**TDA 545:** Objektorienterad programmering

# Föreläsning 9:
# **Arv och UML**

Magnus Myréen

Chalmers, läsperiod 1, 2015-2016

# Quiz

Vad betyder `static` ?

Varför skriver man `get`-metoder?

```
public int getPos() {
    return pos;
}
```

Varför deklarerar man oftast variabler som `private` ?

Vad är en specifikation?

# Idag

Läsanvisning: kap 10 och lite 15; för nästa gång: kap 10 och 11

- arv (`extends`)
- `Object`
- `instanceof`
- lite UML

# Idén med arv (inheritance)

I Java kan klasser ärva egenskaper från en annan klass.

# Idén med arv (inheritance)

I Java kan klasser ärva ~~egenskaper~~ från en annan klass.

metoder och variabler

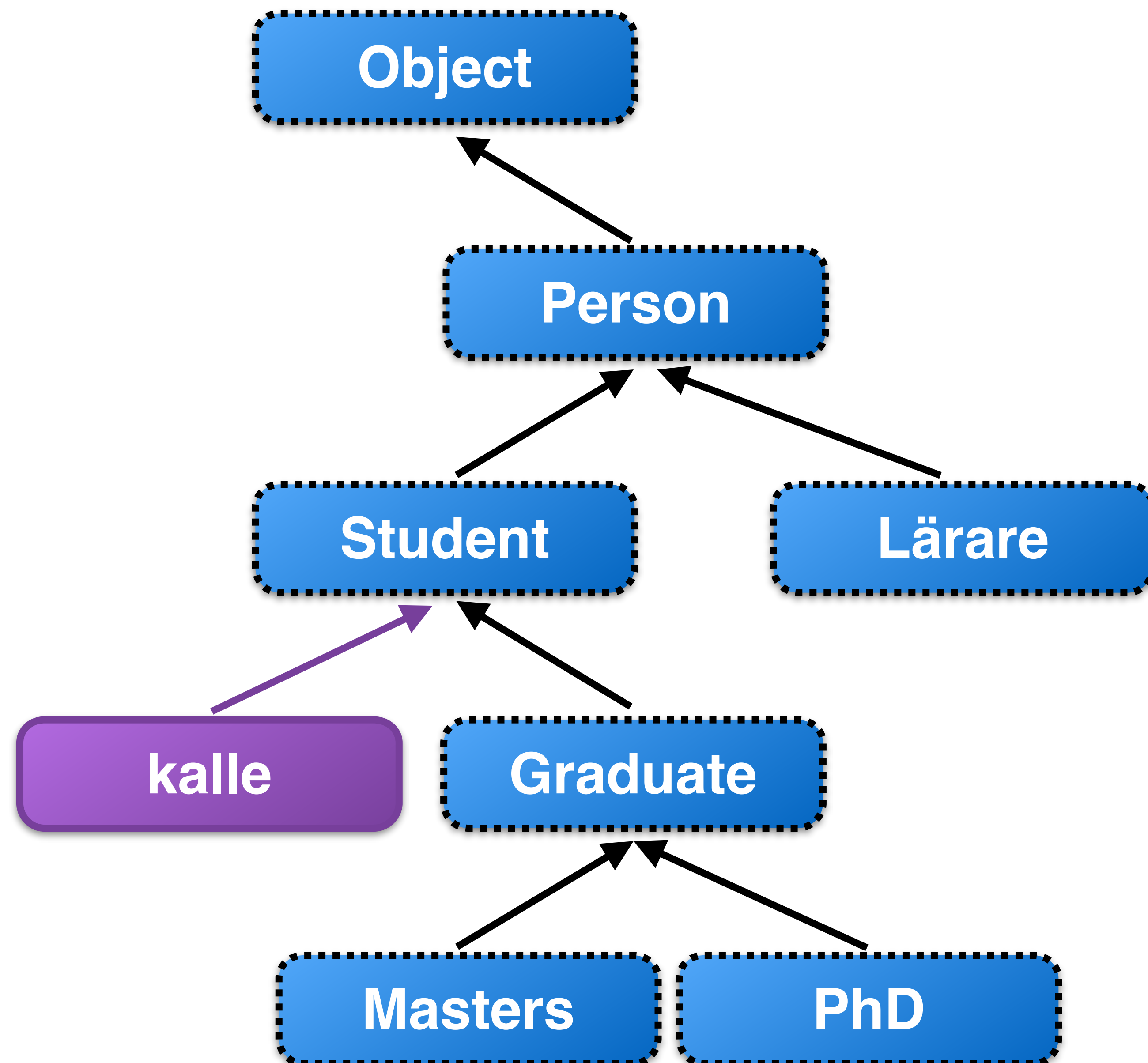
I Java kan klasser ärva <u>kod</u> från en annan klass.

*Man slipper att skriva samma kod flera gånger.*

*… och dessutom vet Java att klasserna är relaterade.*

*(dvs typ kompatibla)*

# Arv (inheritance)




```
public class Student extends Person { ... }
```

Mycket av det vi vet om kalle har med studenter och personer att göra.

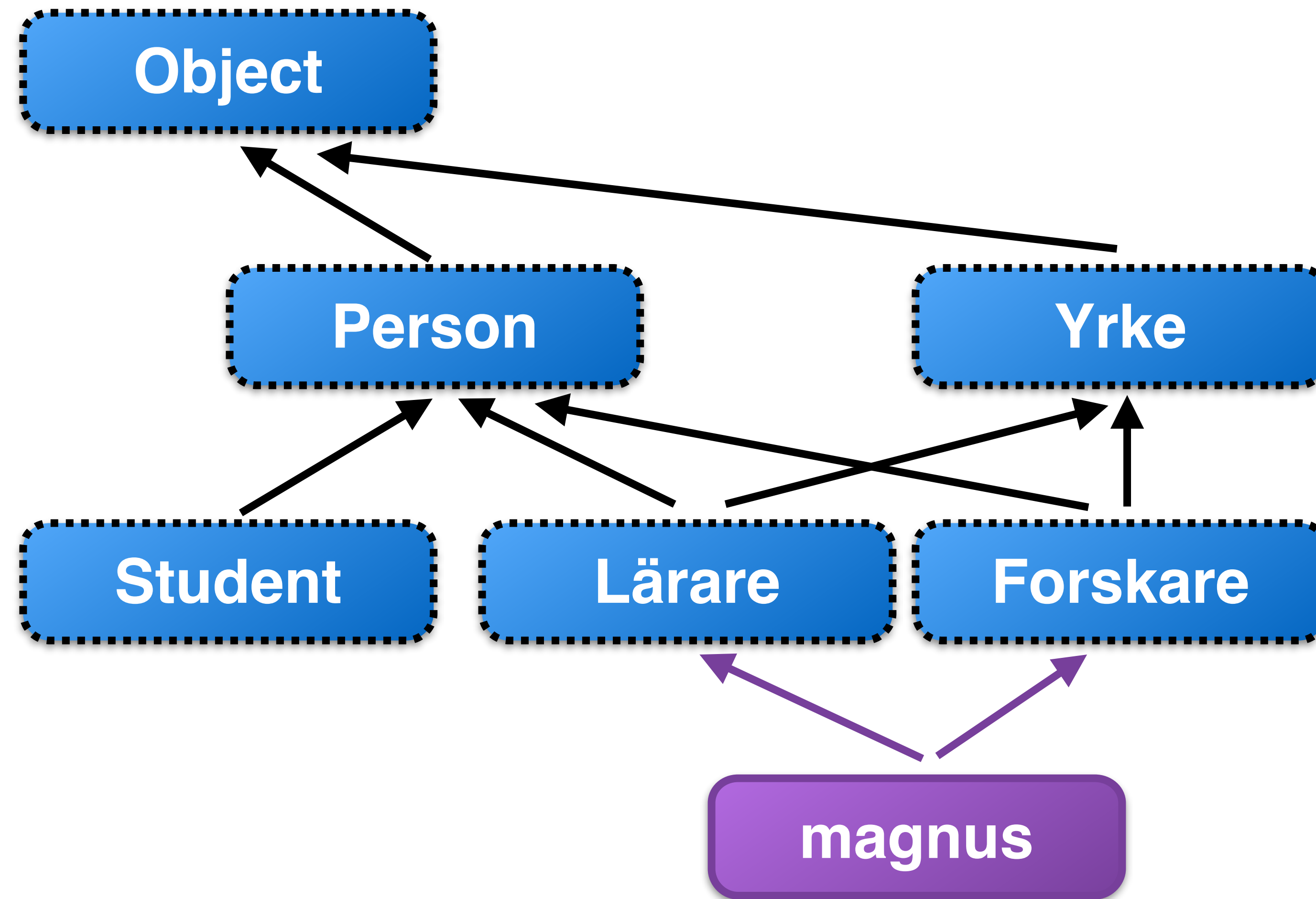
En **Student** är en “specialiserad” **Person**.

Klasser organiseras i trädliknande strukturer.

Utgående från en förälderklass ("superklass”) skapar man en ny klass, en barnklass ("subklass”) som ärver alla egenskaper (tillstånd och beteende) från superklassen som i sin tur ärver från sin superklass.

# Multipelt arv finns i verkligheten

***Men inte i Java!*** Java tillåter *inte* multipelt implementationsarv.



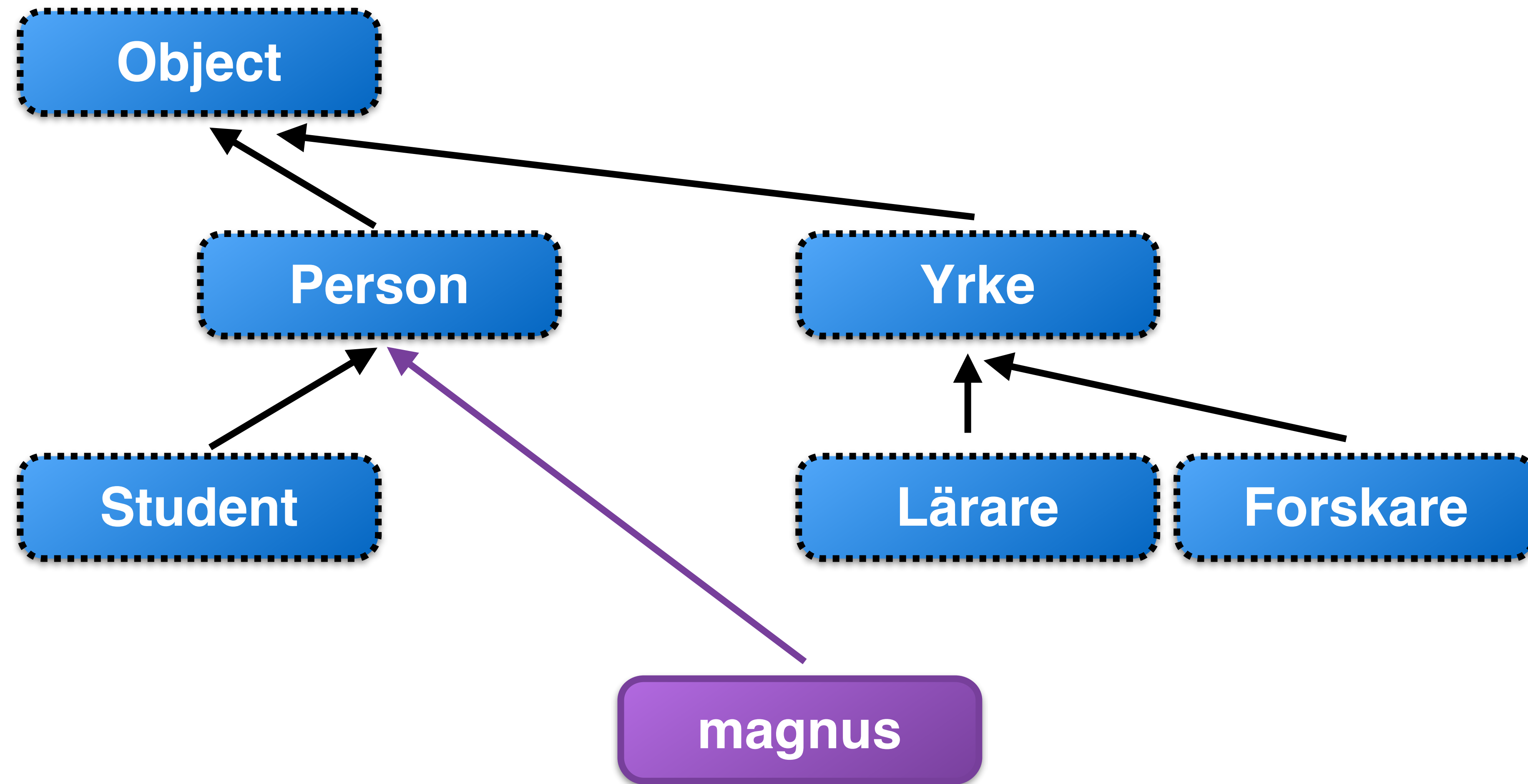

Hur hitta lämpliga abstraktioner, klasser?

**Tänk:** Vad är väsentligt för mitt program?
Hur kan jag använda klasserna?

(Borde yrke vara
tillstånd hos personer?)

# Multipelt arv finns i verkligheten

***Men inte i Java!*** Java tillåter *inte* multipelt implementationsarv.




Hur hitta lämpliga abstraktioner, klasser?

**Tänk:** Vad är väsentligt för mitt program?
Hur kan jag använda klasserna?

(Borde yrke vara tillstånd hos personer?)

# Överskuggning, metodbindning

## Överskuggning (Override)

(jämför: överlagring = overloading)

Man kan ändra beteendet och lägga till nya egenskaper och beteende till en klass vid arv.

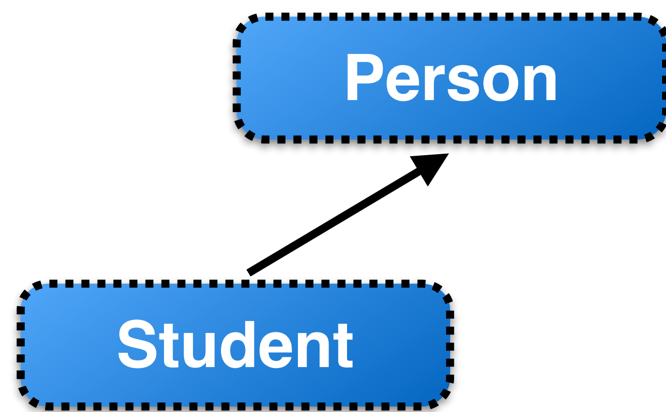
Dock inte ta bort saker.

Undantag eller ändrat beteende hanteras genom att en subklass överskuggar sitt arv från superklassen genom att ha en metod med samma namn och samma parameterprofil (signatur).

## Metodbindning:

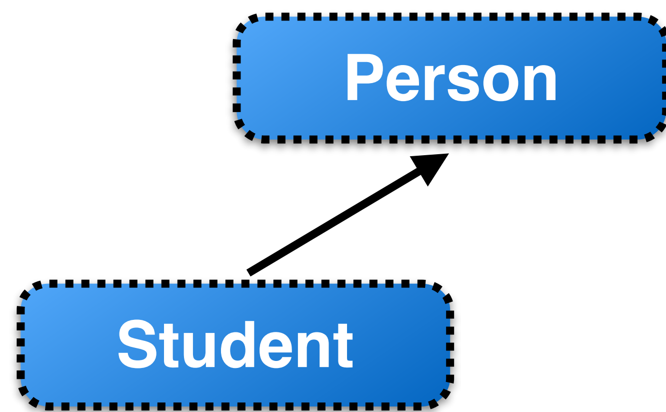
När man skall söka efter en metod så börjar man där man är (meddelandemottagarens klass). Finns där ingen metod med rätt namn och parameterprofil så söker man i föräldern osv.

# Exempel




```
public class Person {
  private String name;
  private int age;
  public Person(String n, int ag) {
     name = n; age = ag;
  }
  public String toString( ) {
    return getName();
  }
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }
  public final void setName (String n) {
    name = n;
  }
}
```

```
public class Student extends Person {
  private String uniName;
  public Student(String n, int age,
                 String uni) {
    super(n, age);
    uniName = uni;
  }
  public String toString() {
    return super.toString() +
             " student at " + uniName;
  }
  public String getUniName () {
    return uniName;
  }
}
```

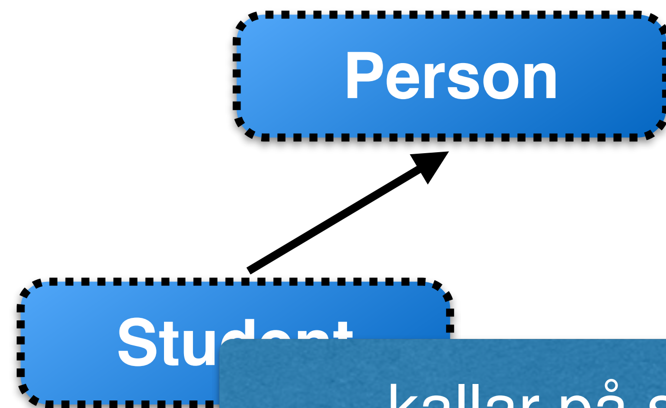



# Exempel

```
public class Person {
  private String name;
  private int age;
  public Person(String n, int ag) {
     name = n; age = ag;
  }
  public String toString( ) {
    return getName();
  }
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }
  public final void setName (String n) {
    name = n;
  }
}
```

```
public class Student extends Person {
  private String uniName;
  public Student(String n, int age,
                 String uni) {
    super(n, age);
    uniName = uni;
  }
  public String toString() {
    return super.toString() +
             " student at " + uniName;
  }
  public String getUniName () {
    return uniName;
  }
}
```

# Exempel




kallar på superklassens konstruktör, dvs Persons kons.

säger att vi ärver från **Person**

```
public class Person {
  private String name;
  private int age;
  public Person(String n, int ag) {
     name = n; age = ag;
  }
  public String toString( ) {
    return getName();
  }
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }
  public final void setName (String n) {
    name = n;
  }
}
```

```
public class Student extends Person {
  private String uniName;
  public Student(String n, int age,
                 String uni) {
    super(n, age);
    uniName = uni;
  }
  public String toString() {
    return super.toString() +
             " student at " + uniName;
  }
  public String getUniName () {
    return uniName;
  }
}
```

betyder att subklasser (some t.ex. Person) *inte* får överskugga denna metod.

kallar på superklassens version av metoden **toString**

# Arv (inheritance)

Med arv, dvs "**extends**":

- får man en kopia av alla variabler och metoder från superklassen
- man kan lägga till nya variabler och metoder
- man kan justera existerande metoder genom överskuggning

- man kan inte ta bort saker, t.ex. (public kan inte bli private)
- **konstruktören ärvs inte** (använd super(…))
- **privata variabler och metoder ärvs inte**

ja… privata variabler ärvs nog, men man kommer inte åt dem direkt: man måste använda get och set metoder som är **public**

# Exempel

```java
public class Queue {

    private Object[] q;
    private int l;

    /** Constructs an empty queue. */
    public Queue () {
        this.q = new Object[10];
        this.l = 0;
    }

    /** Returns true if the queue is empty. */
    public boolean isEmpty () {
        return (this.l == 0);
    }

    /** Returns the first element in the queue, if the queue is
     * non-empty. */
    public Object getFirst () {
        return this.q[0];
    }

    /** Adds an element to the back of the queue. */
    public void put (Object o) {
        if (l < q.length) {
            q[l] = o;
            l = l+1;
        } else {
            Object[] tmp = new Object[q.length+1];
            for (int i=0; i<q.length; i++) {
                tmp[i] = q[i];
            }
            tmp[q.length] = o;
            q = tmp;
            l = l+1;
        }
    }

    /** Removes the first element from the queue, if the queue is
     * non-empty. */
    public void pop () {
        if (!(isEmpty())) {
            for (int i=1; i<l; i++) {
                q[i-1] = q[i];
            }
            l = l-1;
            q[l] = null;
        }
    }

    public String toString() {
        String str = "";
        for (int i=0; i<this.l; i++) {
            str = str + " " + this.q[i];
        }
        return str;
    }

}
```

```java
public class QueueWithLog extends Queue {

    private int numberOfPuts = 0;
    private static int allPuts = 0;

    public void put (Object o) {
        super.put(o);
        numberOfPuts = numberOfPuts + 1;
        allPuts = allPuts + 1;
    }

    public String toString() {
        return super.toString() + " puts=" + numberOfPuts;
    }

    public static int getAllPuts() {
        return allPuts;
    }

}
```

```java
public class Main {

    public static void main(String[] args) {
        QueueWithLog t1 = new QueueWithLog();
        QueueWithLog t2 = new QueueWithLog();
        t1.put("A");
        t1.put("B");
        t1.put("C");
        t1.put("D");
        t1.put("E");
        t2.put("X");
        t2.put("Y");
        t2.put("Z");
        System.out1.println("t1 = " + t1.toString());
        System.out1.println("t2 = " + t2.toString());
        System.out1.println(QueueWithLog.getAllPuts());
    }

}
```

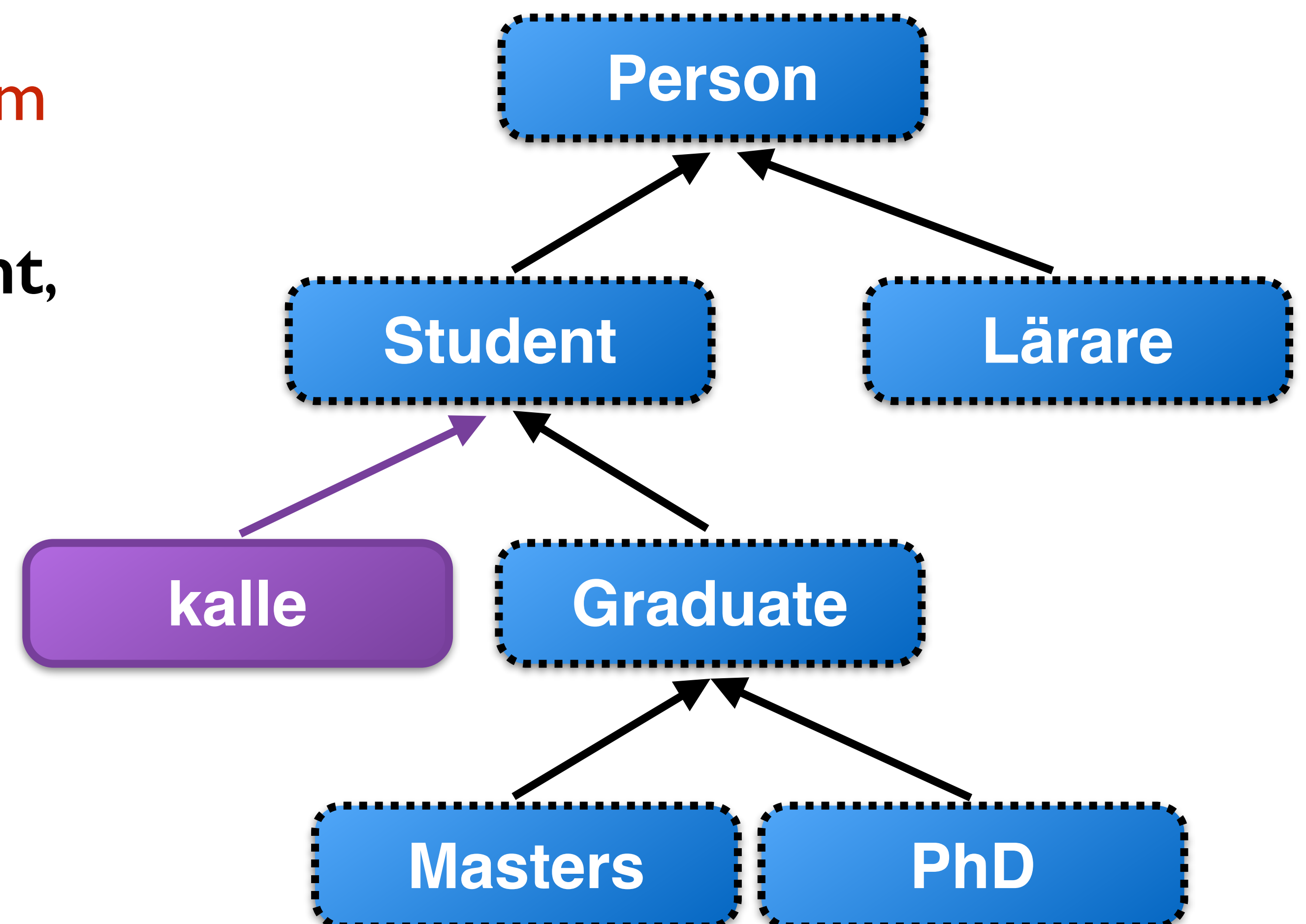
# Typ kompabilitet 1

(type compatible)

I vilken klass som helst kan man skriva:

```
public static boolean isOlder (Person p1, Person p2) {
  return p1.getAge() > p2.getAge();
}
```

Vi kan anropa med objekt av typ **Person** eller med något objekt som är en **Person** dvs objekt av alla subklasser till **Person** som **Student**, **PhD**, **Lärare** osv.



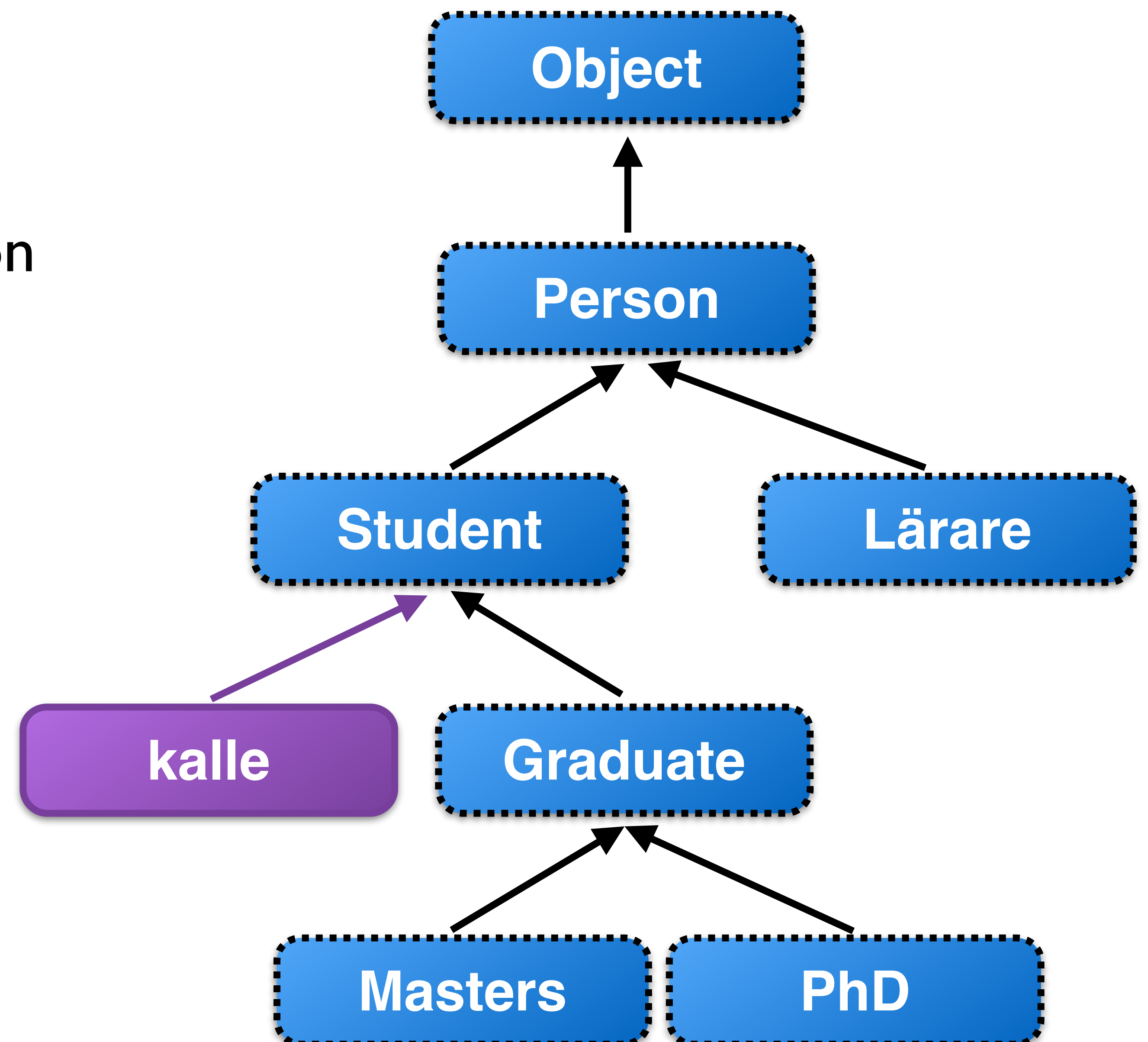
# instanceof

(instans av)

`instanceof` runtime-testar om ett objekt är instans av en viss klass. Om uttrycket

```
exp instanceof ClassName
```

är sant så är *exp* en instans av *ClassName* eller en instans av någon av *ClassName*'s subklasser.



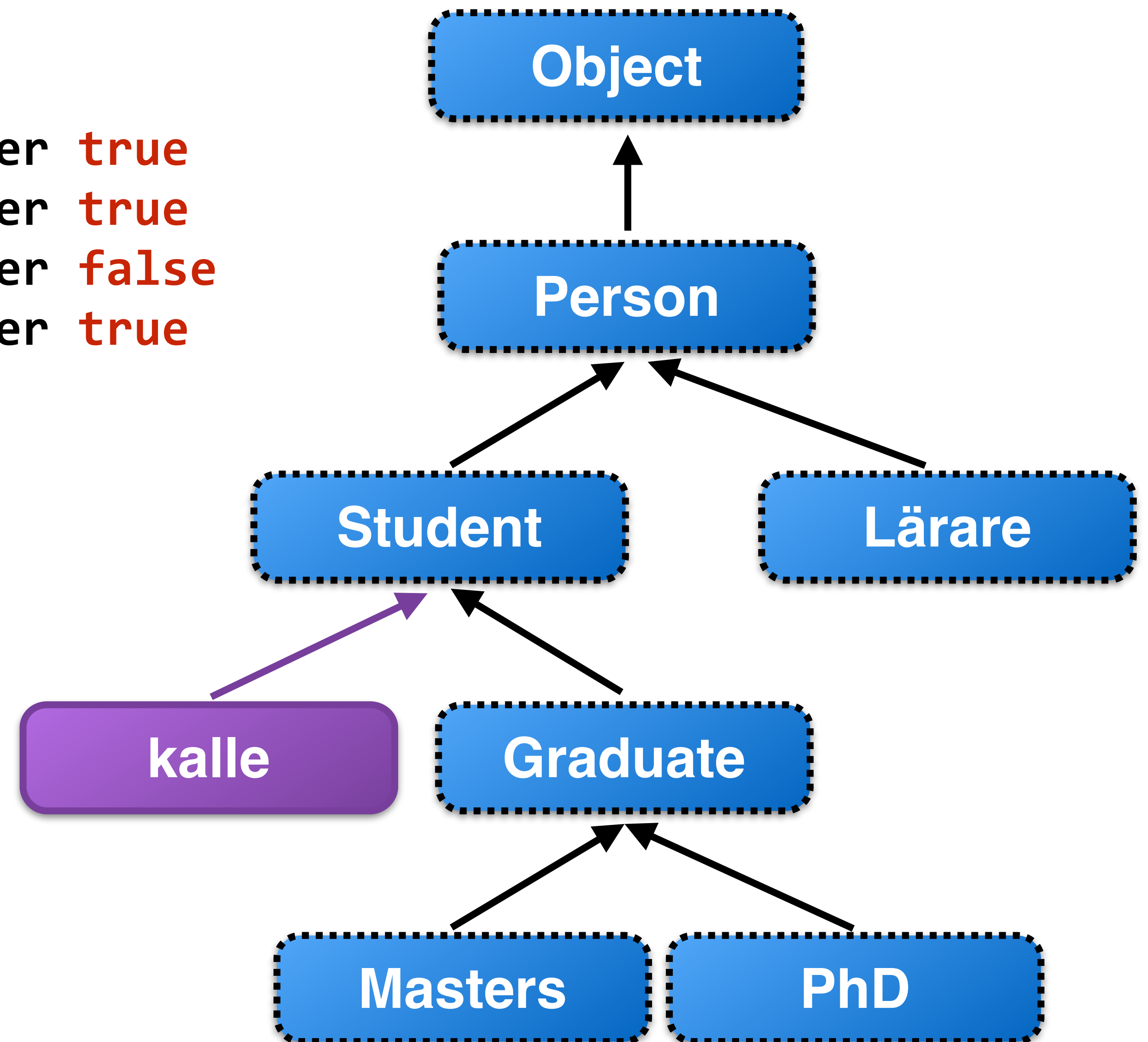

*Obs.* att det är en rätt svag test eftersom arv ju är transitivt, i vårt student-exempel så är en phd en instans av Phd, Graduate, Student, Person och Object.

# instanceof

(instans av)

```
PhD phd = new PhD(...);
Student stud = new Student(...);
Person p = new Student();
if( phd instanceof PhD ) ..      // ger true
if( phd instanceof Student ) .. // ger true
if( stud instanceof PhD ) ..    // ger false
if( p instanceof Student ) ..   // ger true
```

instanceof används ofta före en typkonvertering för att vara säker på att den går bra. Man kan alltid konvertera "uppåt" men inte nedåt utan problem och då använder man instanceof för att avgöra om det går också nedåt.



# statisk eller dynamisk typ

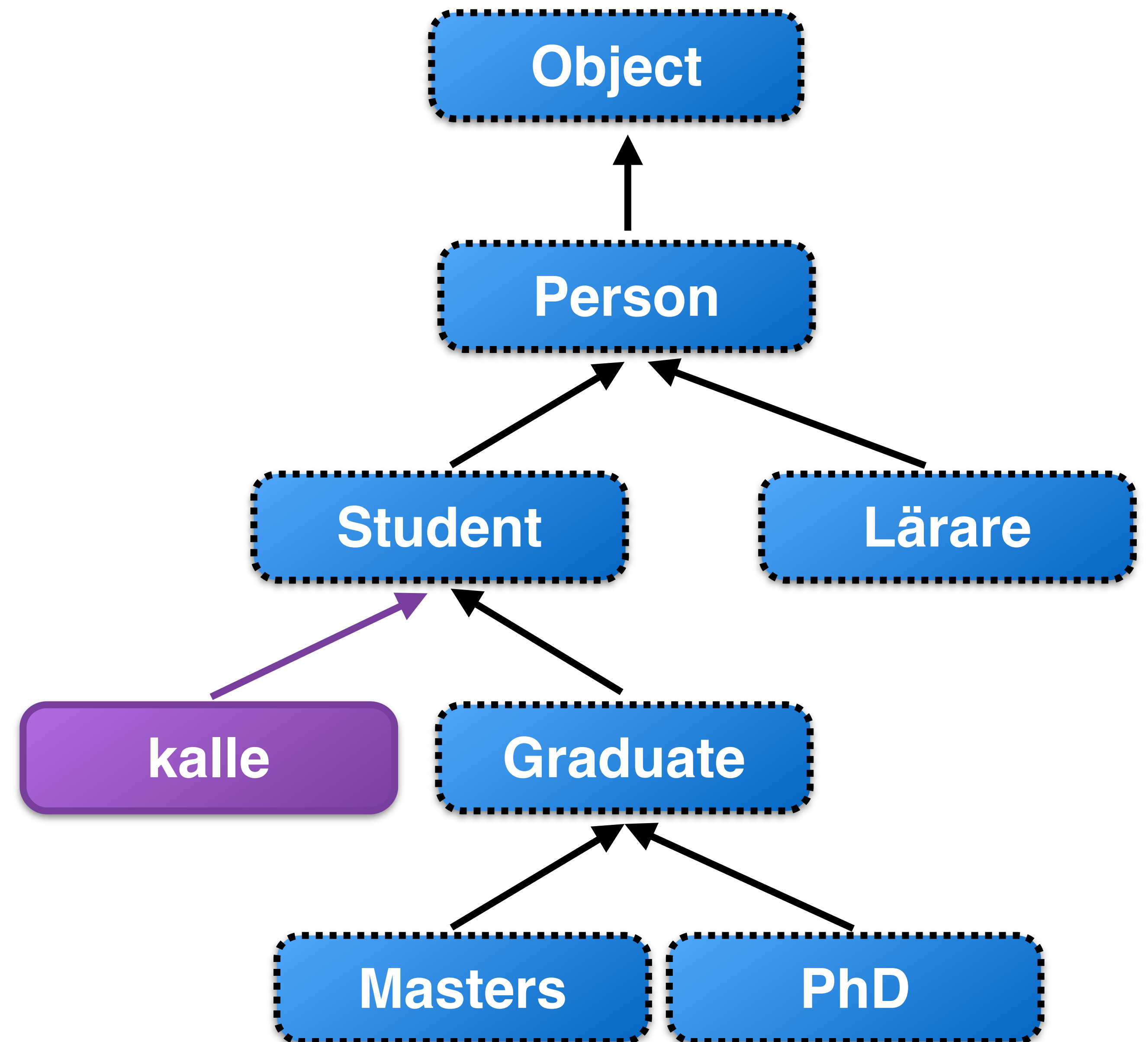
(static or dynamic type)

```
Person p;
String svar = < läs från användaren >
if (svar.equals("p") {
  p = new Person(...);
} else if (svar.equals("l") {
  p = new Lärare(...);
} else if (svar.equals("s") {
  p = new Student(...);
}
```

*Vad har p för typ här?*
(dvs vad kan vi göra med p?)

p har statiska typen **Person**
Det är allt kompilatorn kan se.

p's dynamiska typen är inte känd innan vi kör programmet, men det kommer att vara **Person**, **Lärare** eller **Student**.



# statisk eller dynamisk typ

(static or dynamic type)

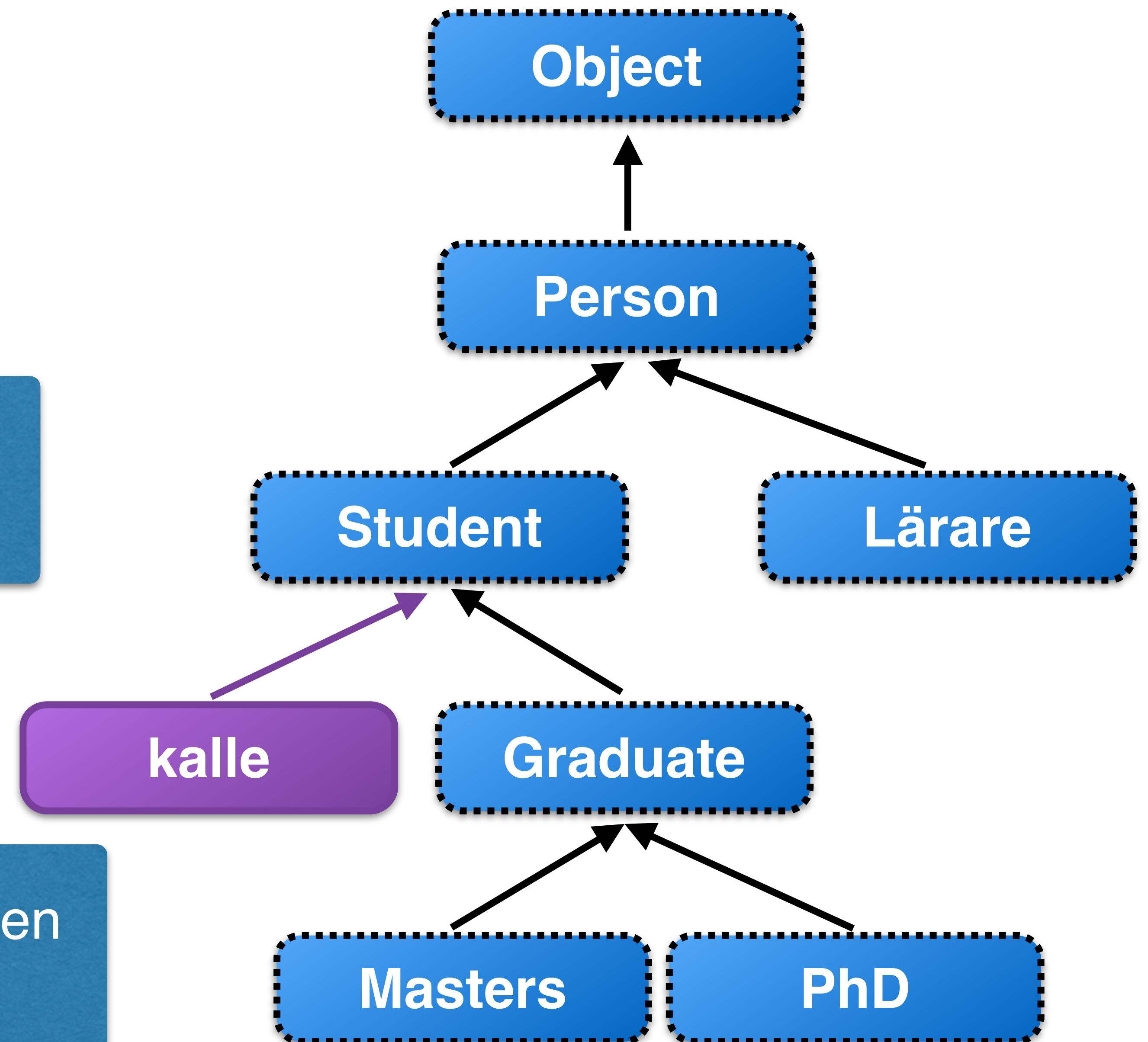
```
Person p;
String svar = < läs från användaren >
if (svar.equals("p") {
  p = new Person(...);
} else if (svar.equals("l") {
  p = new Lärare(...);
} else if (svar.equals("s") {
  p = new Student(...);
}

…
…
…

if (p instanceOf Student) {
  x = (Student p).getUniName();
}
```

vi kan kolla vad den dynamiska typen är när programmet kör

här konverterar vi (den statiska) typen neråt, från **Person** till **Student**



# statisk eller dynamisk typ

(static or dynamic type)

**statisk typ** = typen som kompilern ‘ser’

**dynamisk typ** = typen som objektet egentligen har när programmet körs
(beror på situationen)

**statisk typen** är alltid samma eller mera
abstrakt (högre upp) än den **dynamiskta typen**

# Typ kompabilitet 2 (Generisk metod)

```
class PersonDemo {
  public static void printAll (Object[] arr) {
    for(int i=0; i<arr.length; i++) {
      if ( arr[i] != null ) {
        System.out.println("arr[" + i + "] is " + arr[i].toString());
      }
    }
  }
  public static void main(String[]args){
    Person[] p = new Person[4];
    p[0] = new Person("joe",23);
    p[1] = new Student("becky",19,"Chalmer
    p[3] = new Teacher("bob",32,"Uppsala",1);
    printAll(p);
    if ( p[3] instanceof Teacher ) {
      ((Teacher) p[3]).raise(400);
    }
  }
}
```

vi glömde att sätta nånting i **p[2]**, men det fungerar för att vi kollar om objektet är null.

svar: **Person** är subklass av **Object** (alla klasser är det)

**Varför fungerar den här koden?**

# Typ kompabilitet…

Om vi har en **Teacher** med ett variabel "**salary**" hur funkar då:

```
public static boolean earnsMore (Person p1, Person p2) {
  return p1.getSalary() > p2.getSalary();
}
```

**Fel statisk typ.** Kompilern säger att det inte går.

**Men det här fungerar:**

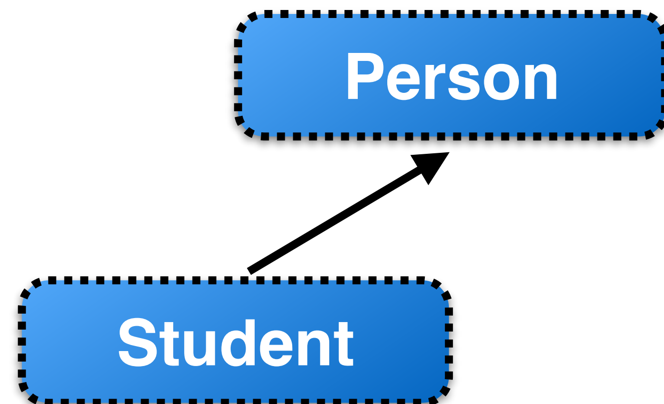
```
if (p1 instanceof Teacher && p2 instanceof Teacher) {
  return ((Teacher)p1).getSalary() > ((Teacher)p2).getSalary();
} else { vad här? }
```

**Ännu bättre:**

```
public static boolean earnsMore (Teacher p1, Teacher p2) {
  return p1.getSalary() > p2.getSalary();
}
```

# Sammanfattning av arv

base class = super class = parent class



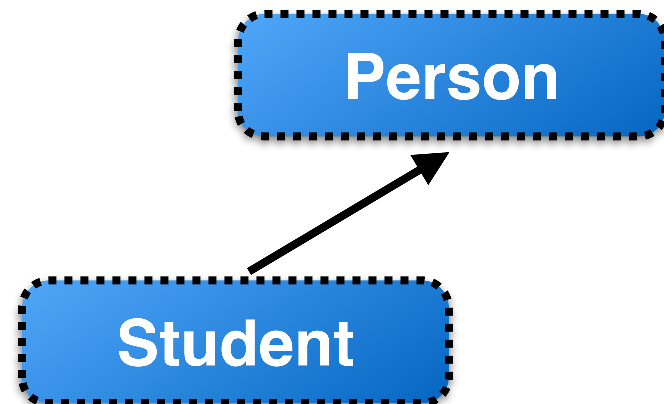

```
public class Person {
   …
}

public class Student extends Person {
  …
}
```

derived class = sub class = child class

- ”a sub class extends its super class”
- konstruktorer ärvs inte
- privata variabler ”ärvs” men kan inte kommas åt annat än via publika getters.
- privata metoder ärvs inte
- en sub klass kan ha nya metoder och variabler
- en sub klass kan även överskugga metoder i super klassen
- metoden x i superklassen kan kommas åt med super.x()
- konstruktorer i super klassen kan man komma åt med super(…)
- superkonstruktorn måste anropas först

# Kortare sammanfattning




```
public class Person {
   …
}

public class Student extends Person {
  …
}
```

är ett enkelt sätt att "***importera all kod***" från en annan klass, i detta fall Person

***Obs.*** man *kommer inte* åt saker som är **private** i Person.

**Fundera:** vad kan man *inte* implementera med arv?

# Frågor

**Fundera:** vad kan man ***inte*** implementera med arv?

Vad är en **superklass**?

Vad är en **subklass**?

Vad är skillnaden mellan **statisk** och **dynamisk typ**?

På vilket sätt **konverterar** man mellan dem?

Kan arv ***söndra*** superklasser?

# UML

Ni behöver inte kunna mycket UML

… men ni ska vet vad det är och ni ska kunna skissa enkla saker i UML.

# UML
## en notation för att beskriva relationer

Notation: [optional], Foo = nonterminal,
(Multi = kort för Multiplicity)

| Name |
|---|
| Fields |
| Methods |

Man kan beskriva fält och metoder med Java- eller UML syntax.

**Fält**:

J: [Visibility] [Type] Name [Multi] [=initialValue]

```
public int duration = 100
```

U: [Visibility] Name [Multi] [ :Type] [=initialValue]

```
+ duration : int = 100
```

**Metoder**:

Java: [Visibility] [Type] Name ([Parameter, ...])

```
int add(int dx, int dy)
```

UML: [Visibility] Name ([Parameter, ...]) [ :Type]

```
~add(dx : int, dy : int) : int
```



# Visibility

| Java | UML | tolkning |
|---|---|---|
| public | + | tillgänglig för alla |
| protected | # | i alla klasser i samma paket och alla subklasser |
| | ~ | alla i samma paket (package) |
| private | - | enbart inom klassen |

## Multiplicity

Betecknar antalet förekomster av fältet och är en komma separerad sekvens av heltalsintervall.

| | |
|---|---|
| l..u | från l till u inklusive, u kan vara * |
| i | ett heltal |
| * | 0, 1, 2, 3, ... |

## Parametrar

Java: Type Name

UML: Name : Type

# Exempel

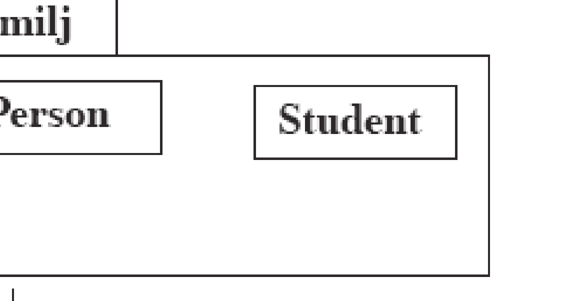
| Java | UML |
|---|---|
| Point | Point |
| private int x<br>private int y | - x : int<br>- y : int |
| public void move(int dx, int dy) | + move(dx:int, dy:int) |

Korta former

| Point |
|---|
| x<br>y |
| move() |

| Point |
|---|

Paket och objekt




| p1 : Point |
|---|
| x = 0<br>y = 0 |

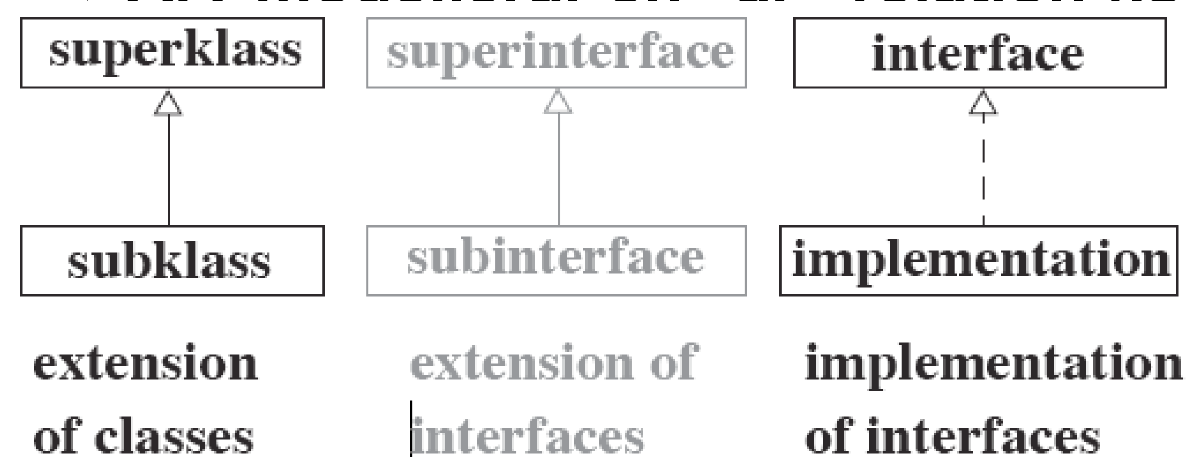
Ni behöver inte kunna så mycket UML men ni måste känna till relationerna, se nedan.



# Att modellera Relationer och Struktur

- **Inheritance** (arv) (en "är" relation) inklusive "extension" och "implementation"
- **Association** (association) (känner till / har) inkl. "aggregation" och "composition"
- **Dependency** (beroende)

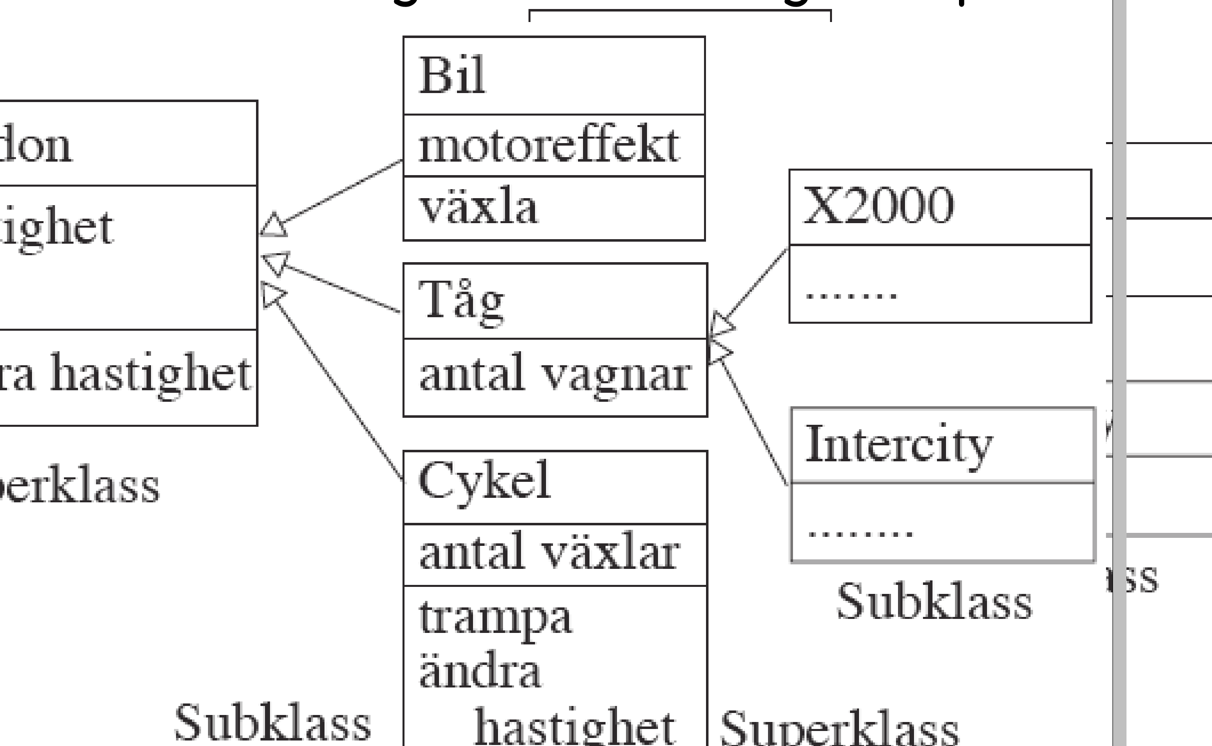
## Arv modellerar en "är"-relation (Is-a)




aggregate = förenande, sammanlagd, summa, förena
aggregation = sammanhopning, samling
association = förbindelse, umgänge, samband,anknytning
composition = (musik)komposition, sammansättning, bildande

# Arv modellerar en "är" - relation

Beskriver allmänna gemensamma egenskaper




Subklassen kan vara:

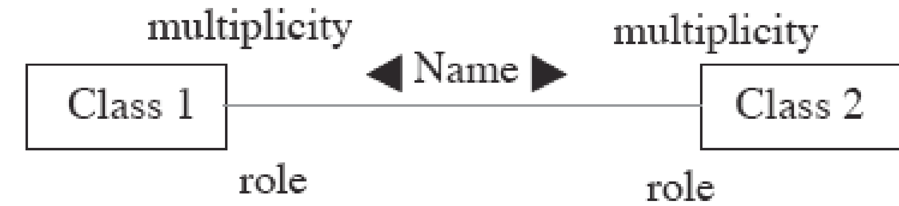
- en utökning av superklassen
  dvs man lägger till
  egenskaper (instansvariabler) och
  beteende (instansmetoder)
- en specialisering av superklassen
  dvs vissa metoder ersätts

Implementeras med Java konstruktionen arv

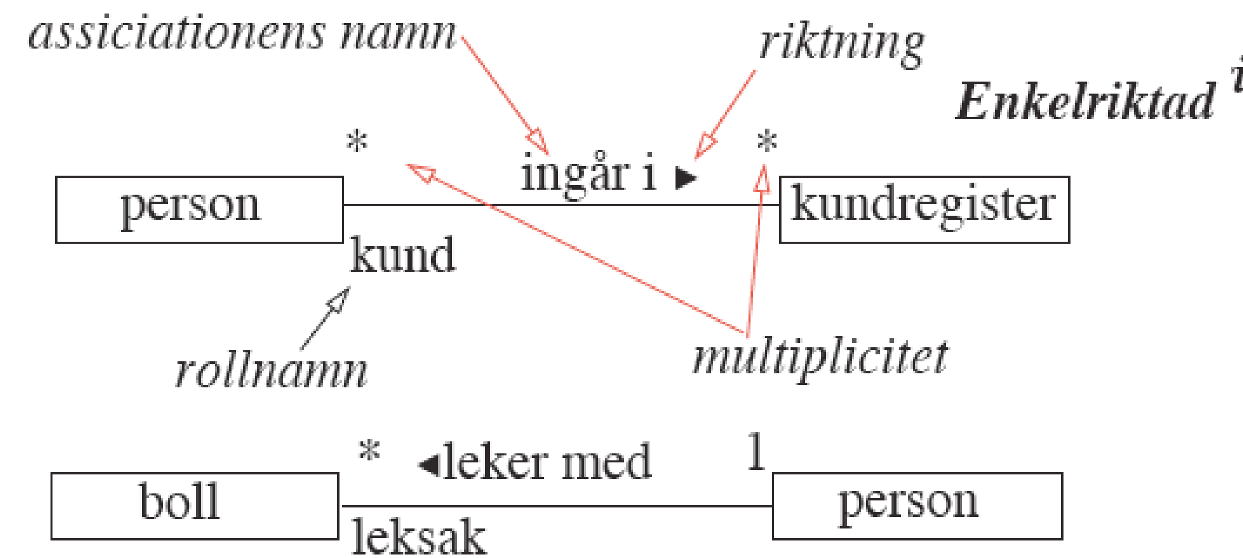


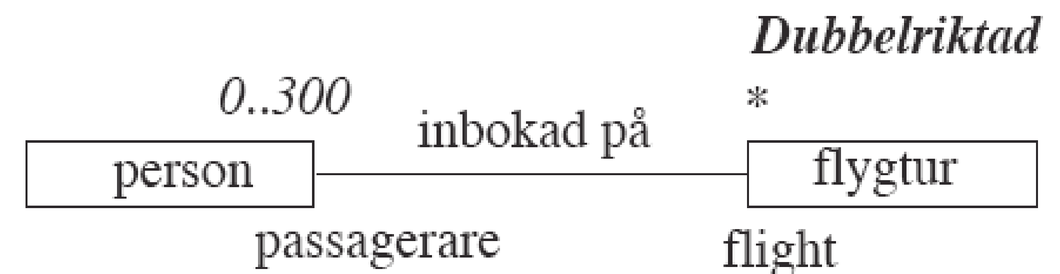
# Association modellerar “känner till”

En binär “känner till” relation mellan klasser.



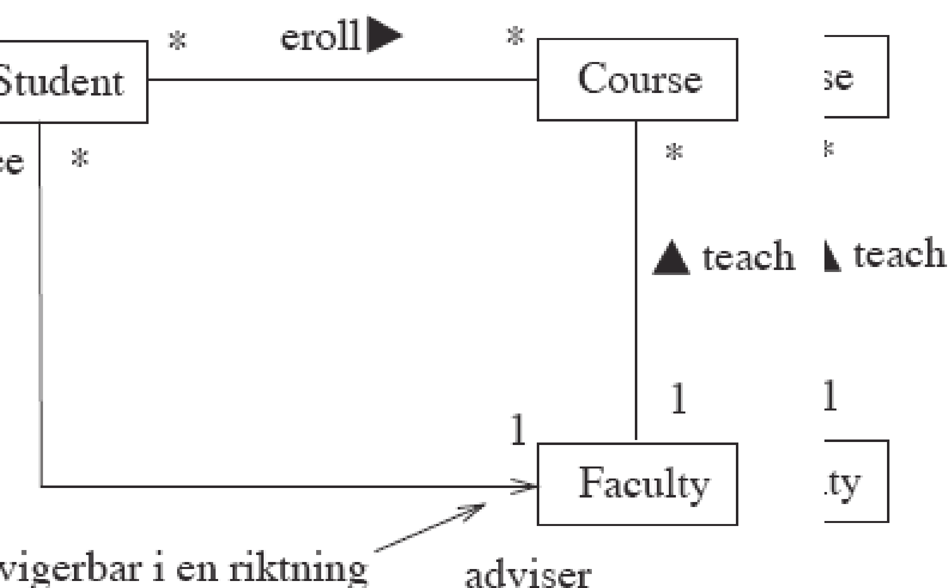

2 objekt känner till varandra:




Implementeras med referenser
dvs ett objekt har en pekare till andra objekt

## många->många och många->en




- En student deltar (enrolls) i en kurs
- en student kan ta många kurser
- en kurs kan ha många studenter
- en lärare kan ha många kurser
- en kurs har bara en lärare (nåja)

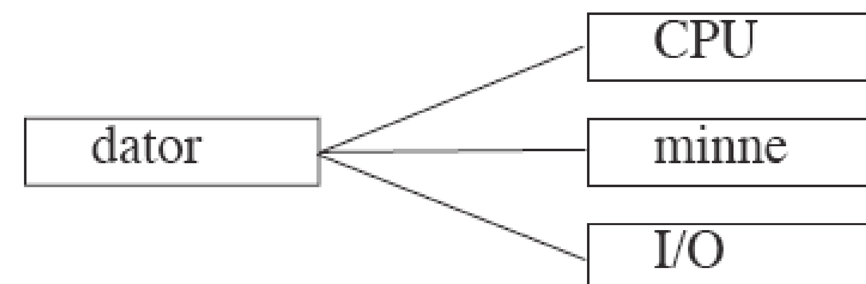


## Aggregation och Composition - en "har" (has-a) relation

**Aggregation** är en speciell form av association som representerar en "har" relation. Betecknas med en ◇ i aggregatändan av relationen.

**Composition** är en starkare form av aggregation där komponenterna är helt beroende (ägda) av aggregatet. Betecknas med en ◆ i aggregatändan av relationen.

Ett objekt är en del av ett annat objekt. Ägaren har exklusiv tillgång till de inneslutna objekten.




Implementeras med referenser dvs motorn innehåller en instansvariabel som är en pekare till en cylinder.

# Dependency (beror av)

Om man tex vill kunna hämta information från ladok så måste man känna till strukturen hos kurser och studenter, man är "beroende"

---

Man kan också modellera dynamiskt beteende i UML med sekvensdiagram och tillståndsdiagram.

---

Det finns grafiska verktyg för att rita UML diagram tex argoUML

http://argouml.tigris.org/

# Mycket nya glosor idag - igen!

arv, inheritance,

trädliknande arvstrukturer

förälderklass ("superklass") , barnklass ("subklass")

base class = super class = parent class

derived class = sub class = child class

multipelt arv, multipelt implementationsarv

extends

överskuggning (override), , överlagring (overloading)

parameterprofil

metodbindning

final, super, this

typkompabilitet, typkonvertering

instanceof

statisk / dynamisk typ

Uniform Modeling Language UML

visibility, multiplicity

protected,

relationer: arv, association, aggregation, composition

referenser, implementeras