TDA 548: Grundläggande Programvaruutveckling

Föreläsning 2, vecka 7:

Arv, subtyper och Javas API

Magnus Myréen

Chalmers, läsperiod 1, 2016-2017

Nästa vecka

Inget nytt material

… men vad ska vi göra?

Skicka förslag så kanske vi gör det du vill att vi gör!

Idag kommer det ännu nytt material:

- `arv` (`extends`)
- `Object`
- `instanceof`
- lite UML
- abstrakta klasser dvs `abstract`

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 kod från en annan klass.

Man slipper att skriva samma kod flera gånger.

… och dessutom vet Java att klasserna är relaterade.
(dvs typ kompatibla)

Varning! Varning! Varning!

Vissa tycker att arv (implementationsarv) leder till dålig kod. Jag håller delvis med.

JAVA TOOLBOX

By Allen Holub

HOW-TO

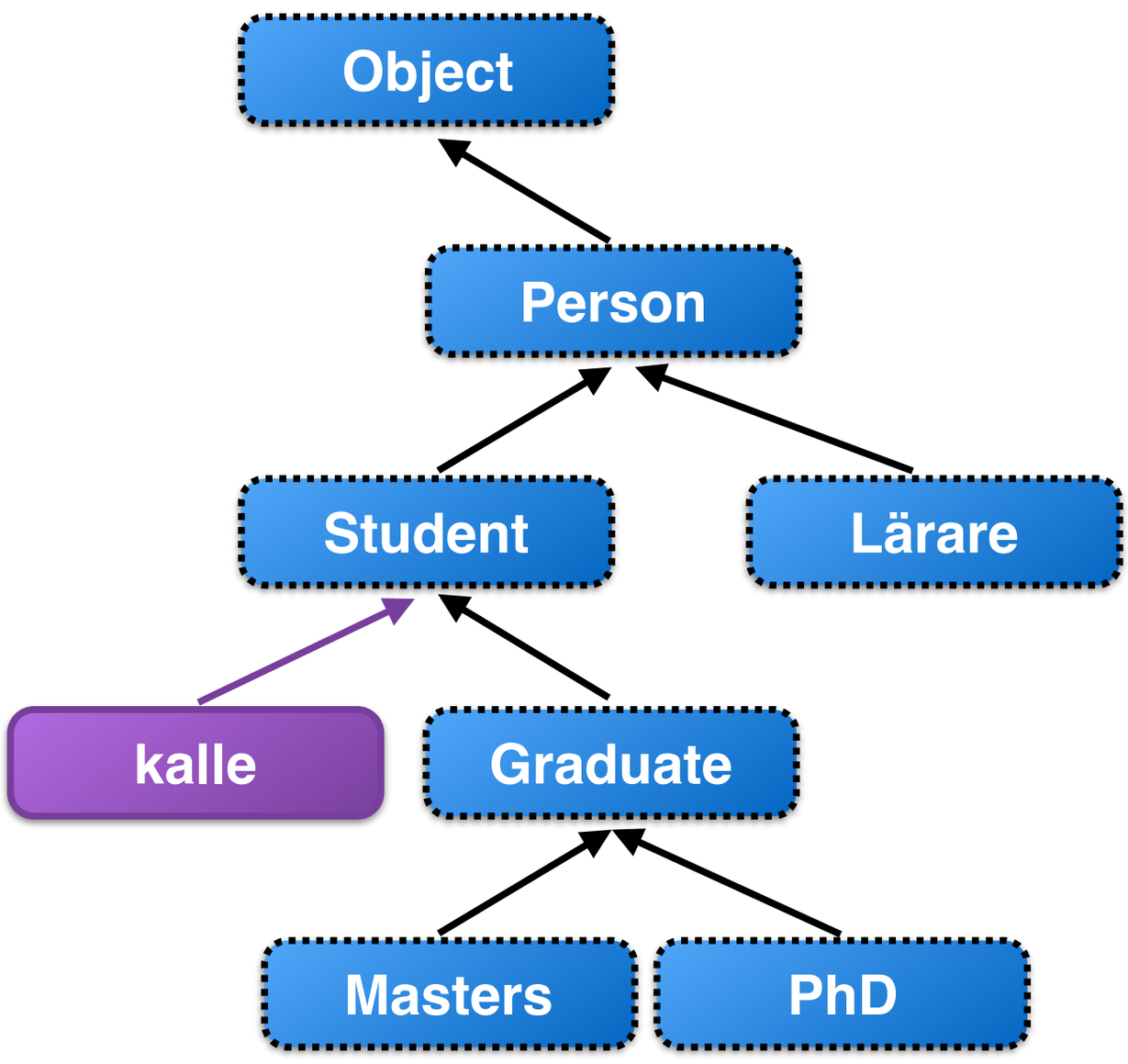
Why extends is evil

Improve your code by replacing concrete base classes with interfaces

http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html

Det är oftast bättre att använda **interface**s, om det går.
Nästa kurs handlar mera om design i OO.

Arv (inheritance)



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

En **Student** är en "specialiserad" **Person**.

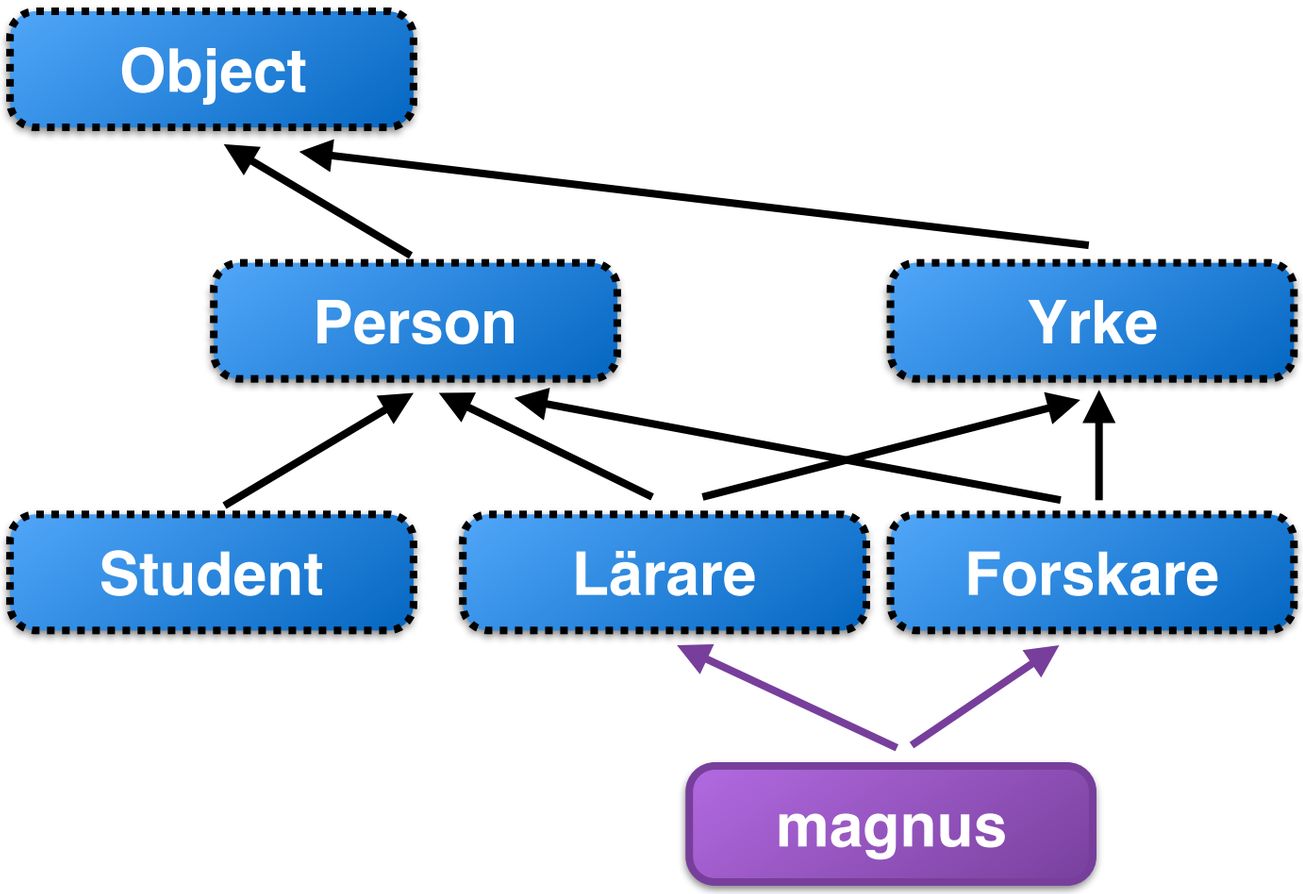
Klasser organiseras i trädliknande strukturer.

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.

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

Multipelt arv finns i verkligheten

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



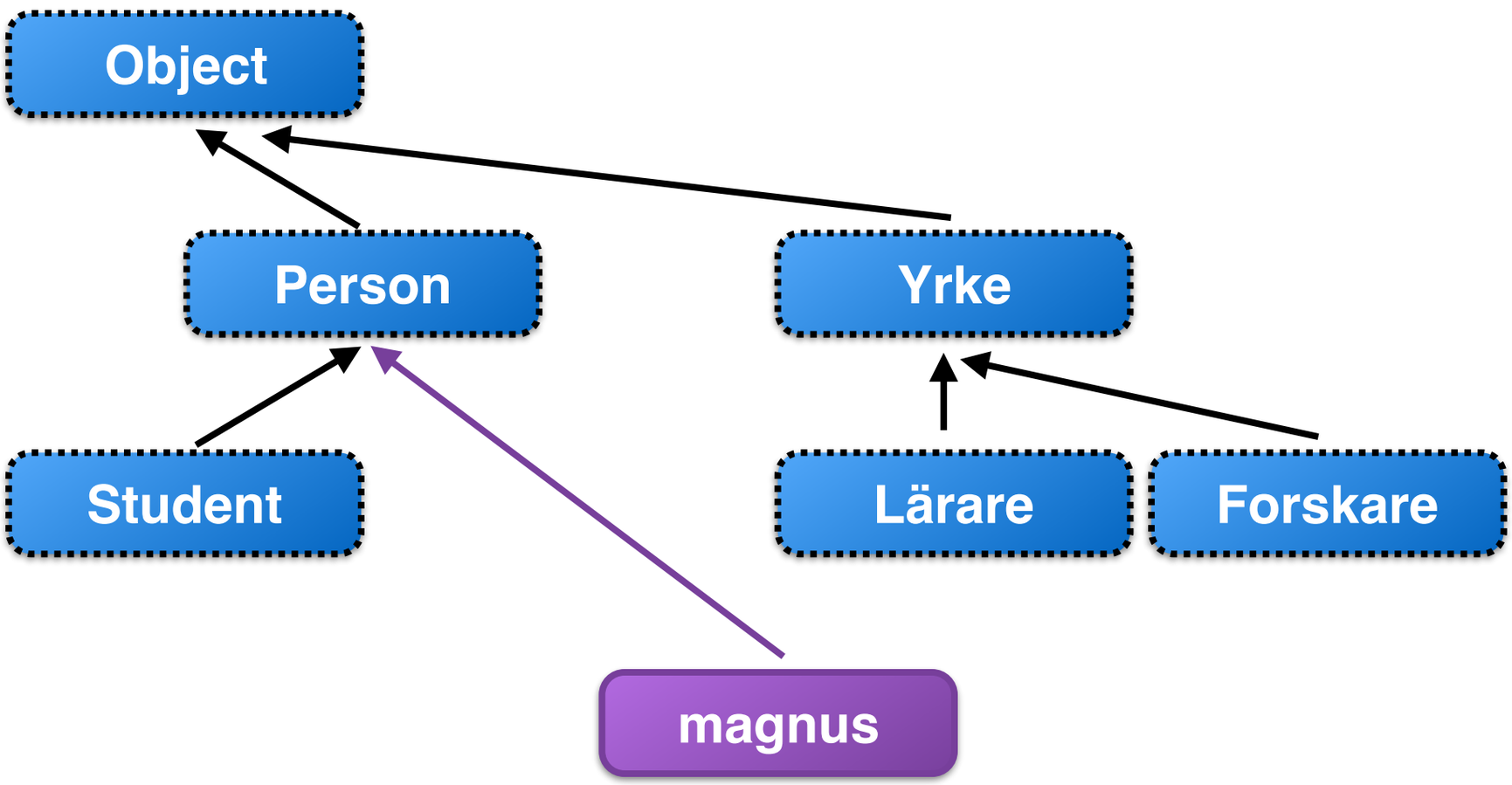
Hur hitta lämpliga abstraktioner, klasser?

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

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.

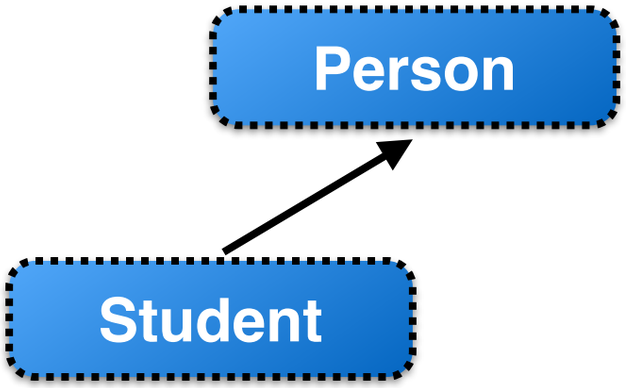
Överskuggning (Override)

(jämför: överlagring = overloading)

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

Dock inte ta bort saker.

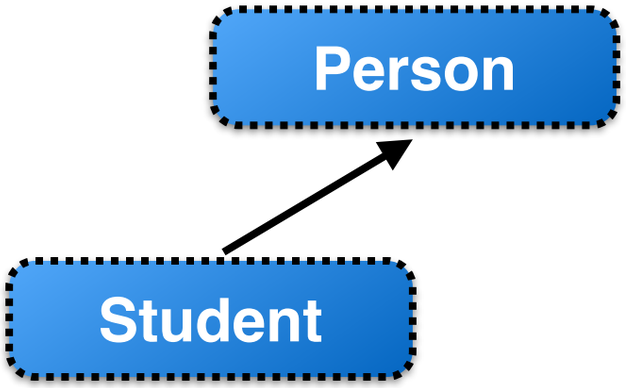
Undantag eller ändrat beteende hanteras genom att en subklass överskuggar sitt arv från superklassen genom att ha en metod med samma namn och samma parameterprofil (signatur).



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;
  }
}
```

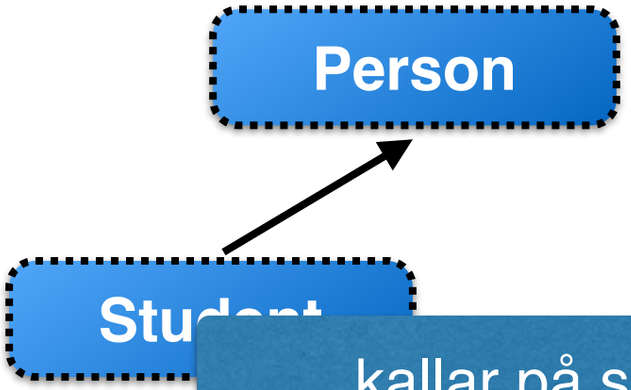
```
public class Student extends Person {
  private String uniName;
  public Student(String n, int age,
                 String uni) {
    super(n, age);
    uniName = uni;
  }
  public String toString() {
    return super.toString() +
             " student at " + uniName;
  }
  public String getUniName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

kallar på superklassens version av metoden **toString**

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

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

```
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;
    }

}
```

```
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;
    }

}
```

```
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());
    }

}
```

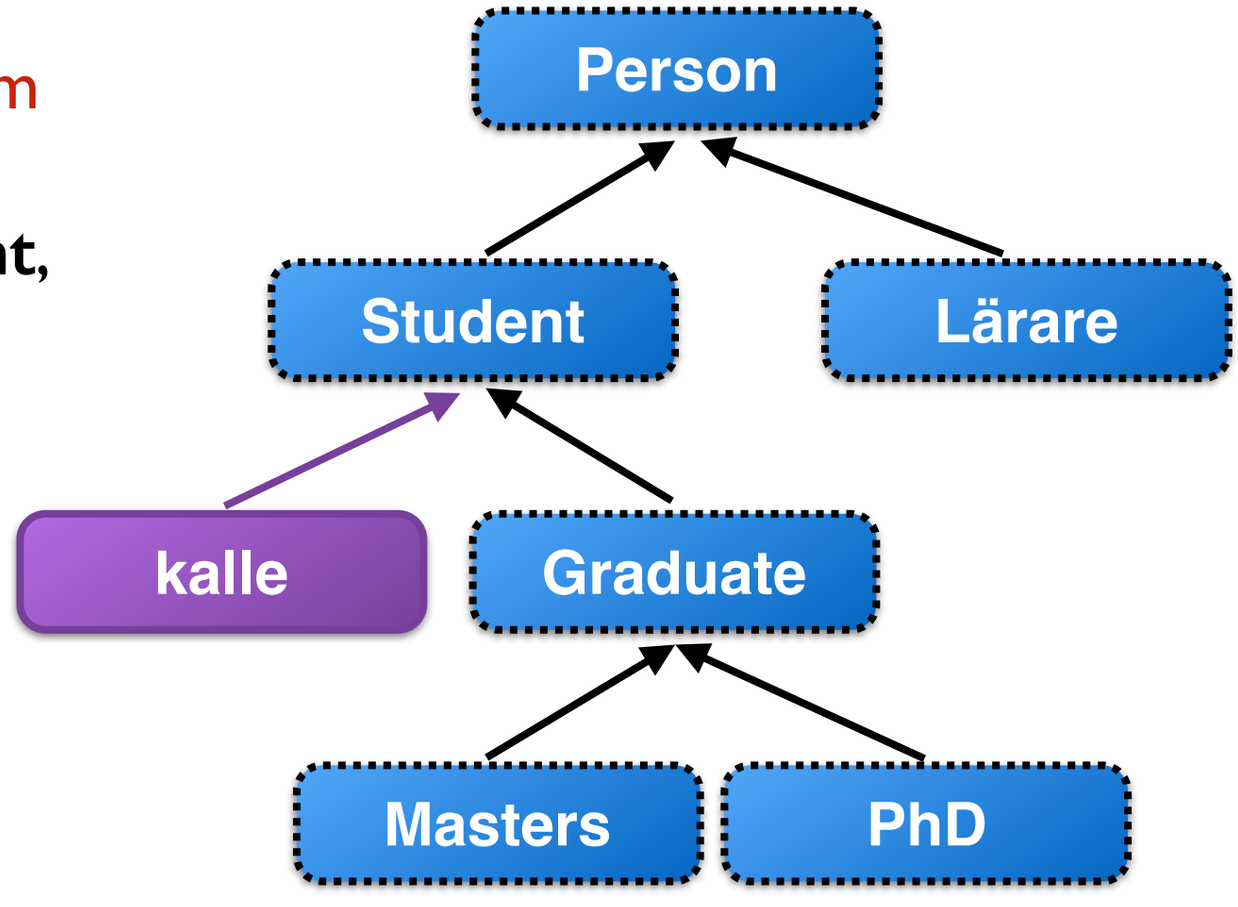
Typ kompabilitet 1

(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

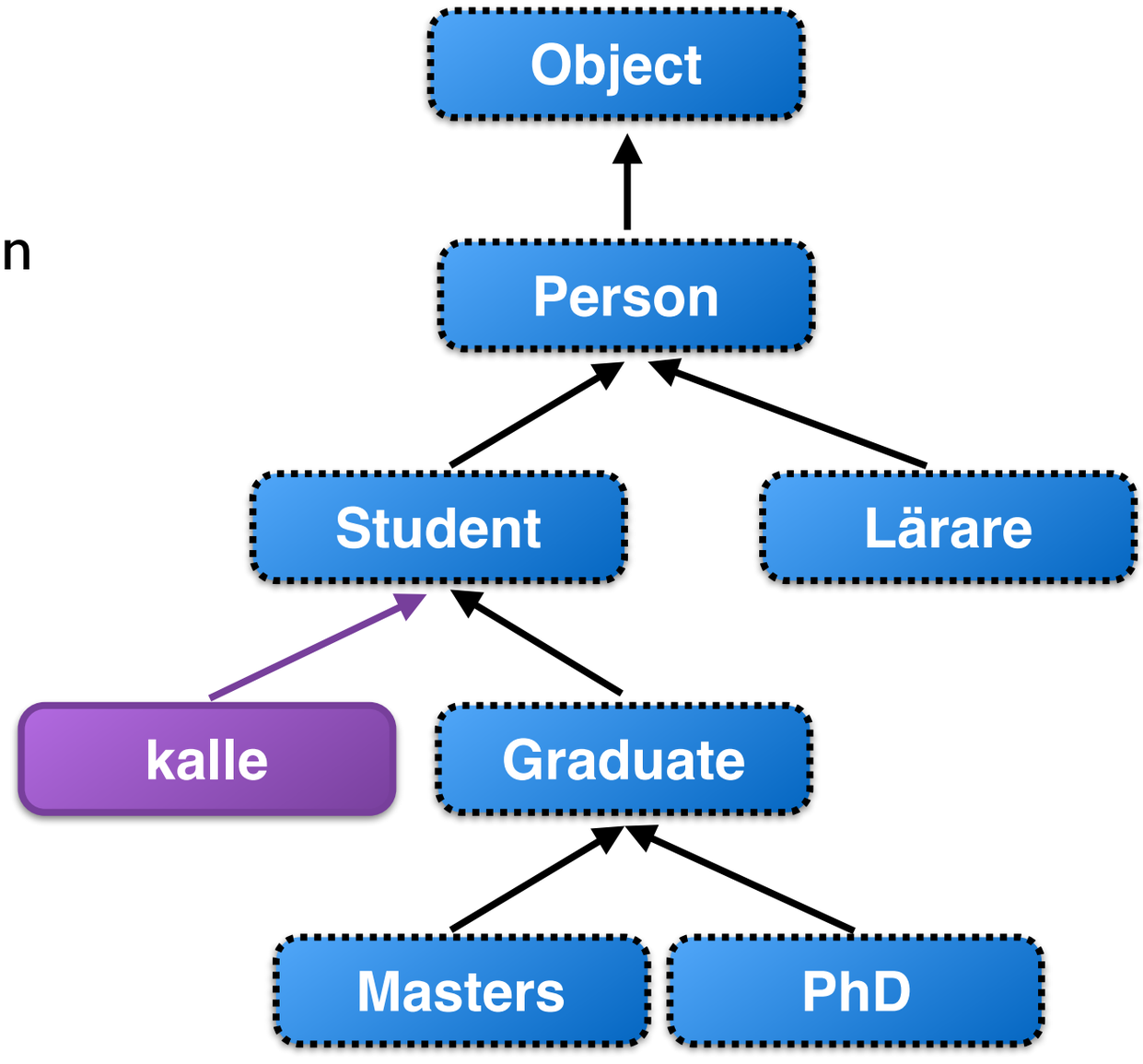
(instans av)

`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.

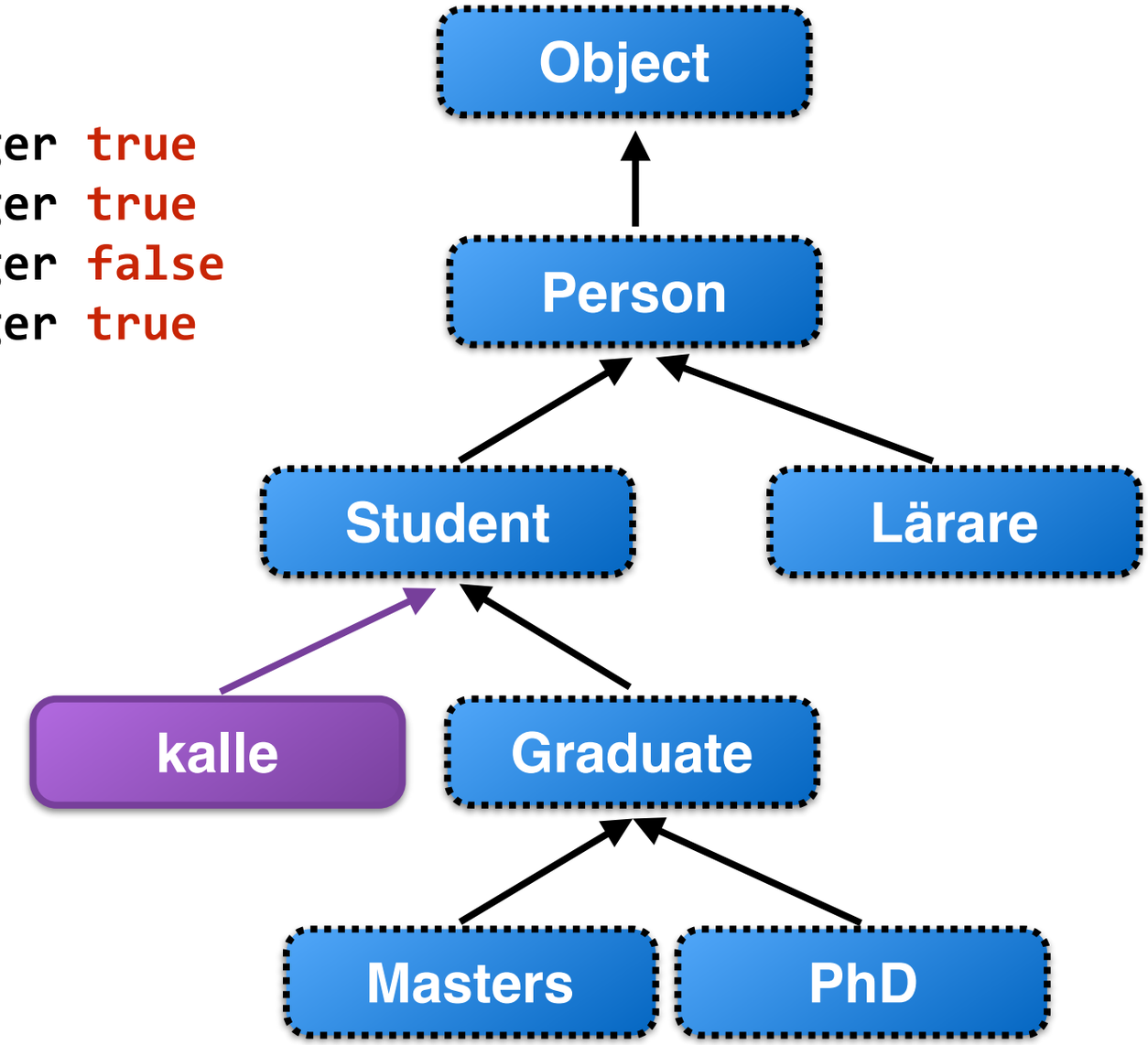


instanceof

(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

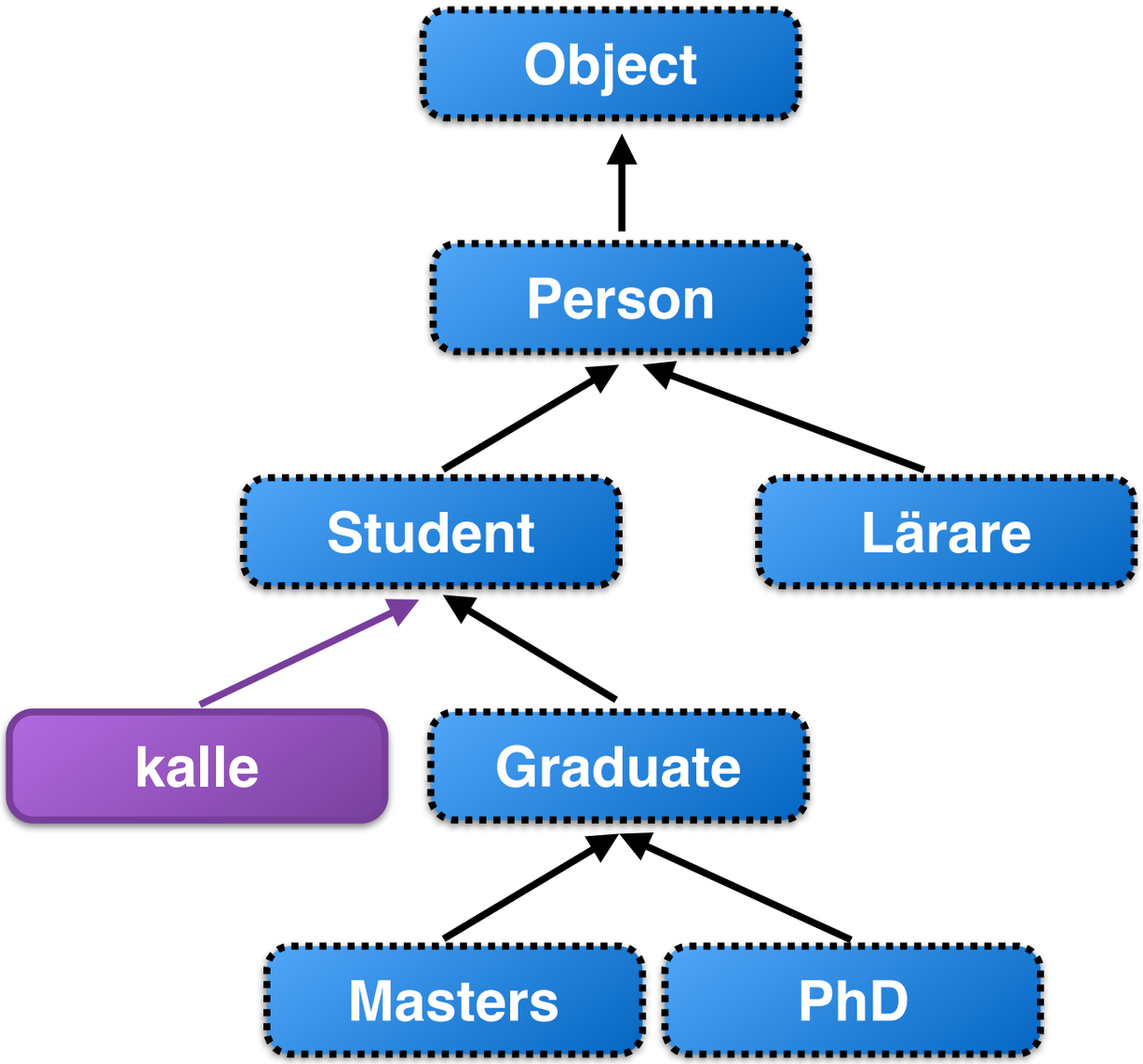
(static or dynamic type)

```
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)

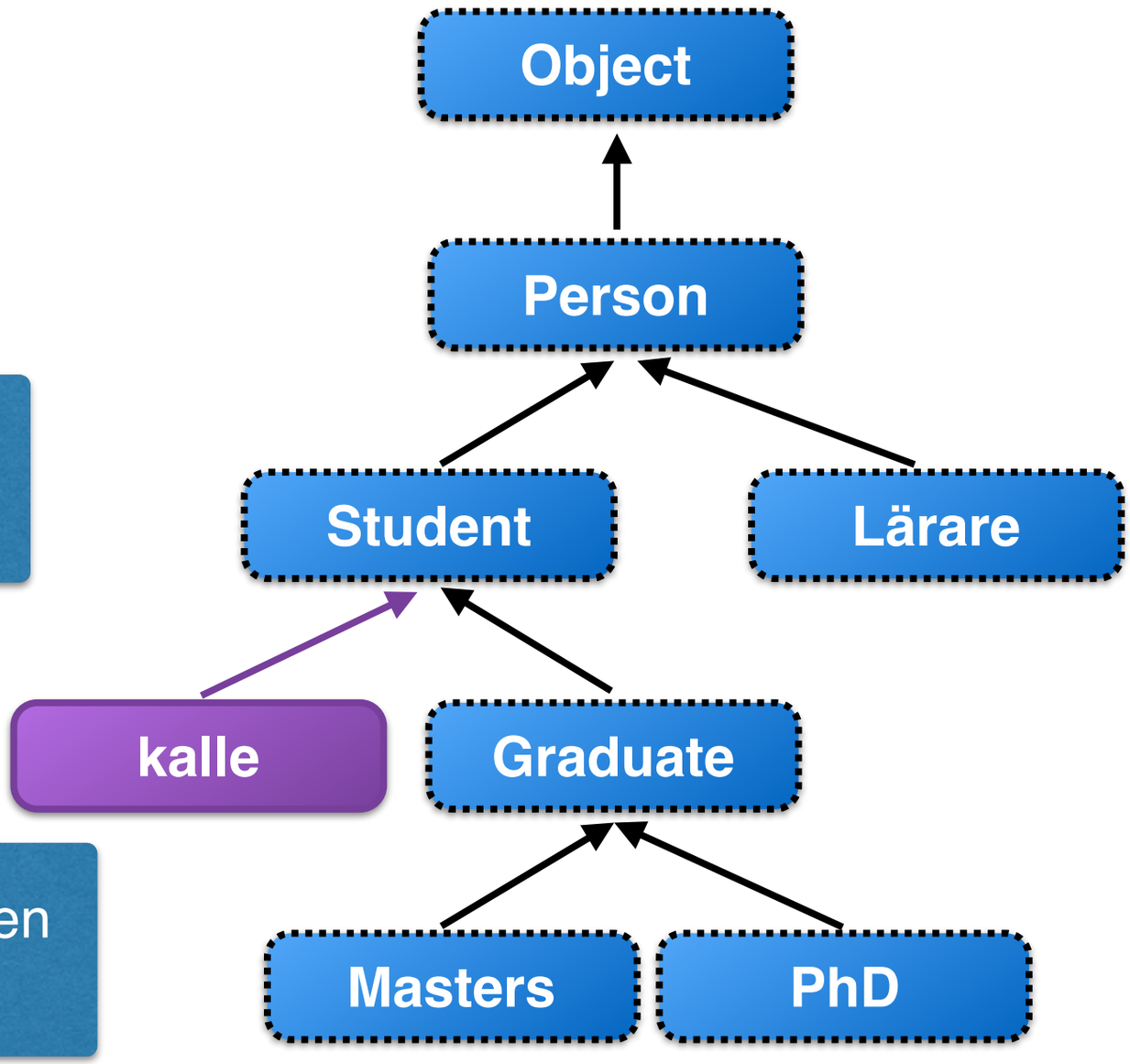
```
Person p;
String svar = < läs från användaren >
if (svar.equals(“p”) {
   p = new Person(...);
} else if (svar.equals(“l”) {
   p = new Lärare(...);
} else if (svar.equals(“s”) {
   p = new Student(...);
}

…
…
…

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:

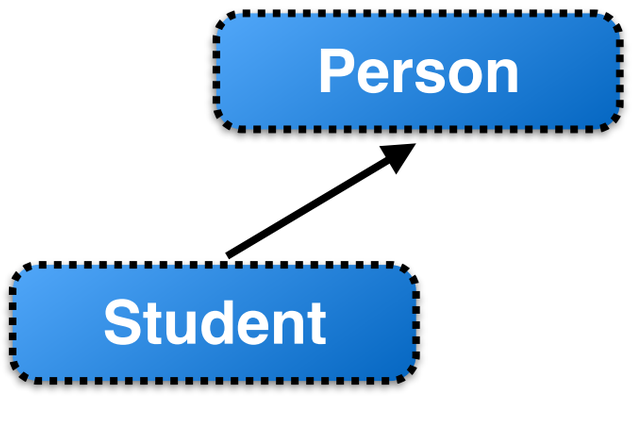
```
if (p1 instanceof Teacher && p2 instanceof Teacher) {
  return ((Teacher)p1).getSalary() > ((Teacher)p2).getSalary();
} else { vad här? }
```

Ännu bättre:

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

Sammanfattning av arv

base class = super class = parent class



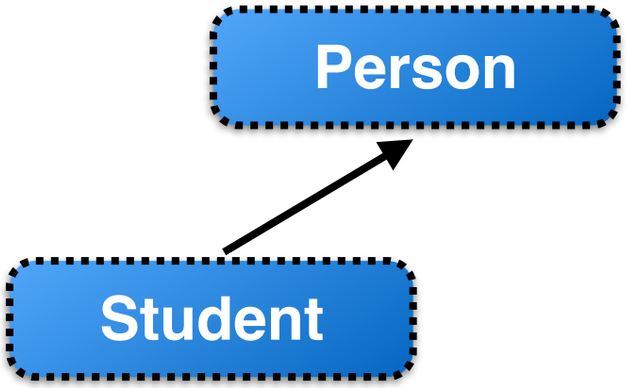

```
public class Person {
  …
}

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)

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

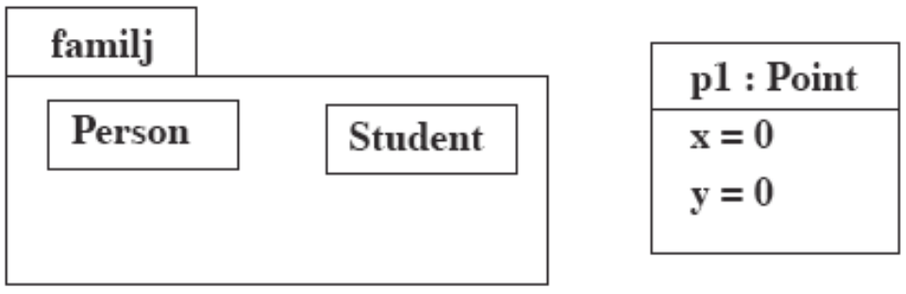
UML: Name : Type

Exempel

Java	UML
Point	**Point**
private int x **private int y**	**- x : int** **- y : int**
public void move(int dx, int dy)	**+ move(dx:int, dy:int)**

Korta former

Paket och objekt

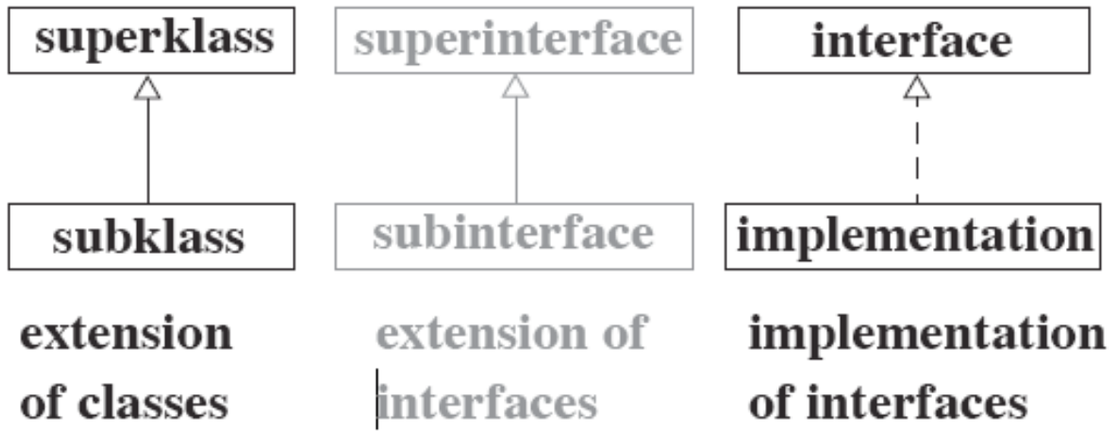


Ni behöver inte kunna så mycket UML men ni måste känna till relationerna, se nedan.

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

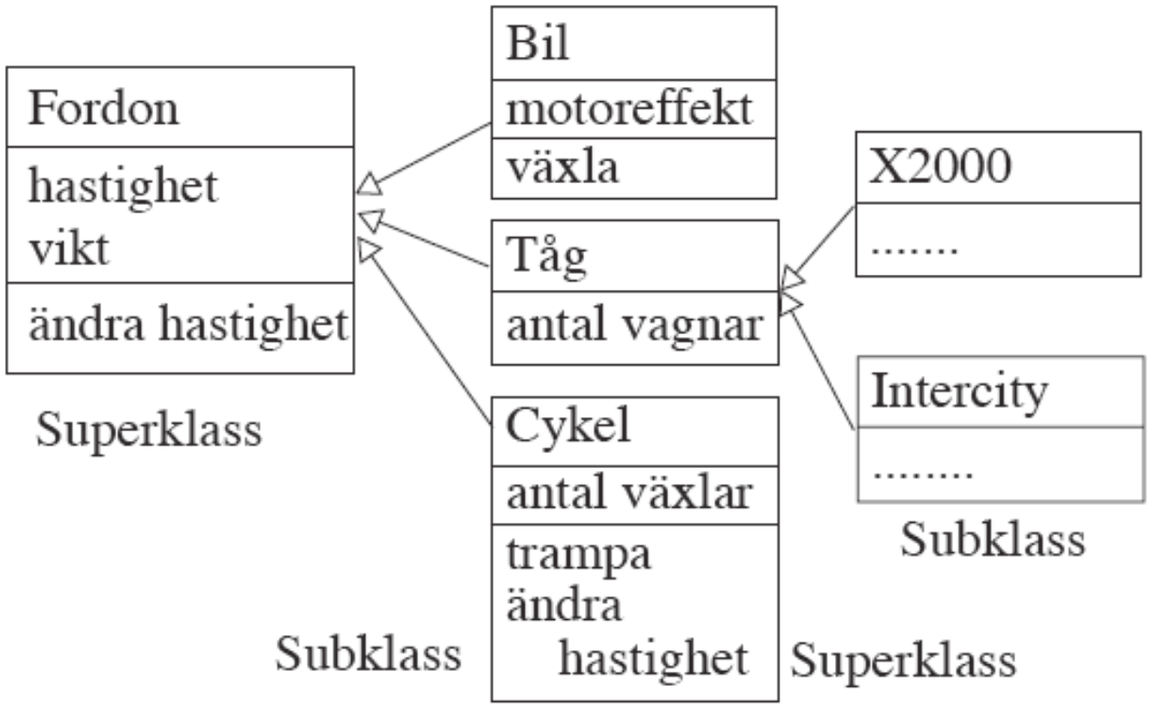
aggregation = sammanhopning, samling

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)

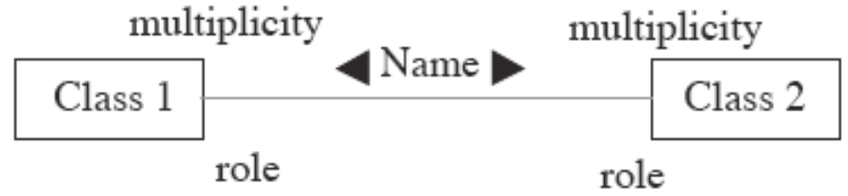
- en specialisering av superklassen

  dvs vissa metoder ersätts

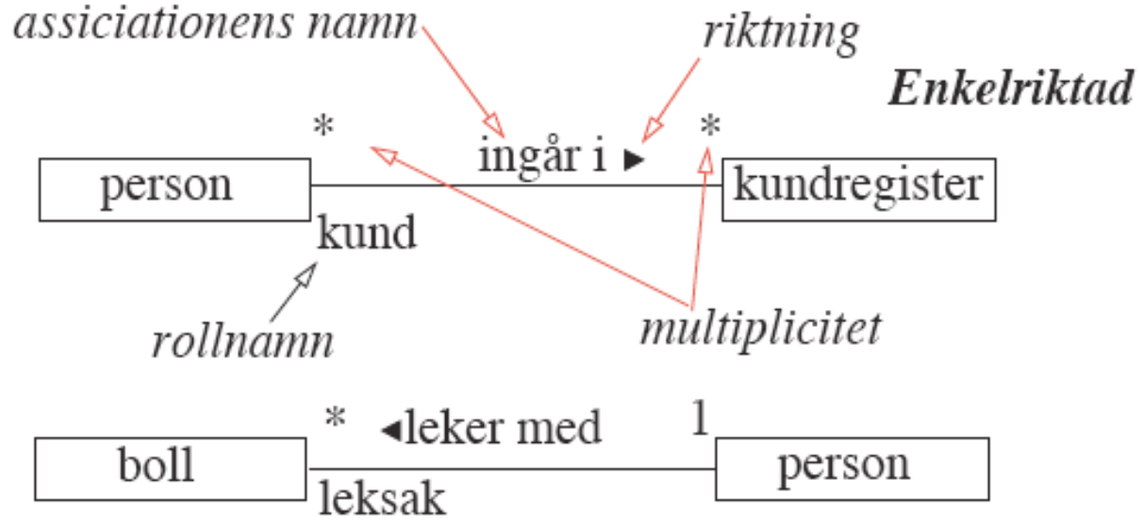
Implementeras med Java konstruktionen arv

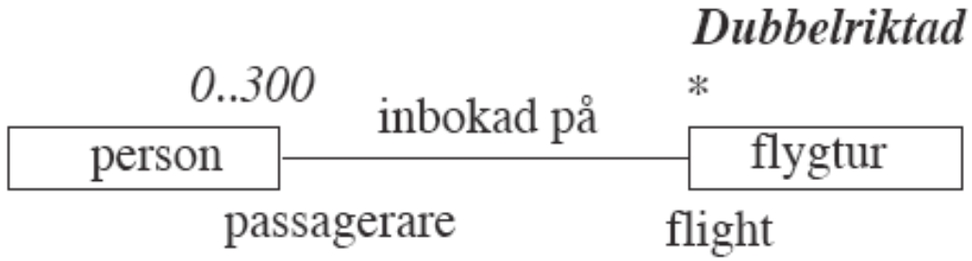
Association modellerar "känner till"

En binär "känner till" relation mellan klasser.



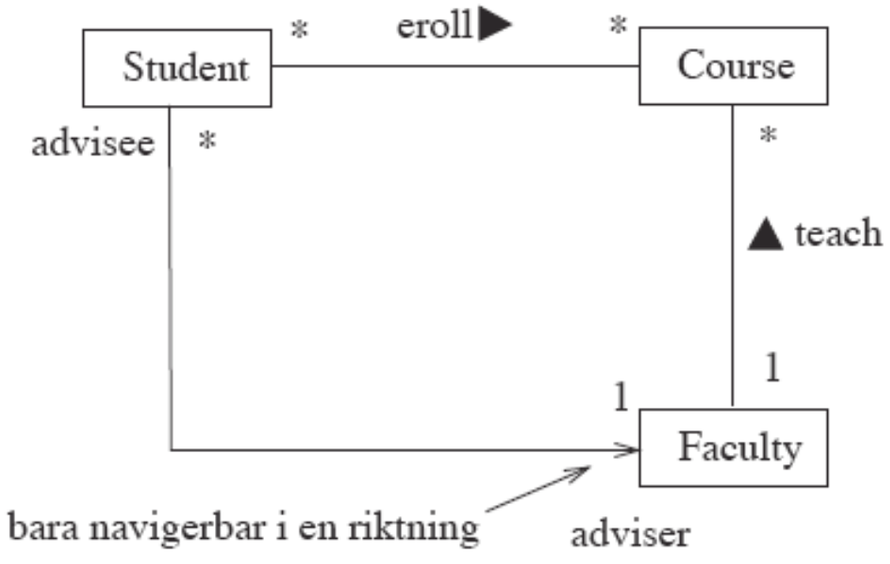
2 objekt känner till varandra:





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

många-> många och många -> en

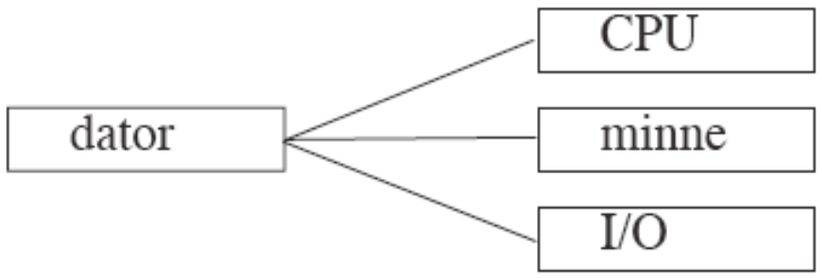


- En student deltar (enrolls) i en kurs
- en student kan ta många kurser
- en kurs kan ha många studenter
- en lärare kan ha många kurser
- en kurs har bara en lärare (nåja)
-

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 här del i 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/

Underhållning

https://www.youtube.com/watch?v=RnqAXuLZIaE

Påminnelse

klass- och instansvariabler

Klass- och instansvariabler/metoder

Klassvariabler — static — hör till klassen.

Instansvariabler — icke-static — hör till instanser av klassen (dvs objekt).

I koden:

```
public class PoliceVolvo {
  public static final int topSpeed = 150;
  private int speed;
  public void setSpeed (int newSpeed) {
    speed = newSpeed;
  }
  public int getSpeed () {
    return speed;
  }
  public static int getTopSpeed() {
    return topSpeed;
  }
}
```

Hur man använder det:

var som helst, kan man använda klassens variabler och metoder

```
PoliceVolvo.topSpeed
PoliceVolvo.getTopSpeed()
```

endast en kopia finns globalt,

men instans variabler och metoder finns bara inne i instanser av klassen

```
PoliceVolvo v = new PoliceVolov();
v.setSpeed(50);
```

det finns en kopia av instans variablerna/metoderna i varje instans

Klass- och instansvariabler/metoder

Klassvariabler — static — hör till klassen.

Instansvariabler — icke-static — hör till instanser av klassen (dvs objekt).

I koden:

```
public class PoliceVolvo {
  public static final int topSpeed = 150;
  private int speed;
  public void setSpeed (int newSpeed) {
    speed = newSpeed;
  }
  public int getSpeed () {
    return speed;
  }
  public static int getTopSpeed() {
    return topSpeed;
  }
}
```

här skapas en ny instans

då kan vi använda instansmetoder

Hur man använder det:

var som helst, kan man använda klassens variabler och metoder

```
PoliceVolvo.topSpeed
PoliceVolvo.getTopSpeed()
```

endast en kopia finns globalt,

men instans variabler och metoder finns bara inne i instanser av klassen

```
PoliceVolvo v = new PoliceVolov();
v.setSpeed(50);
```

det finns en kopia av instans variablerna/metoderna i varje instans

Klass- och instansvariabler/metoder

Klassvariabler — static — hör till klassen.

Instansvariabler — icke-static — hör till instanser av klassen (dvs objekt).

man kan använda klassvariabler/metoder i instans metoder, men ***inte*** tvärtom

I koden:

```
public class PoliceVolvo {
  public static final int topSpeed = 150;
  private int speed;
  public void setSpeed (int newSpeed) {
    if (newSpeed <= topSpeed) {
      speed = newSpeed;
    }
  }
  public int getSpeed () {
    return speed;
  }
  public static int getTopSpeed() {
    return topS
  }
}
```

här skapas en ny instans

då kan vi använda instansmetoder

vänder det:

var som helst, kan man använda klassens variabler och metoder

```
PoliceVolvo.topSpeed
PoliceVolvo.getTopSpeed()
```

endast en kopia finns globalt,

men instans variabler och metoder finns bara inne i instanser av klassen

```
PoliceVolvo v = new PoliceVolov();
v.setSpeed(50);
```

det finns en kopia av instans variablerna/metoderna i varje instans

Klass- och instansvariabler/metoder

Klassvariabler — static — hör till klassen.

Instansvariabler — icke-static — hör till instanser av klassen (dvs objekt).

man kan använda klassvariabler/metoder i instans metoder, men ***inte*** tvärtom

varför kan man inte använda instansvariabler i en klassmetod, dvs i en static metod?

Svar: kanske det inte finns instanser… Ifall det finns flera instanser, vilken bör användas? **Oklart!** Därför *förbjuder* Java klassmetoder från att använda instansvariabler.

I koden:

```
public class PoliceVolvo {
  public static final int topSpeed = 100;
  private int speed;
  public void setSpeed (int newSpeed) {
    if (newSpeed <= topSpeed)
      speed = newSpeed;
    }
  }
  public int getSpeed () {
    return speed;
  }
  public static int getTopSpeed() {
    return topSpeed;
  }
}
```

här skapas en ny instans

då kan vi använda instansmetoder

```
PoliceVolvo v = new PoliceVolov();
v.setSpeed(50);
```

det finns en kopia av instans variablerna/metoderna i varje instans

Klass- och instansvariabler/metoder

Kom ihåg:

Klassvariabler — static — hör till klassen.

Instansvariabler — icke-static — hör till instanser av klassen (dvs objekt).

var som helst, kan man använda klassens variabler och metoder

```
PoliceVolvo.topSpeed
PoliceVolvo.getTopSpeed()
```

endast en kopia finns globalt,

men instans variabler och metoder finns bara inne i instanser av klassen

```
PoliceVolvo v = new PoliceVolov();
v.setSpeed(50);
```

det finns en kopia av instans variablerna/metoderna i varje instans

Arv och klassvariabler

Hmm … en fråga: Hur fungerar klassvariabler vid arv?

Hur hittar man svaret? Skriv ett test program!

```
public class A {
  public static int i = 1;
  public static int j = 2;
}
public class B extends A {
  public static int i = 3;
}
```

```
public class Test {
  public static void main(String[] args) {
    System.out.println("A.i = " + A.i);
    System.out.println("A.j = " + A.j);
    System.out.println("B.i = " + B.i);
    System.out.println("B.j = " + B.j);
    B.i = 7;
    B.j = 8;
    System.out.println("A.i = " + A.i);
    System.out.println("A.j = " + A.j);
    System.out.println("B.i = " + B.i);
    System.out.println("B.j = " + B.j);
  }
}
```

Utskrift:

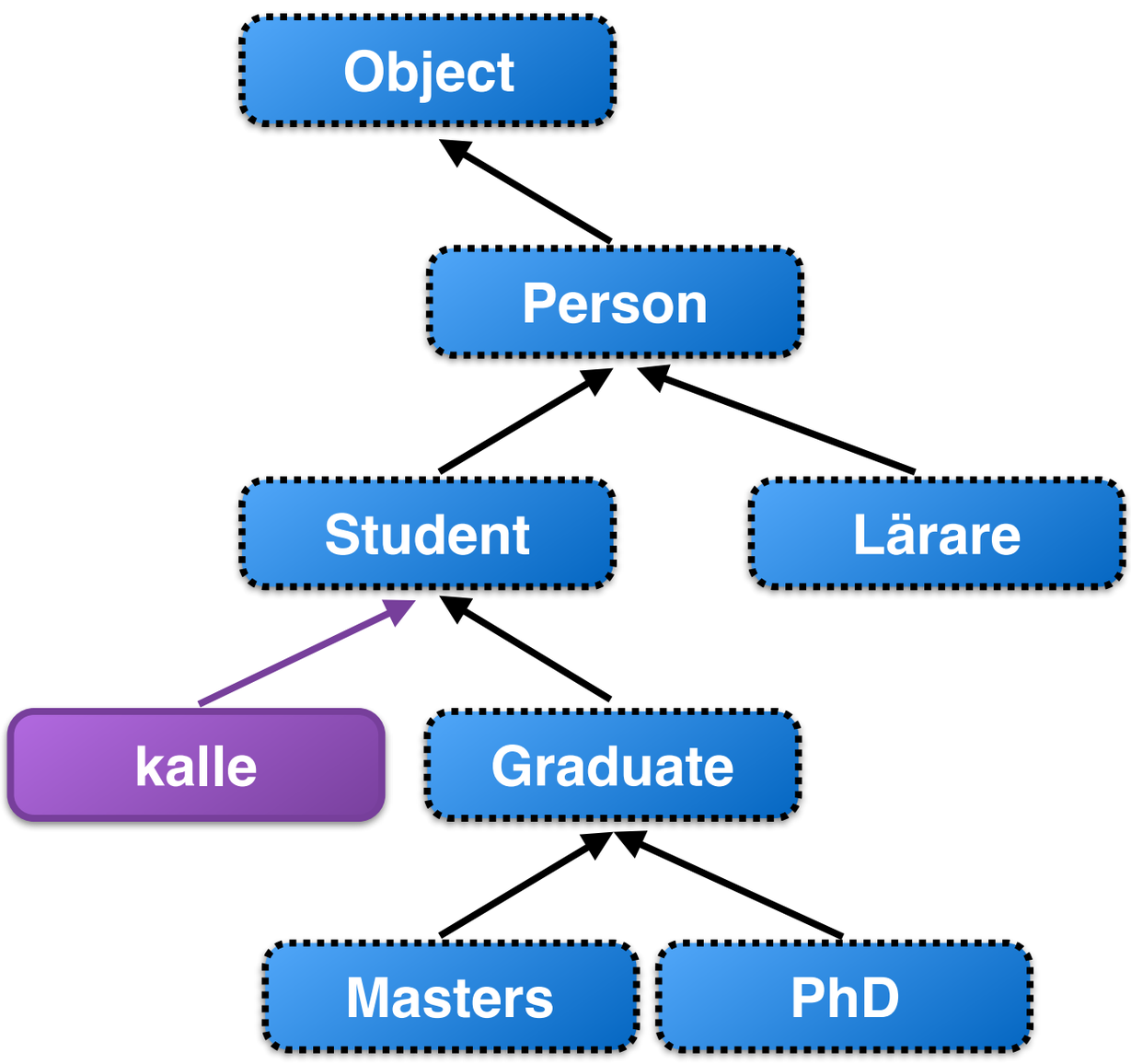
```
A.i = 1
A.j = 2
B.i = 3
B.j = 2
A.i = 1
A.j = 8
B.i = 7
B.j = 8
```

Svar: Den överskuggade variabeln `A.i` är separat från `B.i`.
Men den ärvde variabeln `B.j` är samma som `A.j`.

Nästa koncept:

abstrakta klasser dvs `abstract`

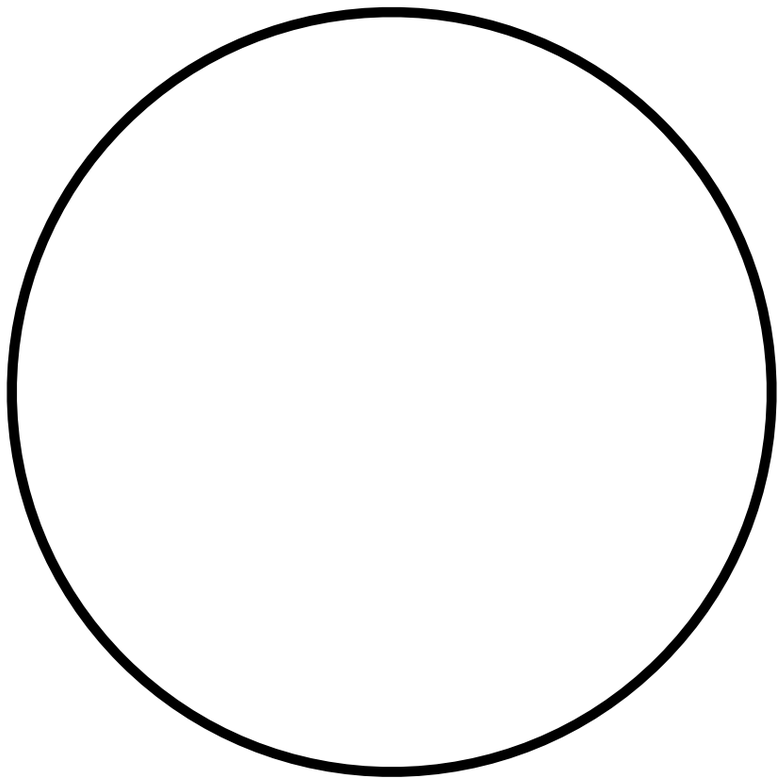
Påminnelse om arv




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

Arv och abstrakta klasser

En klass för cirklar



Circle
- radius - color - position
+ findArea + findPerimeter + move

En klass för cirklar

```
public class Circle1 {
  private double radius;
  private String color;
  // the circles center position
  private int x = 0;
  private int y = 0;
  private static int nbrOfCircles = 0;
  // Constructor with specified
  // radius and color
  public Circle1(double radius,String color) {
    this.color = color;
    this.radius = radius;
    nbrOfCircles = nbrOfCircles+1;
  }
  // Default constructor
  public Circle1() {
    this(0, "none");
  }
  // Getter method for radius
  public double getRadius() {
    return radius;
  }
  // Setter method for radius
  public void setRadius(double radius) {
    this.radius = radius;
  }
  // methods for color, x, y, ...
  public void setColor(String color) {
    this.color = color;
  }
  // move relative ...
  public void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
  // ...
  public double findArea() {
    return radius*radius*Math.PI;
  }
  public double findPerimeter() {
    return 2*radius*Math.PI;
  }
}
```

Att använda cirkeln

```
public class TestaGeom1 {
   public static void main(String[] args){
       Circle1 c1 = new Circle1();
       Circle1 c2 = new Circle1(2.0,"brown");
       System.out.println("c1 = " + c1);
       c1.setRadius(2.0);
       c1.setColor("brown");
       System.out.println("c1 = " + c1);
       System.out.println("c2 = " + c2);
       if ( c1 == c2 ) {
           System.out.println("c1 är \"==\" lika med c2");
       }
       if ( c1.equals(c2) ) {
           System.out.println("c1 är \"equals\" lika med c2");
       }
       System.out.println("arean är = “ + c1.findArea());
   }
}
```

Utskrift:

```
c1 = Circle1@270b73
c1 = Circle1@270b73
c2 = Circle1@d58aae
arean är = 12.566370614359172
```

äsch…

Vi måste överskugga `toString` och `equals`

```
public class Circle1 {

  ...

  // Override the toString() method
  // defined in the Object class
  public String toString() {
    return "[Circle] radius = " + radius;
  }
}
```

Vi bestämmer själva vad som skall skrivas ut.

Ny bättre utskrift:

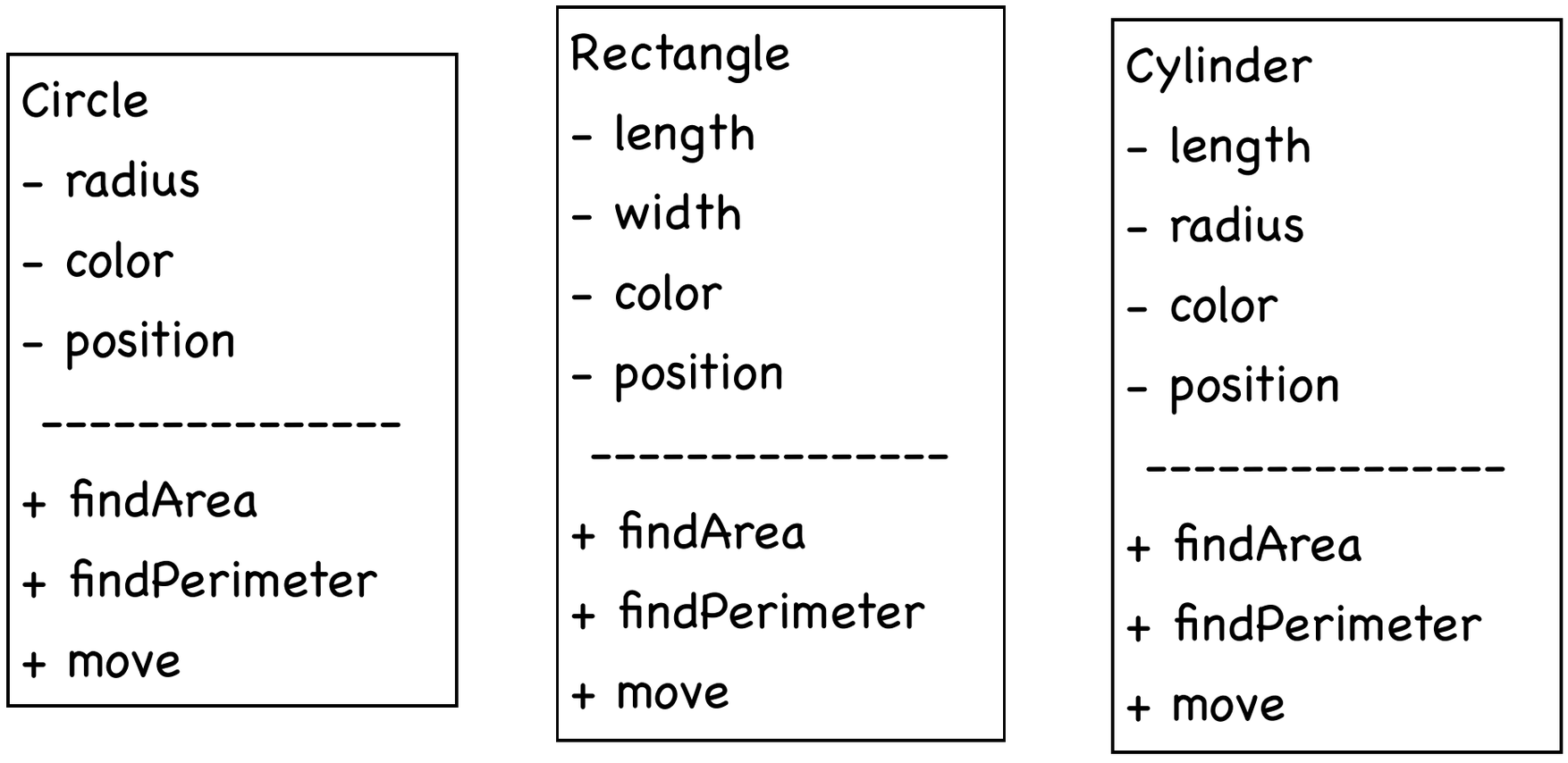
```
c1 = [Circle] radius = 0.0
c1 = [Circle] radius = 2.0
c2 = [Circle] radius = 2.0
arean är= 12.566370614359172
```

Koden för `equals` kommer senare…

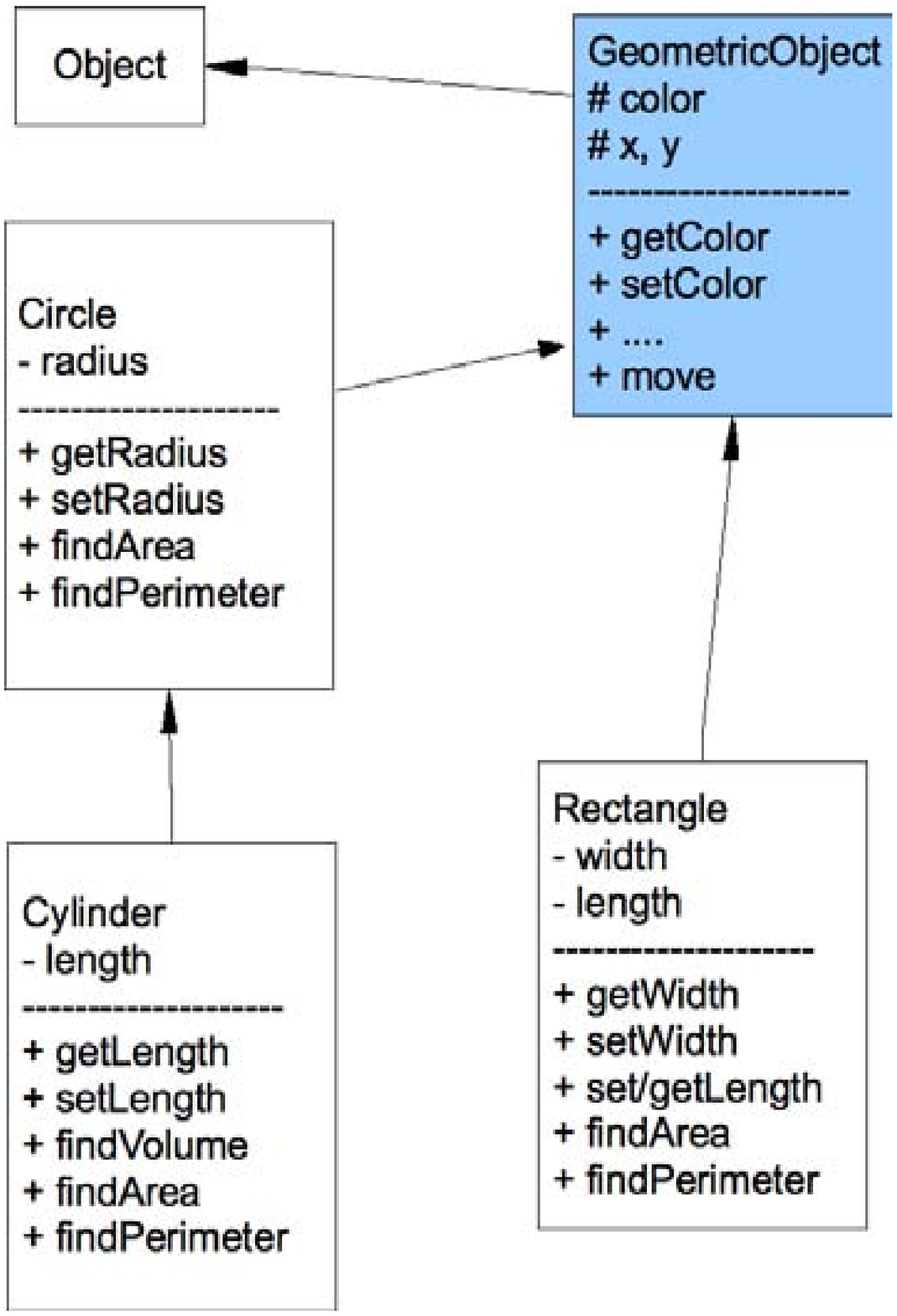
Vi skapar fler geometriska former

De har en del gemensamt eller hur.

Kan vi abstrahera ut de gemensamma egenskaperna? tex color, position, move, findArea samt findPerimeter?



Abstrahera



Men hur göra med findArea/ findPerimeter?

Arean beräknas ju på olika sätt i de olika formerna…

Låt oss vänta med dem ett tag.

GeometricObject och Circle

```
public class GeometricObject{
  private String color;
  private int x = 0;
  private int y = 0;
  public GeometricObject() {
    color = "white";
  }
  public GeometricObject(String color) {
    this.color = color;
  }
  // getters and setters for
  // color, x,y
  ...
  // move relative, ev. final
  public void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

```
public class Circle extends GeometricObject {
  private double radius;
  // Default constructor
  public Circle() {
    this(1.0, "white");
  }
  // with specified radius & color
  public Circle(double radius,String color) {
    super(color);
    this.radius = radius;
  }
  // Getter method for radius
  public double getRadius() {
    return radius;
  }
  public double findPerimeter() {
    return 2*radius*Math.PI;
  }
  // TODO the findArea method
}
```

En svaghet med lösningen

```
public class GeometricObject{
  private String color;
  private int x = 0;
  private int y = 0;
  public GeometricObject() {
    color = "white";
  }
  public GeometricObject(String color) {
    this.color = color;
  }
  // getters and setters for
  // color, x,y
  ...
  // move relative, ev. final
  public void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

Nu kan man skapa objekt som är "geometriska objekt"… Hur ser ett sånt ut?

```
... = new GeometricObject();
```

Vill man förhindra detta kan man göra på (minst) 2 sätt:

Sätt 1:

låt konstruktorerna vara **protected**

```
protected GeometricObject() {
  color = "white";
} ...
```

Då kan den bara användas av klasser i samma paket eller av klasser som ärver klassen GeometricObject.

Sätt 2: gör klassen abstrakt!

Abstrakt klasser!

Vi skulle ju vilja tvinga alla subklasser att ha metoder för findArea och findPerimeter trots att dessa måste implementeras i respektive klass.

Vi kan ha en ”abstrakt klass”.

abstrakt klass ≈ halvfärdig klass

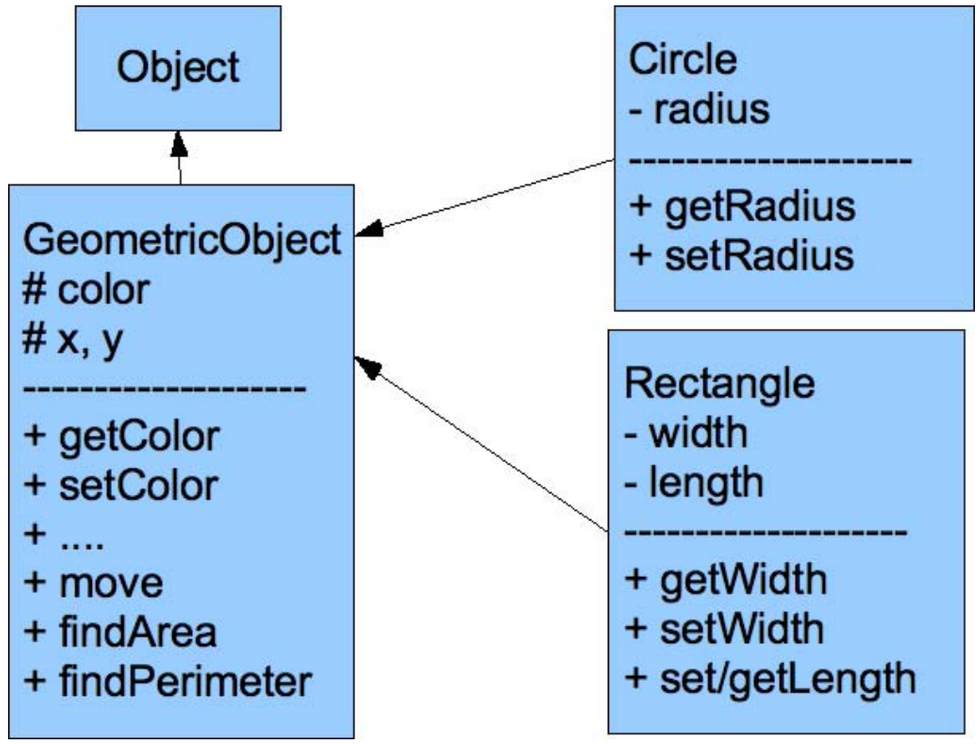
Något fattas alltså.

Man kan inte skapa en instans av en abstrakt klass.

Subklassen måste implementera de abstrakta delarna även om den inte behöver dem.

En abstrakt klass

```
public abstract class GeometricObject{
  private String color;
  private int x = 0;
  private int y = 0;
  public GeometricObject() {
    color = "white";
  }
  public GeometricObject(String color) {
    this.color = color;
  }
  // metoder som vi inte vill implementera här
  public abstract double findArea();
  public abstract double findPerimeter();
  // getters and setters for
  // color, x,y
  ...
  // move relative, ev. final
  public void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

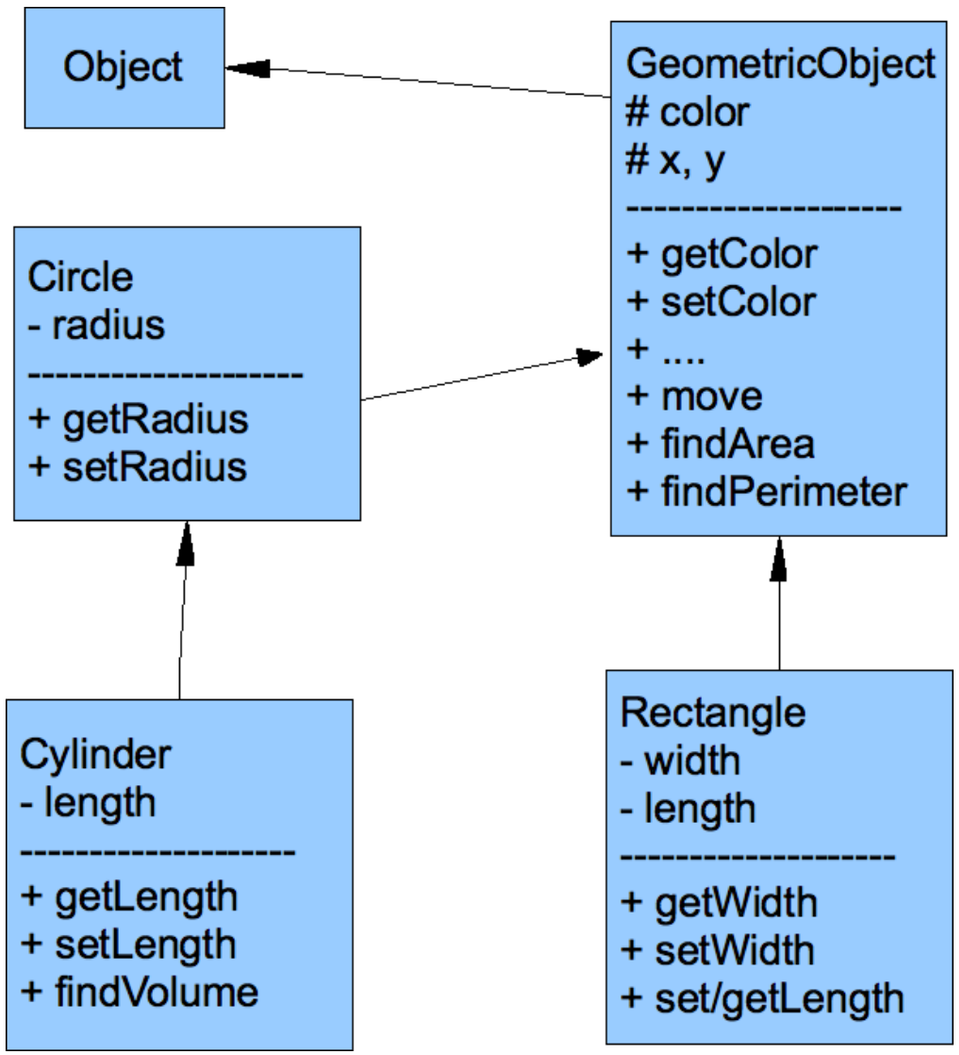


Cirkel klassen som förr:

```
public class Circle extends GeometricObject {
  ...
  public double findPerimeter() {
    return 2*radius*Math.PI;
  }
  ...
}
```

En cylinder klass

```
class Cylinder extends Circle {
  private double length;
  // 3 olika sätt att göra Konstruktorer
  // Default
  public Cylinder() {
    super();
    // defaultvärden ges
    length = 1.0;
  }
  // Construct a cylinder
  public Cylinder(double radius, double length) {
    this(radius, "white", length);
  }
  // Construct a cylinder
  public Cylinder(double radius,
                  String color,
                  double length) {
    super(radius, color);
    this.length = length;
  }
  ...
}
```



Metoder i cylinder klassen

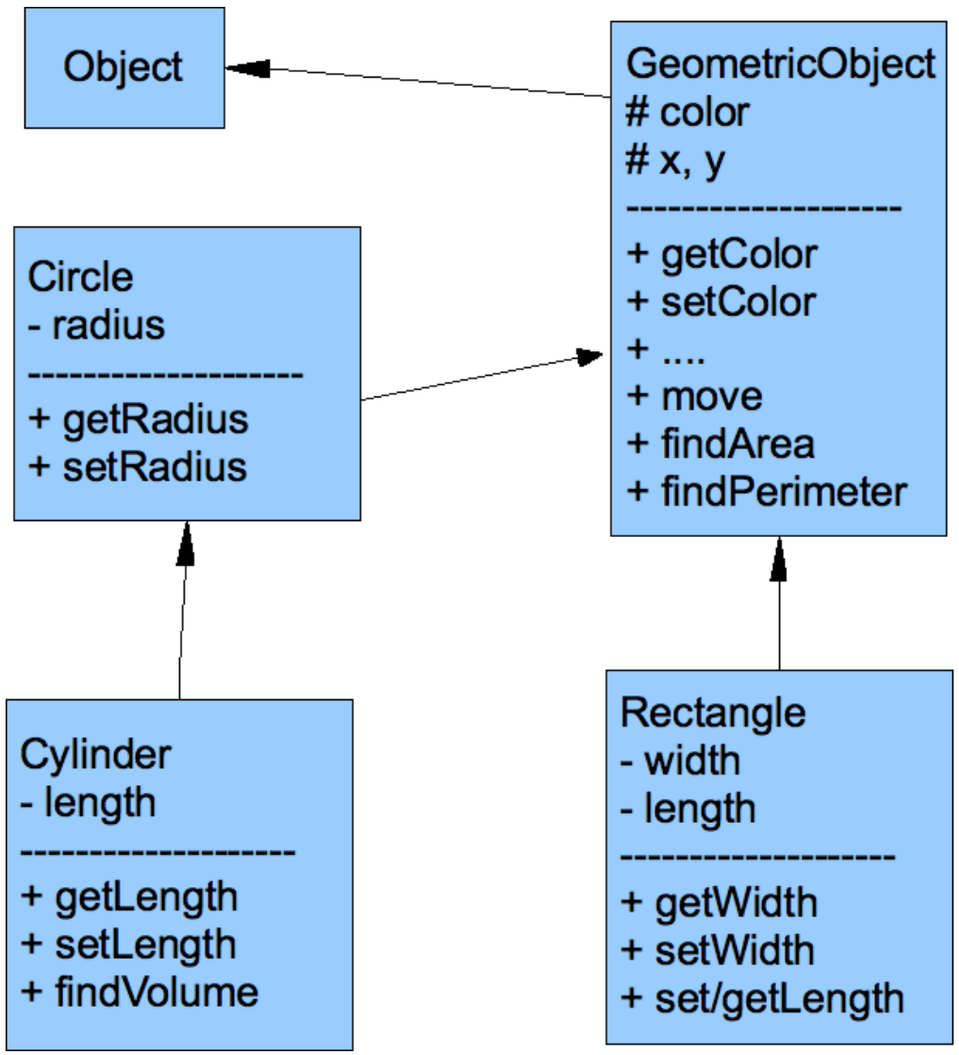
```
class Cylinder extends Circle {

  ...

  public double getLength( ...
  public void setLength( ...
  // Implement the findArea method
  // defined in GeometricObject
  // oops – redan gjort i Circle
public double findArea() {
  return 2*super.findArea()
          + (2*getRadius()*Math.PI)*length;
}
// Find cylinder volume
public double findVolume() {
  return super.findArea()*length;
}

...

// Override the equals() method ...
// Override the toString() method ...
```



Equals är svårast. Vi väntar lite till med den.

Det måste finnas en toString i varje klass.

I GeometricObject:

```
public String toString() {
  return "color " + color + ", x " + x + ", y " + y;
}
```

I Circle:

```
public String toString() {
  return super.toString() + ", radius " + radius;
}
```

I Cylinder:

```
public String toString() {
  return super.toString() + ", length " + length;
}
```

Koden:

```
Cylinder cy1 = new Cylinder(3.5, "white", 4);
System.out.println("cy1: " + cy1);
```

Ger: `cy1: color white, x 0, y 0, radius 3.5, length 4.0`

equals

Korrekt equals test i GeometricObject när arv är inblandat..

```
public class GeometricObject{
  ... allt annat som förr
  public boolean equals (Object rhs) {
    // This is the correct test,
    // if class is not final
    if (this == rhs) { // samma ident.
      return true; // snabbare test
    } else if ( rhs == null
                || this.getClass() != rhs.getClass()) {
      return false;
    } else {
      GeometricObject tmp = (GeometricObject) rhs;
      return color.equals(tmp.getColor())
             && x == tmp.getX()
             && y == tmp.getY();
    }
  }
}
```

kollar om det är samma (sub)klass

typkonvertering neråt från **Object;** varför fungerar det här alltid?

Svar: vi kolla just att detta är samma klass, dvs någon subklass av GeometricObject

equals (forts)

Korrekt equals test i Circle.

```
public class Circle extends GeometricObject {
  private double radius;
  ... allt som förr
  public boolean equals(Object rhs) {
    // Class test not needed,
    // getClass() done in
    // superclass equals
    return super.equals(rhs) && radius == ((Circle)rhs).getRadius();
  }
}
```

Begrepp

Polymorphism och Dynamisk bindning

- en polymorf referens kan referera objekt av olika typ.
- förmågan att överlagra
- dynamisk bindning är förmågan att vid runtime avgöra vilken kod som skall köras utifrån det aktuella objektets typ.

Overloading (Överlagra)

Metoder med samma namn men olika parameterprofil.

Overriding (Överskuggning)

När en metod i en subklass definierar om en metod i superklassen.
Samma namn och samma parameterprofil.

Metoden i superklassen kan fortfarande nås med super.metod(parametrar).

`this` och `super`

`this` refererar till objektet själv.

Får ej ges nytt värde.

```
... if ( this == rhs ) { ...
... this(x, y, z) ...
... metodNamn(this, ...)
... this.x = ...
```

"är **rhs** samma som jag?"

callar på konstruktor metoden

detta objekt som parameter

variabel i detta objekt

`super` refererar till superklassen.

super anropar alltså konstruktorer eller varaibler/metoder i superklassen.

```
super()
super(parametrar)

super.metodnamn(parametrar)
```

anropar superklassens konstruktor, måste ske innan annan kod

metod i superklassen

Kommer ni ihåg interface

Interface kan ärva (en eller *flera*) interface:

```
public interface BetterComparable extends Comparable {
   ...
}
```

En klass kan implementera *flera* interface

```
public class Test implements Comparable, OtherInterface {
   ...
}
```

Man kan typkonvertera till interface typen:

```
Test t = new Test();

Comparable c = t;
```

Underhållning

https://www.youtube.com/watch?v=k4RRi_ntQc8