

# Principles of subclasses

Objekt-orienterad programmering och design

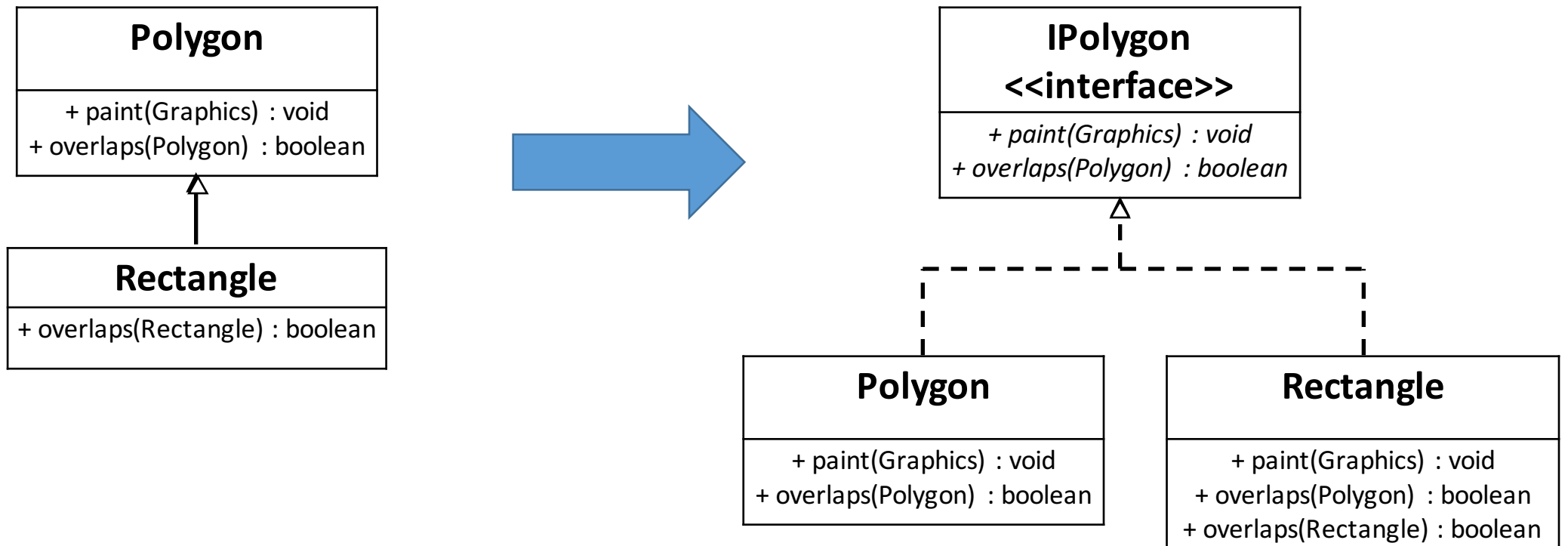
Alex Gerdes, 2016

# Implementation inheritance

- Subclassing, eller *implementation inheritance* (implementationsarv), ger oss två fördelar:
  - Polymorfism
    - Ett objekt av en subclass kan användas som om det var ett objekt av sin superclass.
    - Ger flexibilitet, "Plug-n-play"; kan e.g. arbeta med listor av objekt med superklassen som statisk typ, som vid run time kan vara av antingen superclass eller (någon) subclass.
  - Återanvändning av kod (code reuse)
    - Definiera gemensamma metoder i superklassen som kan ärvas av dess subclass(es).
    - Don't repeat yourself, ingen cut-n-paste.
    - Ger ett enda ställe för förändring vid behov – maintainability.

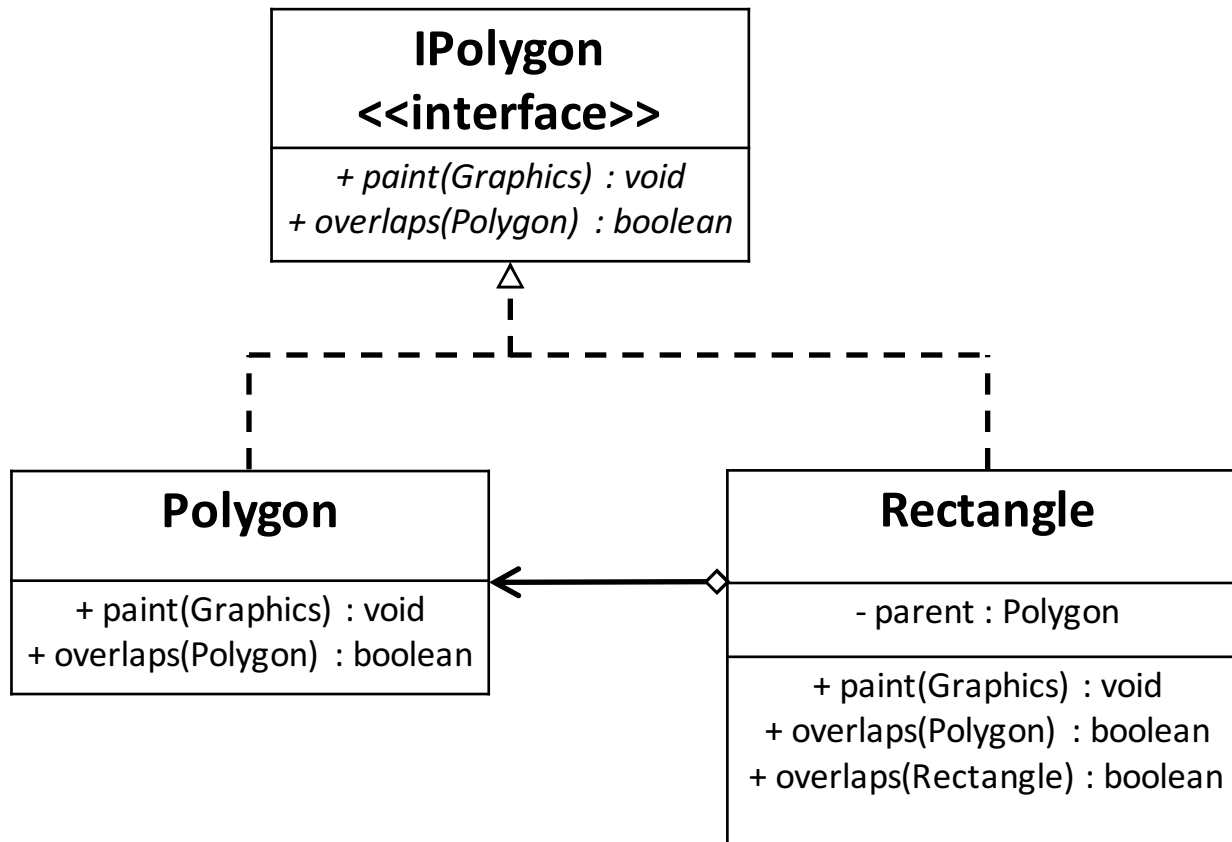
# Polymorfism reclaimed

- Polymorfism kan åstadkommas med hjälp av ett interface istället:



# Code reuse reclaimed

- Code reuse kan åstadkommas med hjälp av delegering istället:



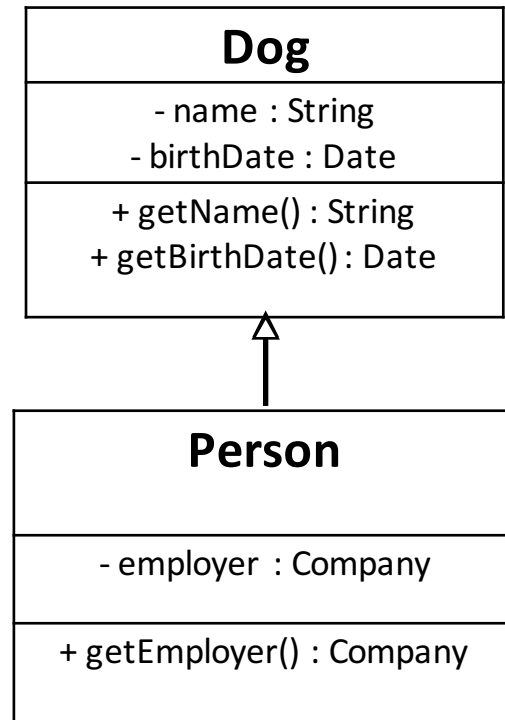
```
class Rectangle {  
  
    private Polygon parent = ...  
  
    public boolean  
        overlaps(Polygon other) {  
        parent.overlaps(other);  
    }  
  
}
```

"Ärvd" metod via delegering

# Motivation

- Antag att vi
  - har behov av att utnyttja polymorfism
  - har behov av att återanvända kod
- Är det då fritt fram att använda implementation inheritance?
- Ledande fråga: Svaret är nej, det är inte riktigt så enkelt.

# En person är en hund



- Vi (antar att vi) har behov av polymorfism (e.g. en lista som innehåller både hundar och personer).
- Vi har återanvändning av kod.
- Så vad är problemet??

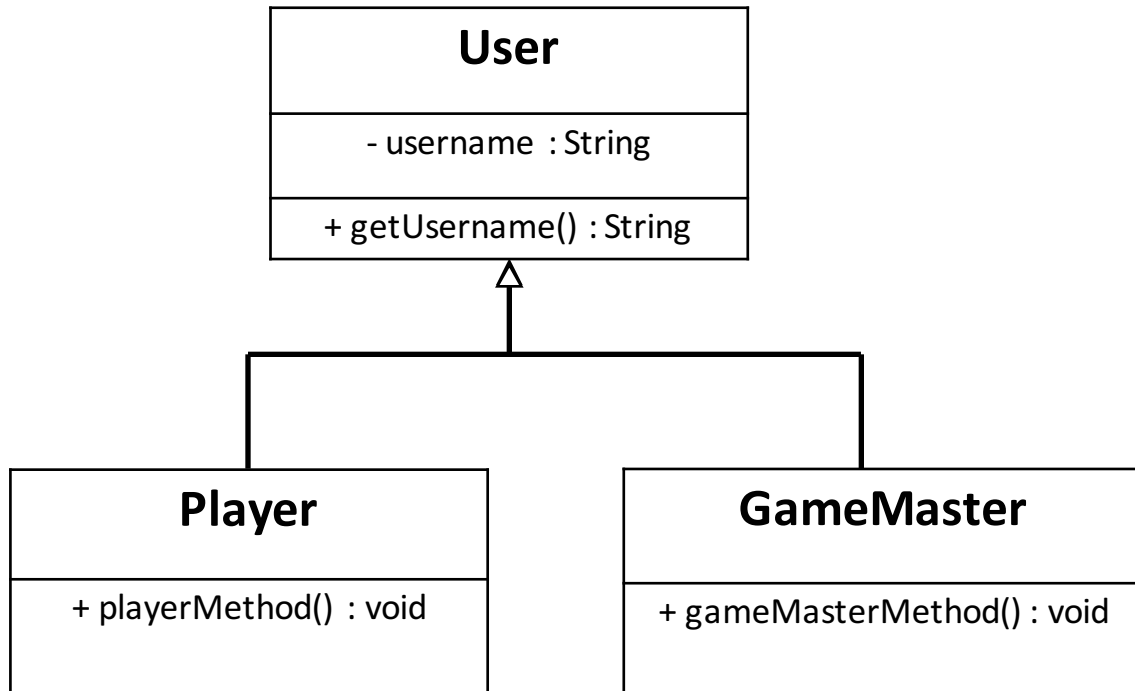
# Konceptuell "Is-A"

- Även om lösningen rent *tekniskt* åstadkommer det vi behöver, så är det inte en *konceptuellt* bra lösning. En person är inte en hund.
- Kod baserat på antagandet att en person är en hund kommer vara mycket förvirrande.
  - Vi tappar simplicity, readability, maintainability.
- Slutsats: Använd aldrig implementation inheritance om det inte föreligger ett faktiskt, konceptuellt "delmängds"-förhållande mellan sub- och superclass.

# Motivation – andra försöket

- Antag att vi
  - har behov av att utnyttja polymorfism
  - har behov av att återanvända kod
  - har ett konceptuellt "Is-a"-förhållande
- Är det fritt fram att använda implementation inheritance nu då?
  - Svar: Nej...

# Roller i ett rollspel



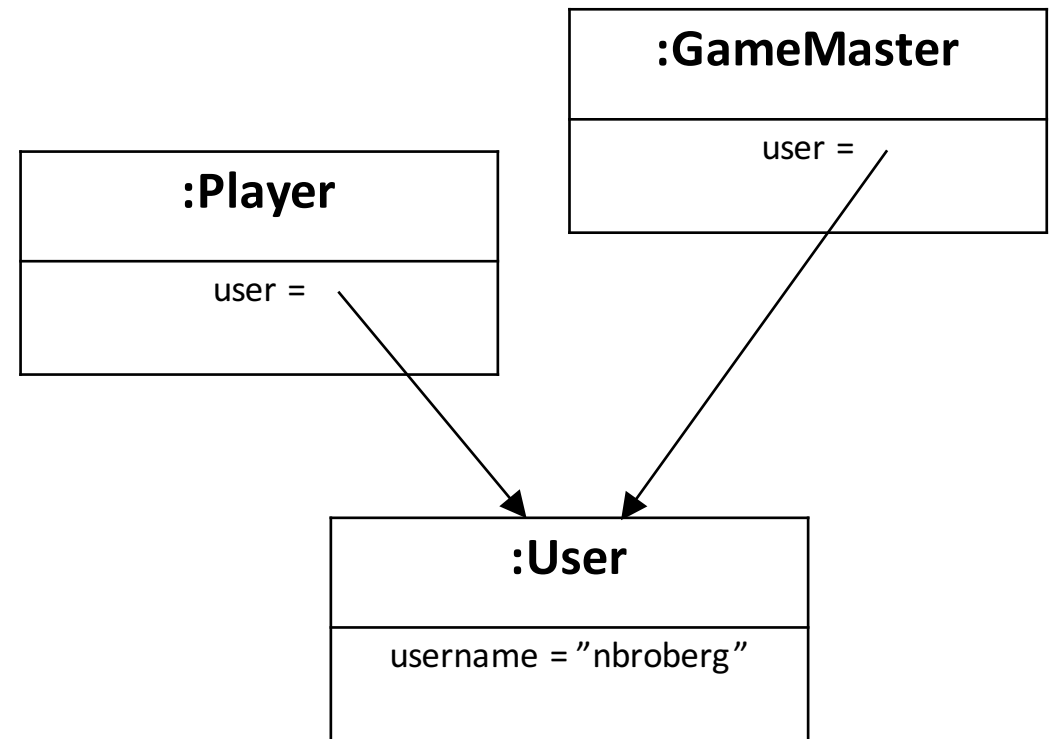
- Vi har behov av polymorfism (e.g. en lista av alla användare oavsett typ).
- Vi har kod vi vill återanvända.
- Både spelare och spelledare är konceptuellt användare.
- Så vad är problemet??

# Stabil "Is-A" med unika positioner

- En användare kan i vissa sammanhang vara en spelare, i andra sammanhang en spelledare. Hur representerar vi detta?
- Om vi använder ett `Player`-objekt för att representera användaren i rollen som spelare, och ett `GameMaster`-objekt för att representera användaren i rollen som spelledare, då kommer dessa två objekt att båda ärva all data (e.g. `username`) från `User`. Denna data dupliceras över två separata objekt. Ändras datan måste den uppdateras på båda ställena för att inte få inkonsekvens – problem!

# Delegering för roller

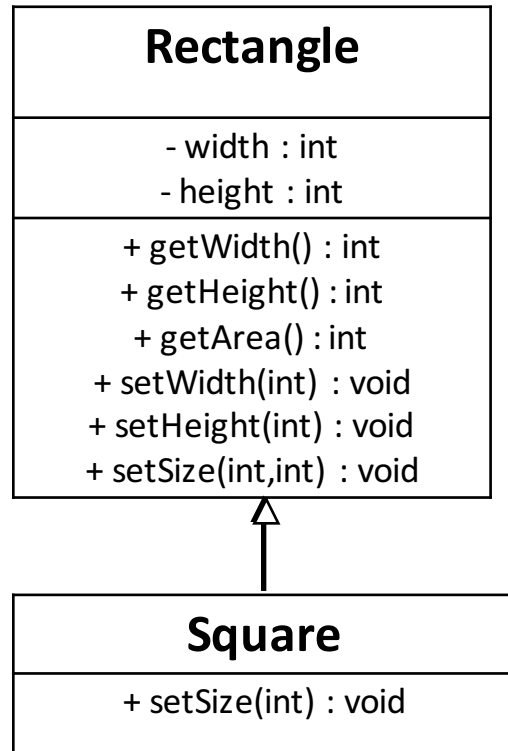
- Vid delegering kan *samma* User-objekt återanvändas för båda rollerna, vilket undviker problemet.
- Slutsats: Använd inte implementation inheritance för att representera *roller* som objekt kan ha – använd alltid delegering i sådana fall.
  - I en korrekt subclass-hierarki återfinns varje objekt på ett enda ställe, och ändras inte.



# Motivation – tredje försöket

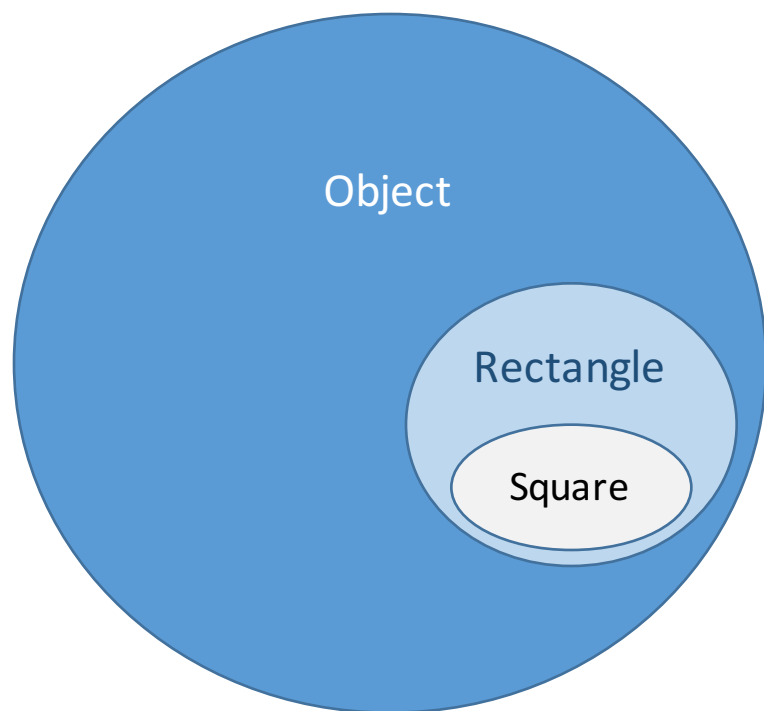
- Antag att vi
  - har behov av att utnyttja polymorfism
  - har behov av att återanvända kod
  - har ett konceptuellt **och stabilt** "Is-a"-förhållande **med unika positioner**
- Är det fritt fram att använda implementation inheritance nu då?
  - Svar: Nej...

# Squaring off

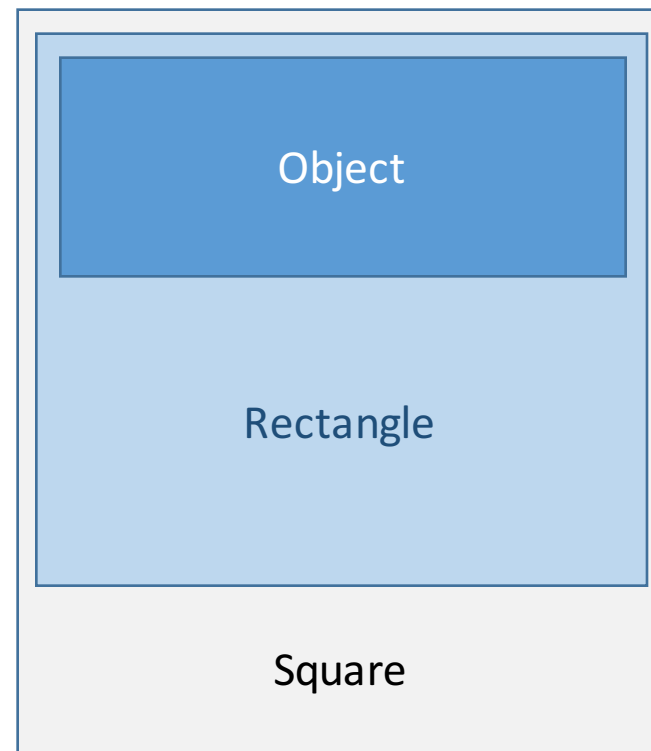


- Vi har behov av polymorfism.
- Vi har kod vi vill återanvända.
- En kvadrat är konceptuellt sett en rektangel, och kommer alltid att vara det (stabilt förhållande).
- Så vad är problemet??

# Två perspektiv på subklasser



Fokus på delmängder: Vad ett objekt *är*.



Fokus på beteende: Vad ett objekt *kan*.

# Delmängder av många anledningar, subklasser endast av en

- En delmängd kan uttryckas som en property:
  - Om  $B \subseteq A$ , då har vi  $B = \{ x \mid x \in A, P(x) \}$  för något  $P$ .
  - Av alla objekt i  $A$  är det bara de som uppfyller  $P$  som är i  $B$ .
- Sådana properties kan komma i otaliga former – endast *en* form av property lämpar sig för subklassning: när  $P(x)$  indikerar att elementen i  $B$  *kan* något som elementen i  $A$  *inte* kan.
  - Elementen i  $B$  har *utökat* beteende jämfört med elementen i  $A$ .

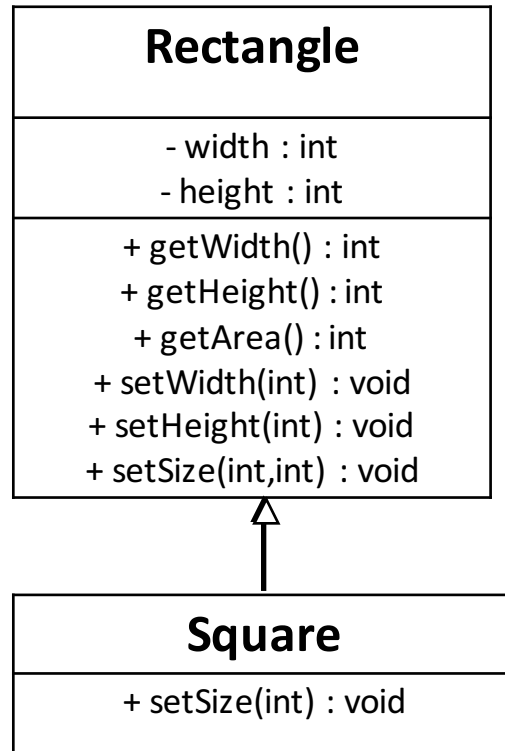
# Dåliga anledningar

- Några exempel på konceptuella "Is-a"-relationer som inte lämpar sig för subclassing:
  - $P(x)$  indikerar att  $x$  har *begränsat* beteende, i.e. *inte* kan något som övriga kan.
    - E.g. en åskådare i ett dataspel är en spelare som inte kan agera.
  - $P(x)$  indikerar en extern begränsning.
    - E.g. en kvadrat är en rektangel som måste ha lika långa sidor.
  - $P(x)$  indikerar ett temporärt tillstånd.
    - E.g. en *öppen* fil är en fil med *temporärt* utökat beteende, e.g. läsa från och skriva till.
    - Mer om hur vi hanterar tillstånd (state) i block 6-2.

# Motivation – fjärde försöket

- Antag att vi
  - har behov av att utnyttja polymorfism
  - har behov av att återanvända kod
  - har ett konceptuellt och stabilt "Is-a"-förhållande med unika positioner
  - har ett *publikt gränssnitt* för subklassen som är en *utökning* av superklassens
- Är det fritt fram att använda implementation inheritance nu då?
  - Svar: Fortfarande nej...

# Forcerat gränssnitt



```
class Square extends Rectangle {

    @Override
    public void setSize(int w, int h) {
        this.setSize(w);
    }

    // ...

}
```



# Live code

- ...

# Liskov Substitution Principle

If for each object **o1** of type **S** there is an object **o2** of type **T** such that for all programs *P* defined in terms of **T**, the behavior of *P* is unchanged when **o1** is substituted for **o2**, then **S** is a subtype of **T**.  
(Barbara Liskov)

- *S* är en *äkta* subtyp till *T* endast om, för varje publik metod som finns i både *S* och *T*:
  - *S*'s metod godtar alla värden som *T*'s metod godtar
  - *S* gör alla beräkningar på denna indata som *T* gör (och kanske fler).

# Motivation – femte försöket

- Antag att vi
  - har behov av att utnyttja polymorfism
  - har behov av att återanvända kod
  - har ett konceptuellt och stabilt "Is-a"-förhållande med unika positioner
  - har ett publikt gränssnitt för subklassen som är en utökning av superklassens
  - har ett publikt *beteende* för subklassen som är en utökning av superklassens
- Är det fritt fram att använda implementation inheritance nu då?
  - Svar: Ja, men...

# Slutsatser

- För att implementation inheritance alls ska vara lämpligt:
  - Ett konceptuellt "Is-a"-förhållande måste föreligga.
    - Om inte – använd ett gemensamt interface och/eller delegering.
  - Vi måste ha (möjligen framtida) behov av polymorfism.
    - Om inte – använd enkel delegering.
  - Vi måste ha behov av att återanvända kod.
    - Om inte – använd interfaces.
  - Subklassens publika gränssnitt ska inkludera superklassens.
    - Om inte – bryt ut det gemensamma gränssnittet till ett interface som båda implementerar.
  - Subklassen ska lägga till extra beteenden och beräkningar, inte ändra redan befintliga.
    - Om inte – hitta minsta gemensamma nämnare i beteende och gör denna antingen till (abstrakt) superklass för båda, eller en (privat) klass som båda delegerar till.

# Med det sagt...

- Nu är det fritt fram att använda implementation inheritance!
  - ... men fortfarande inte nödvändigt, eller ens rekommenderat.
- Fundera alltid ett varv extra om det är motiverat – att använda interfaces och/eller delegering istället är alltid:
  - mer flexibelt – du kan bilda explicita relationer mellan objekten som inte begränsas av subclass-superclass.
  - mer precist – du kan välja exakt de delar som behövs för code reuse och/eller polymorfism.
  - ... men mer verbost – *kan* göra koden mer svårläslig.
- Glöm inte att du bara har *en* möjlighet till att välja superclass (single inheritance), men kan implementera obegränsat antal interfaces.

# Live code

- ...

# Starkt beroende

- Implementation inheritance innebär ett starkt beroende på superklassen. Starka beroenden är dåliga för våra mål: extensibility, reusability, maintainability.
- Mycket mer om beroenden fr o m block 4-2.

Favor composition over inheritance.

What's next

Block 3-2:

Subtyping, co- och contra-variance