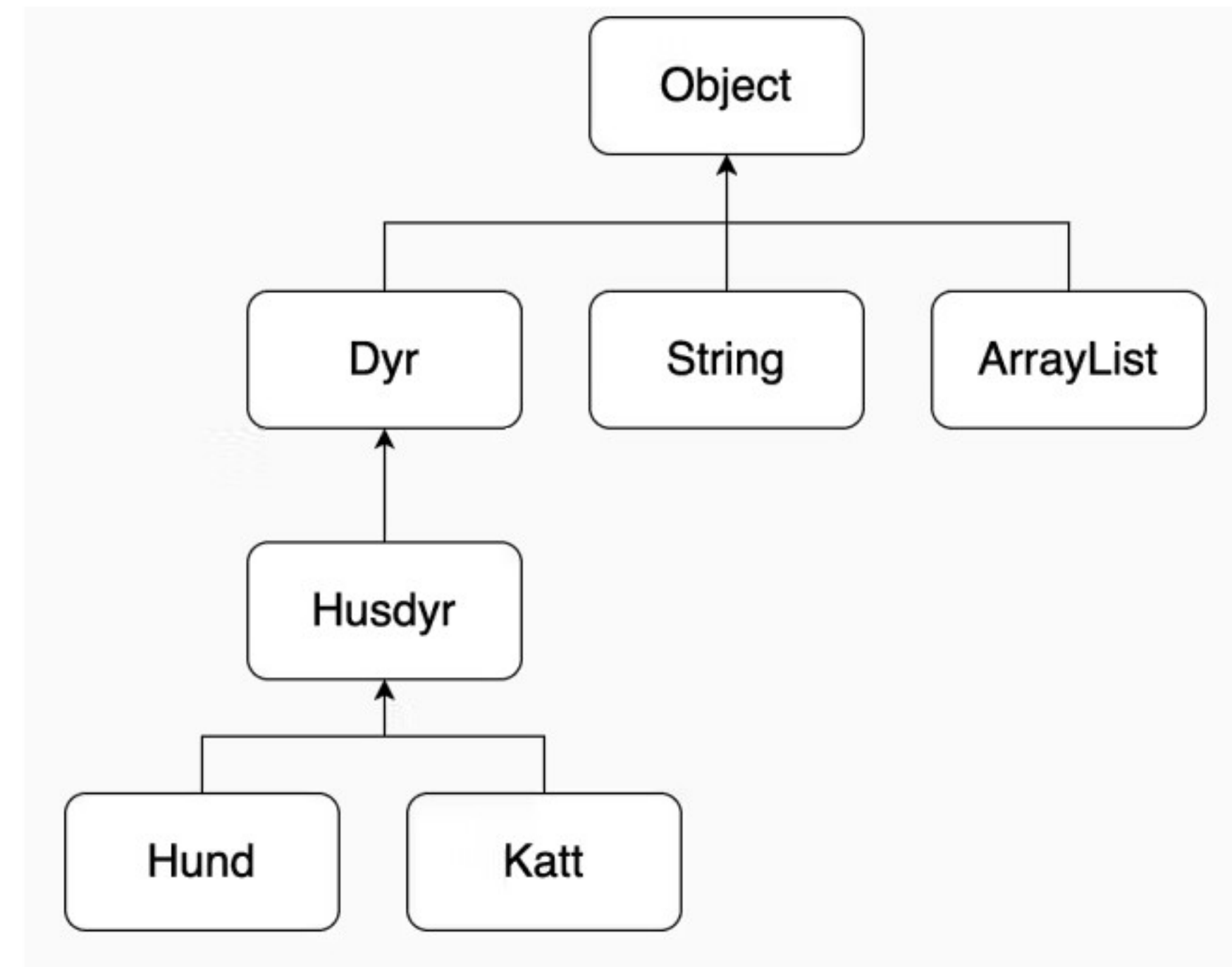
Nøkkelord

Super i konstruktør

- `super(variabel);` i konstruktøren til en subklasse brukes super for å sende verdier opp til superklassens konstruktør
- Et kall på super må legges først i konstruktøren
- Hvis man ikke kaller på super så legger java inn et tomt kall på super når programmet kompileres: `super();`
- Hvis en klasse ikke har konstruktør så legger også java inn et tomt kall på super
- Man må derfor ha med super hvis konstruktøren i superklassen tar inn noen verdier (Ikke redefiner superklassens variabler i subklassens konstruktør)

Klassen Object

- Alle klasser arver fra klassen Object (automatisk)
- Metoder som clone(), equals() og toString()
 - Derfor kan man kalle på toString() før den er implementert



Referanser

- Variabel av type A kan referere til objekt av B, men ikke omvendt
- Kan derfor skrive:
 - A var1 = new A();
 - A var2 = new B();
 - B var4 = (B) var2;
- men ikke:
 - B var5 = new A();
 - B var6 = (B) new A();

```
class A {}  
  
class B extends A {}
```


Polymorfi

- Spesialisering i subklasser
- I stedet for å bruke instanceof
- Hvis en metode ikke er final kan metoden overskrives (@Override)
- Ulike subklasser kan da ha den samme metode-signaturen, men forskjellige implementasjon
- Hvis man ønsker å bruke superklassen sin implementasjon kan man skrive:
 - super.metodenavn();

```
// I klassen Hund
@Override
public void lagLyd() {
    System.out.println("Voff");
}
// I klassen Katt
@Override
public void lagLyd() {
    System.out.println("Mjau");
}
```

Interface

- Interface sikrer at en klasse har implementert noen metoder
- Kun abstrakte metoder, ligner på en abstrakt klasse uten implementasjoner og variabler (Kan ha konstanter)
- I java kan en klasse kun arve fra én annen klasse, men fra flere interfaces
- En klasse deler muligens også egenskaper med andre klasser som ikke har samme superklasse, kan da bruke interface
- Eksempler: List, Comparable, Iterable

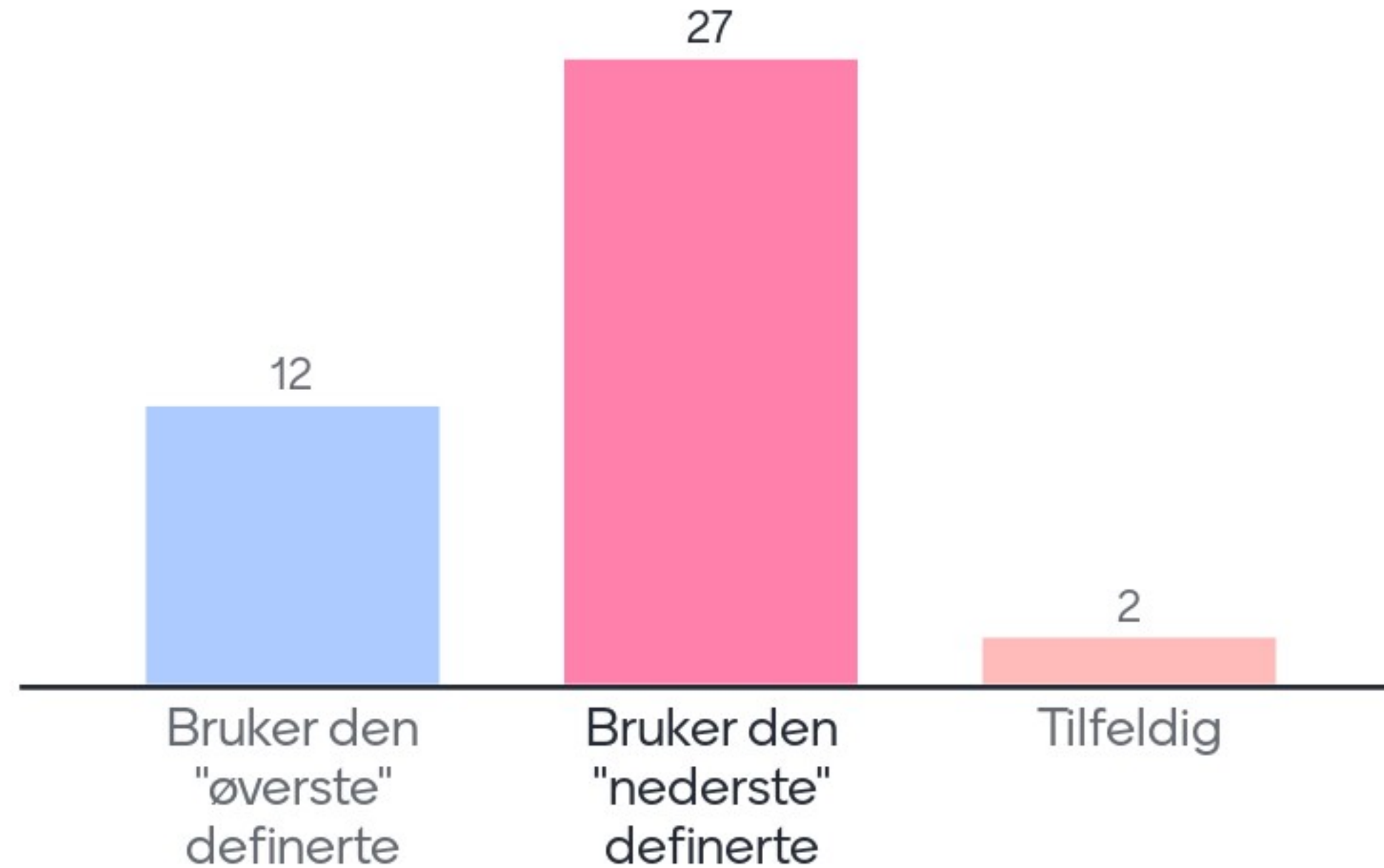
Interface

- Kan bruke interface som typen til en peker
- Example var1 = new B();
- Har da kun tilgang til de metodene deklarerert i interfacet

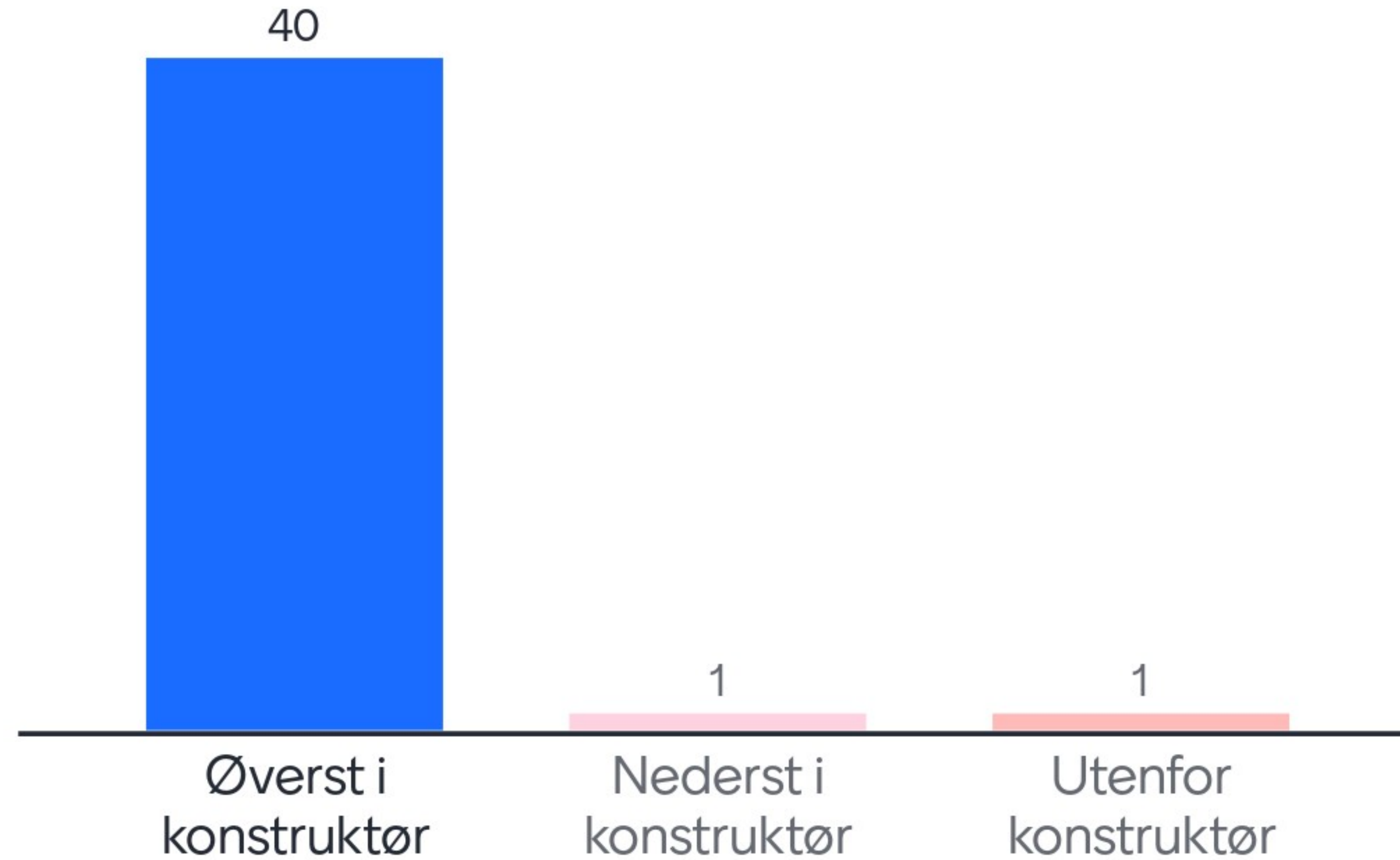
```
interface Example {  
    public void method1();  
}  
  
class A {}
```

```
class B extends A implements Example {  
    @Override  
    public void method1() {  
        System.out.println("Ex.");  
    }  
}
```

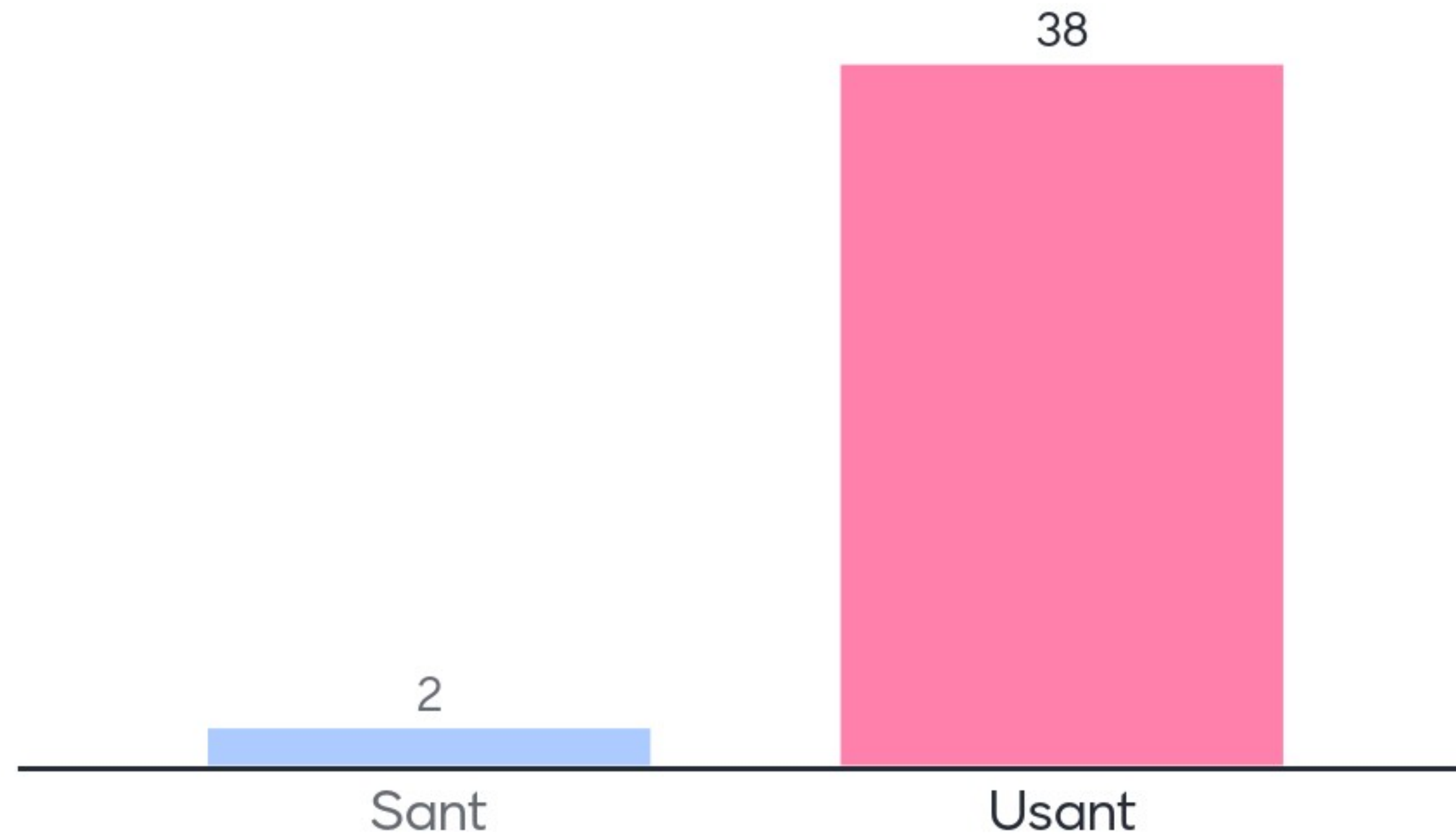
Hvordan systemet vet hvilken metode den skal benytte seg av dersom det er flere polymorfe metoder



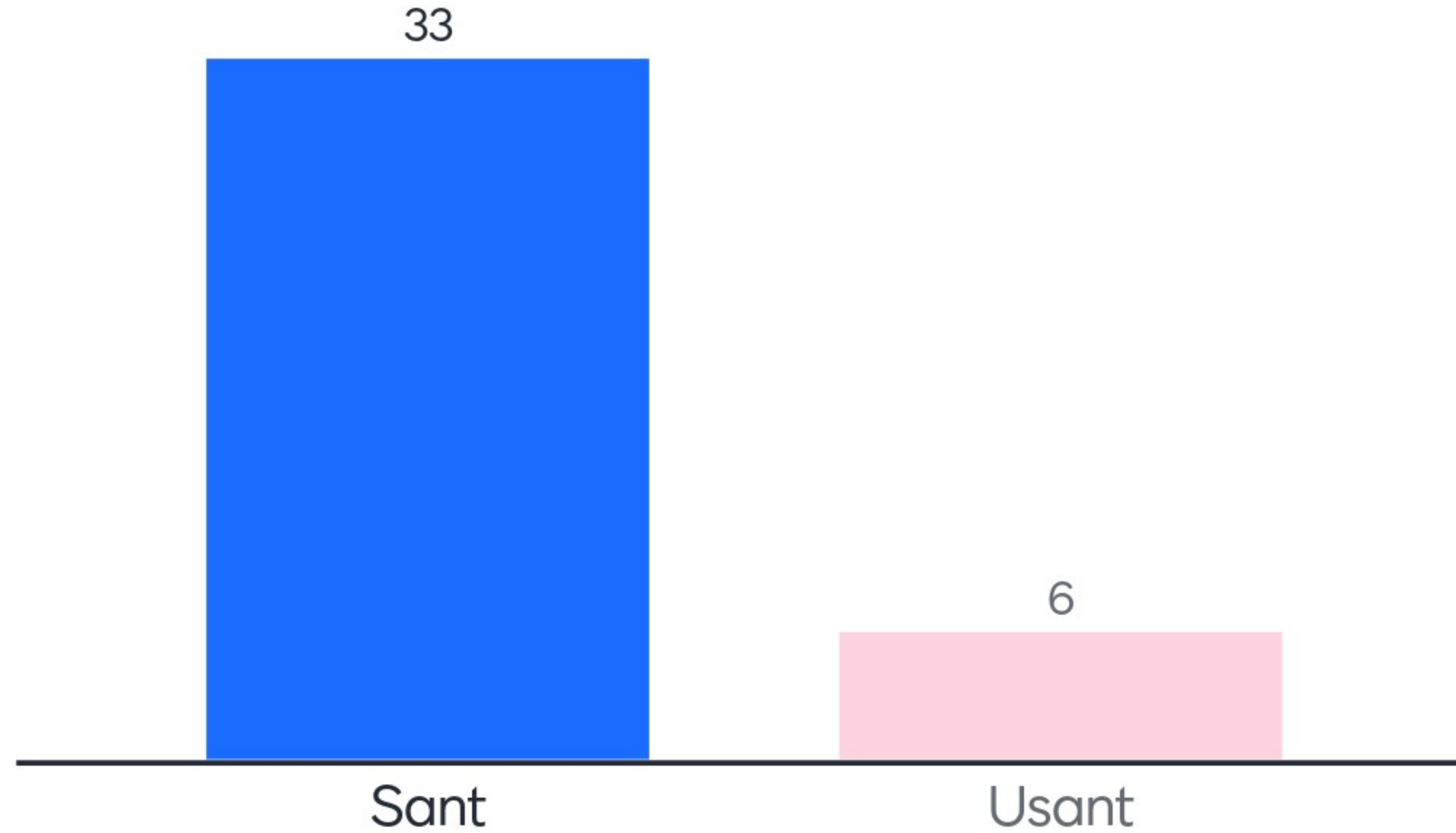
Et kall på `super()` må legges ...



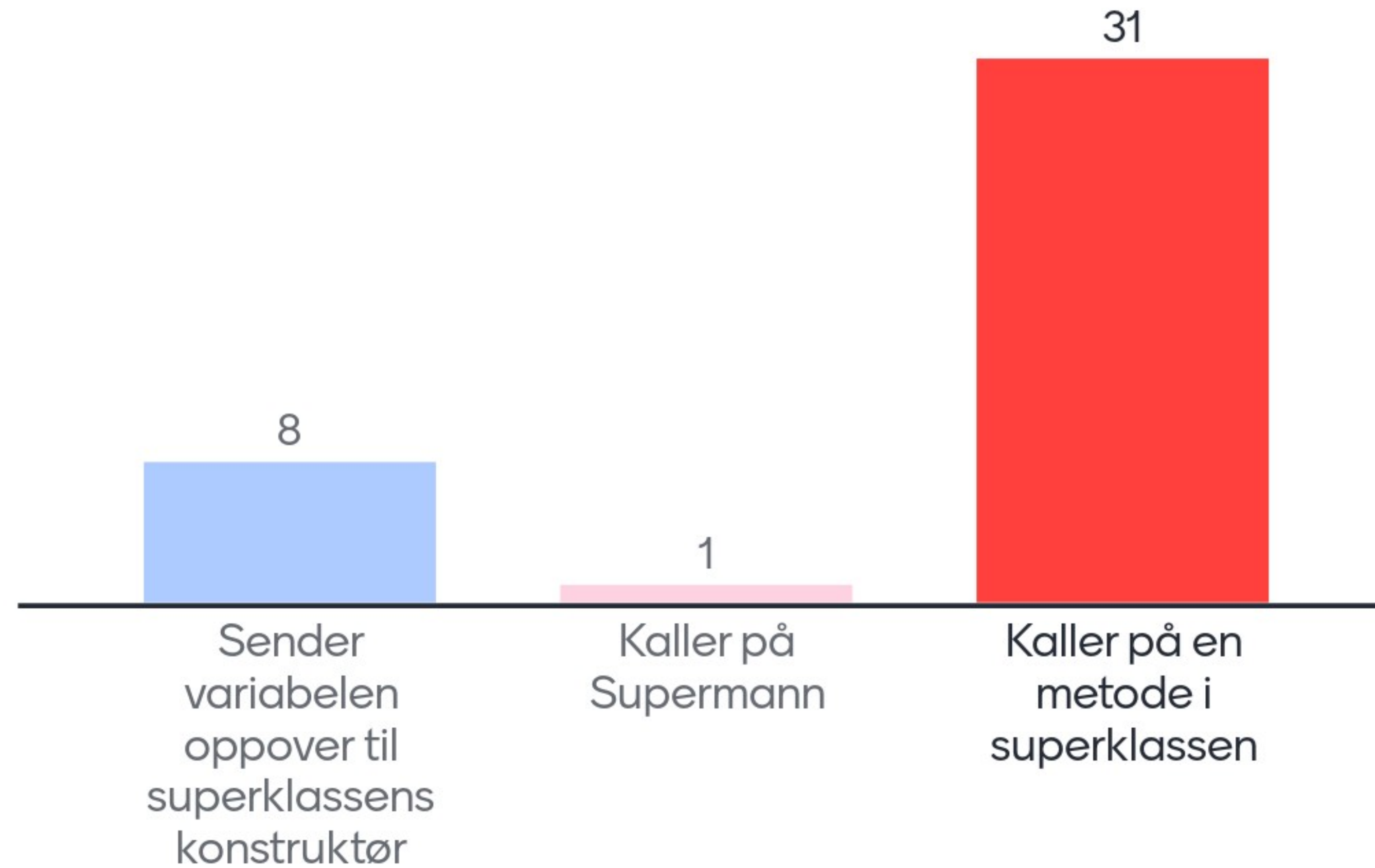
En klasse kan arve fra flere superklasser i Java



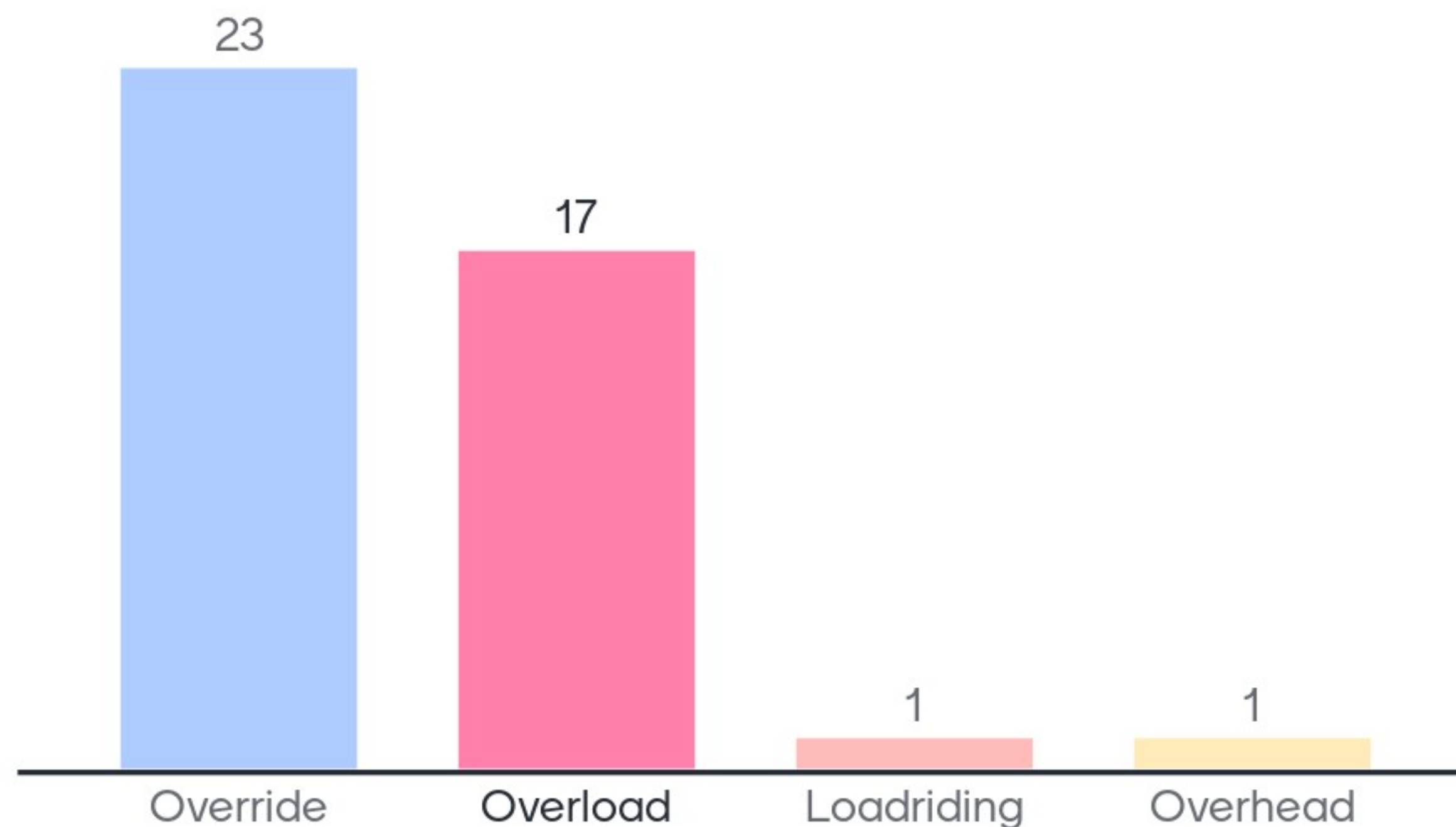
Et interface kan brukes som referansetype



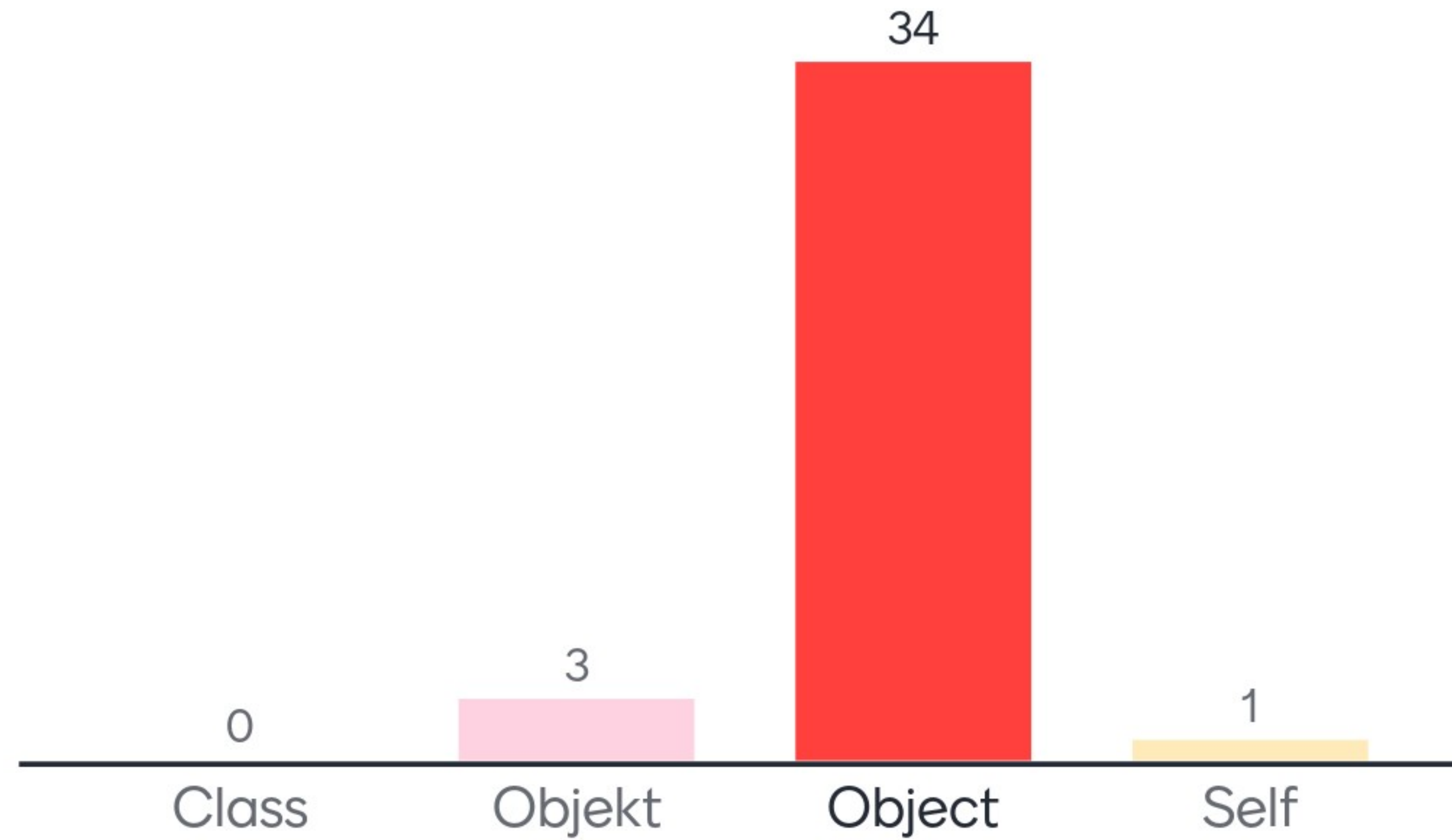
Å skrive "super."



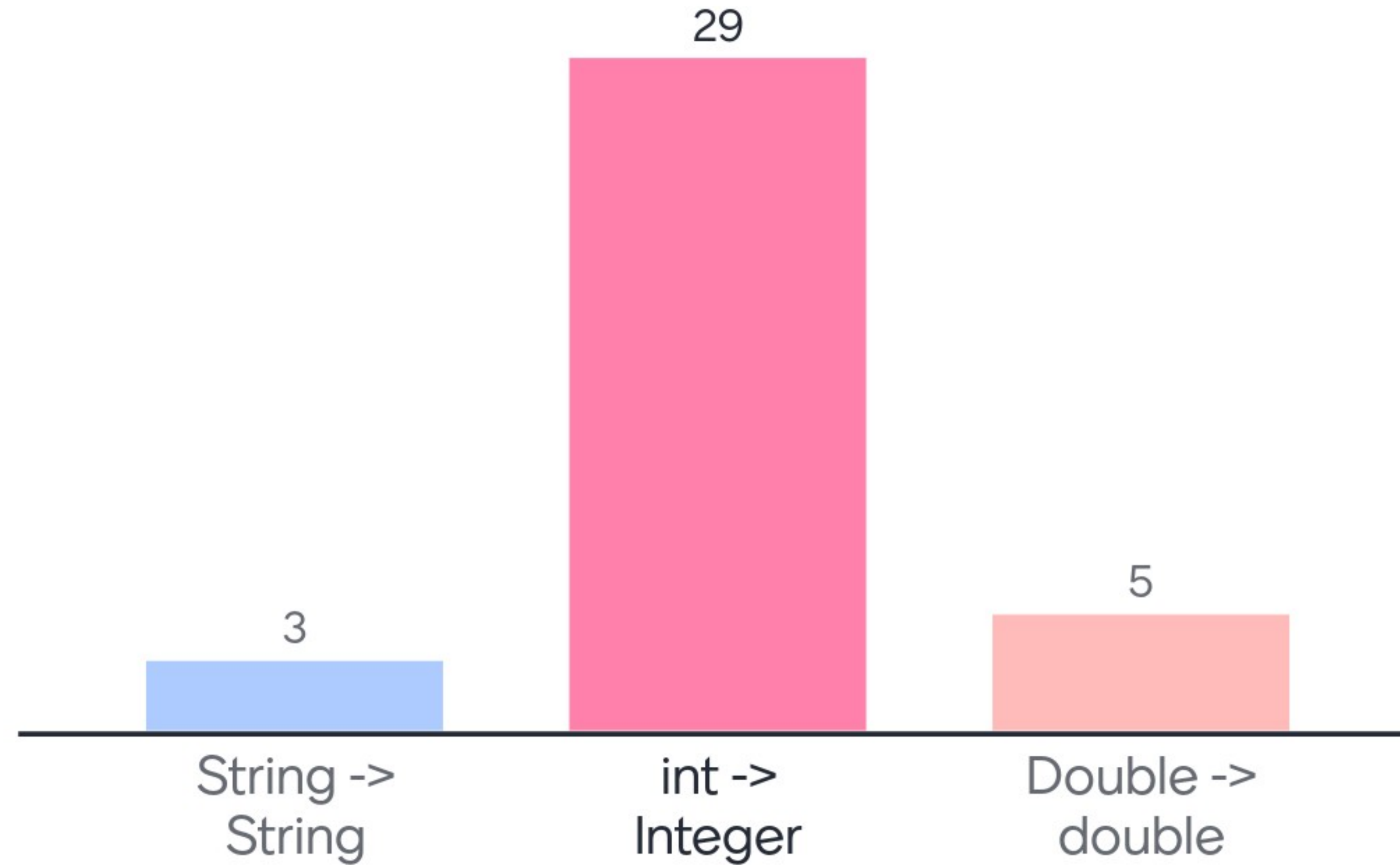
Å definere metoder med like navn men ulike signaturer kalles



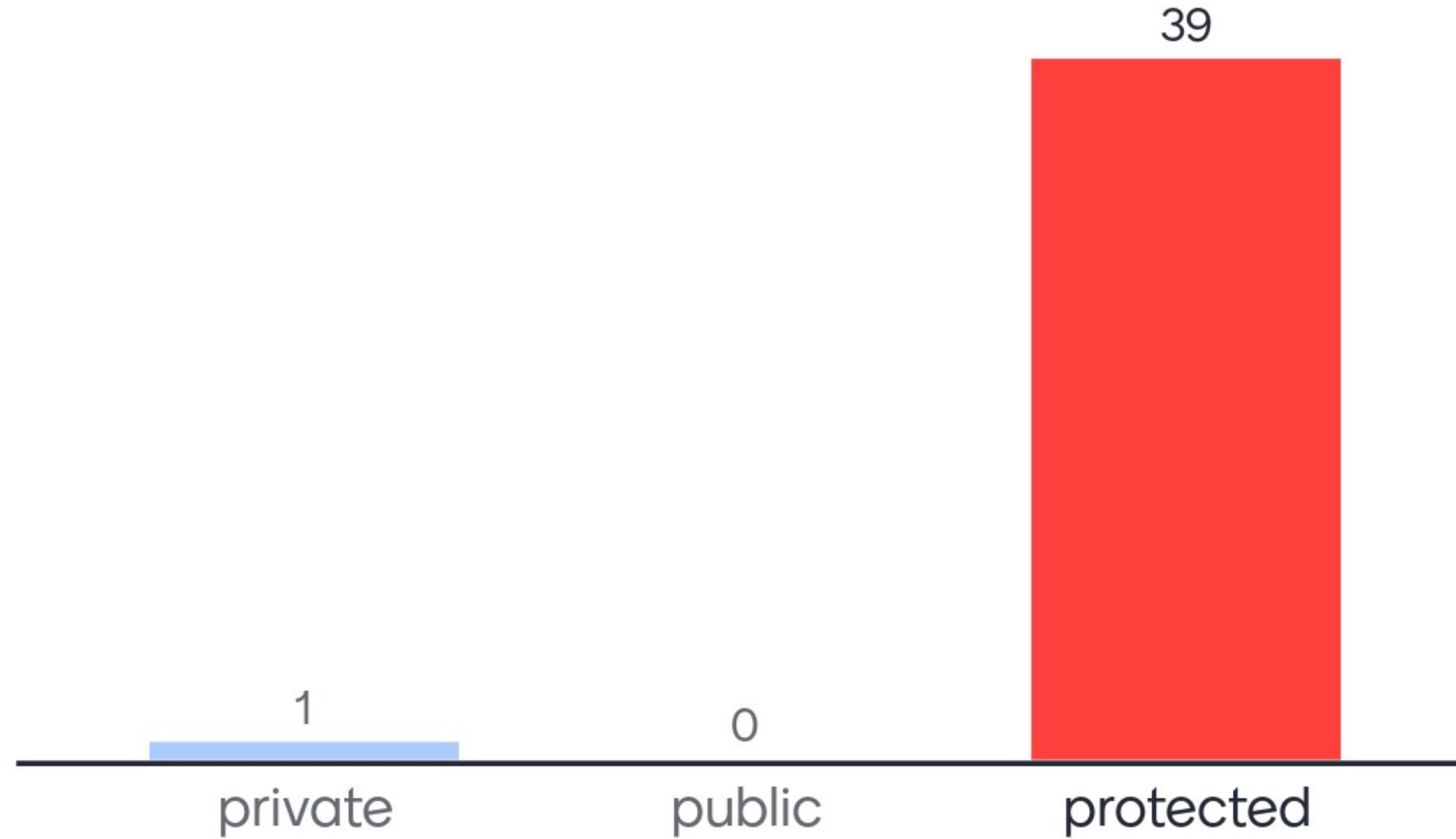
Alle klassers mor



Et eksempel på "boxing"



Gjør at kun subclasser kan se egenskapen



Forskjellen mellom en abstrakt klasse og interface

- Abstrakte klasser kan inneholde fullstendige implementasjoner
- Grensesnitt: kun signaturer
- Ingen innmat i grensesnitt (lite unntak)

Abstrakte klasser på eksamen

- Står sjeldent "skriv den abstrakte klassen ..."
- Lese mellom linjene
- Eks. "Alle stier er enten kjerreveier eller naturstier..."
- Nøkkelord som: *enten, kun, det ene eller det andre, skal ikke gå an å lage objekter av ...*