TDA 545: Objektorienterad programmering

Föreläsning 10:
Abstrakta klasser, gränssnitt

Magnus Myréen

Chalmers, läsperiod 1, 2015-2016

Idag

Läsanvisning: kap 9, 10 och 11

Idag:

skillnaden mellan klass- och instansvariabler

abstrakta klasser dvs `abstract`

gränssnitt dvs `interface` och `implements`

(att läsa indata dvs input)

Nästa gång: *rekursion*; läsanvisning: kap 19, men bara till sida 812

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

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

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

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

änder 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: …änder det:

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

…nda …der …t, …etoder …y klassen

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`.

Underhållning

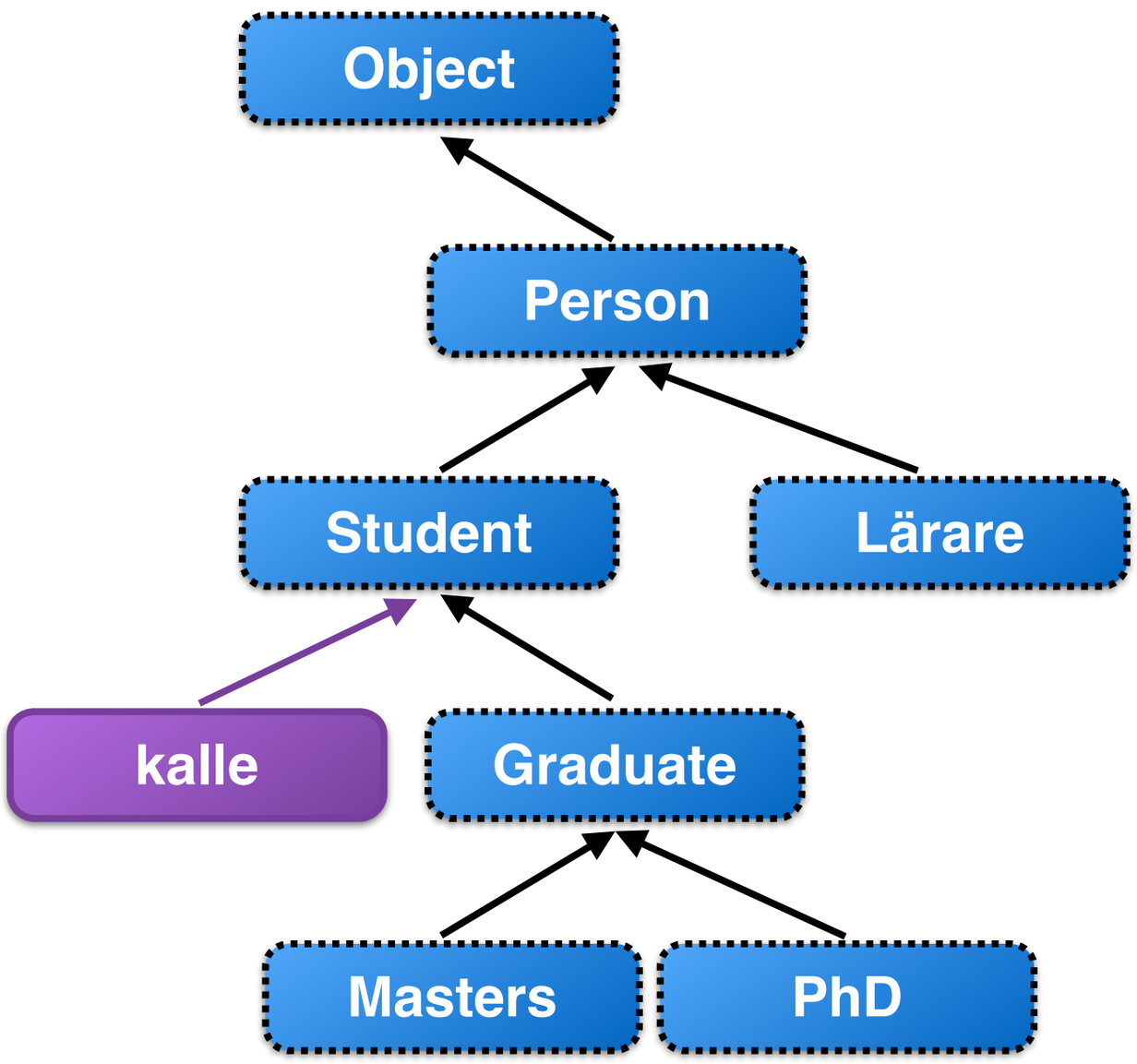
https://www.youtube.com/watch?v=k4RRi_ntQc8

Tillbaka till kursen…

Nästa koncept:

abstrakta klasser dvs `abstract`

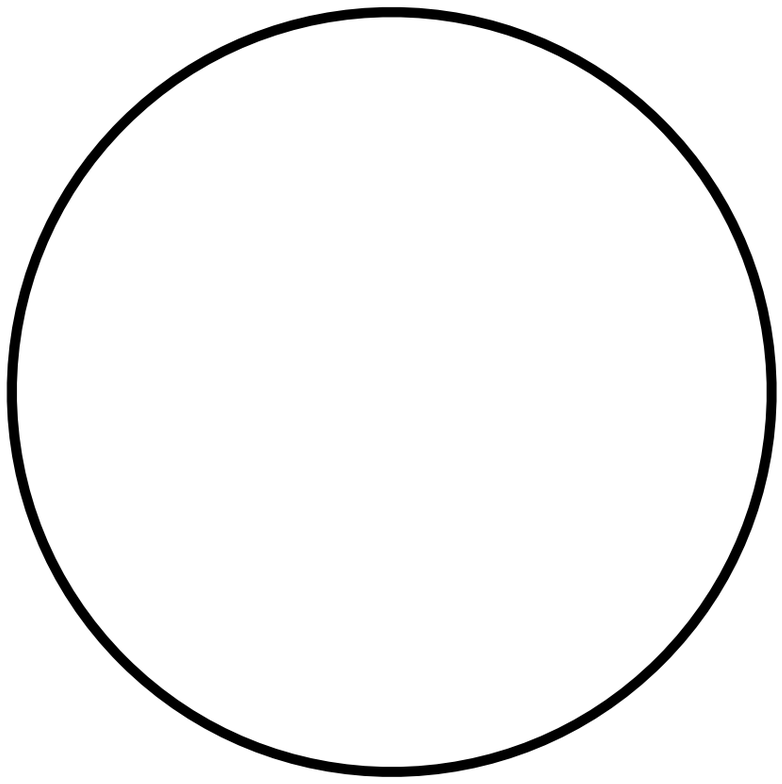
Påminnelse om arv




```
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:

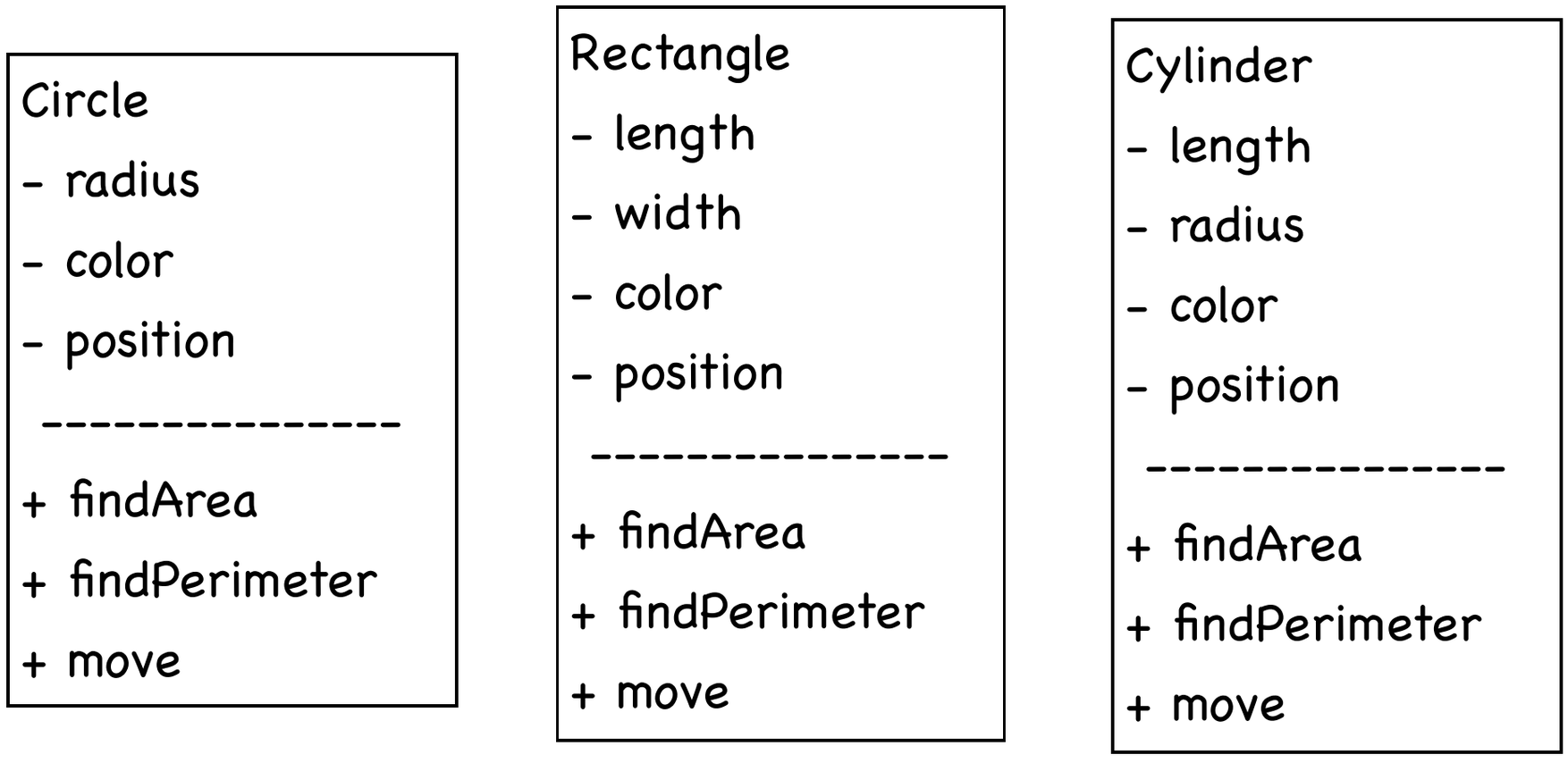
```
c1 = [Circle] radius = 0.0
c1 = [Circle] radius = 2.0
c2 = [Circle] radius = 2.0
arean är= 12.566370614359172
```

Koden för `equals` kommer senare…

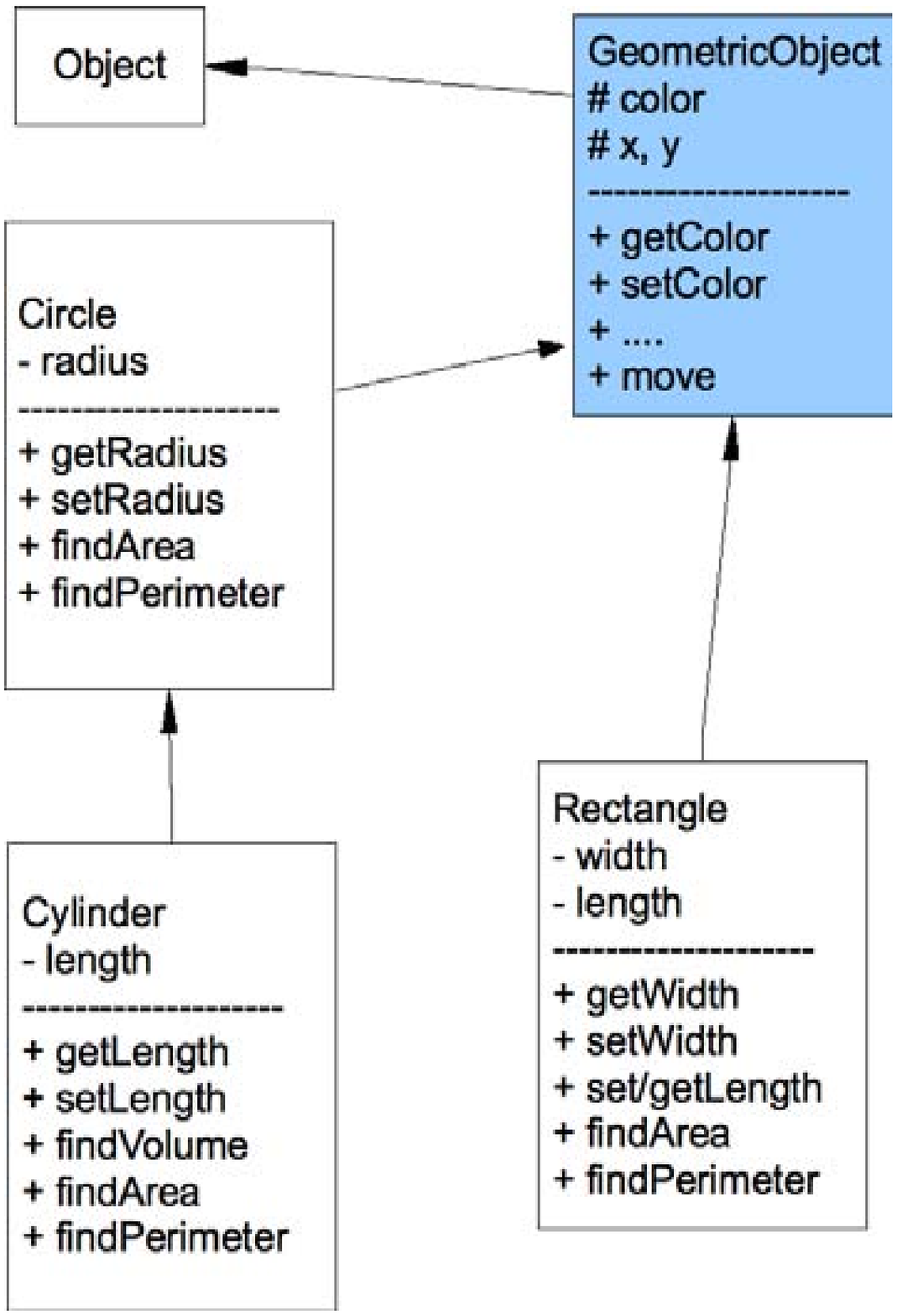
Vi skapar fler geometriska former

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

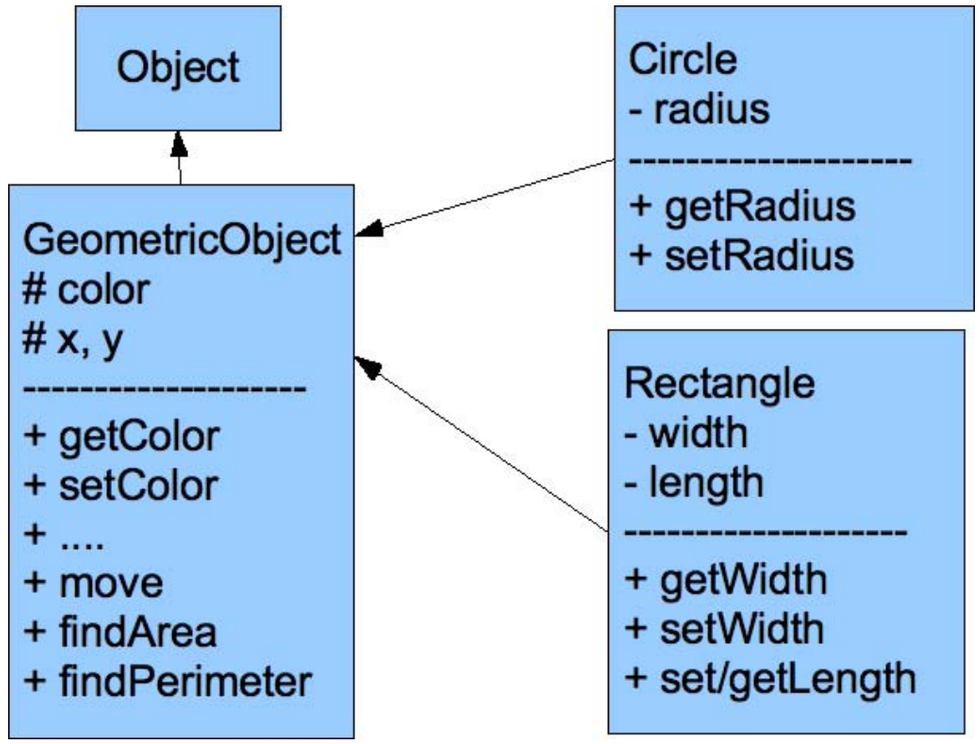
Något fattas alltså.

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;
  }
}
```

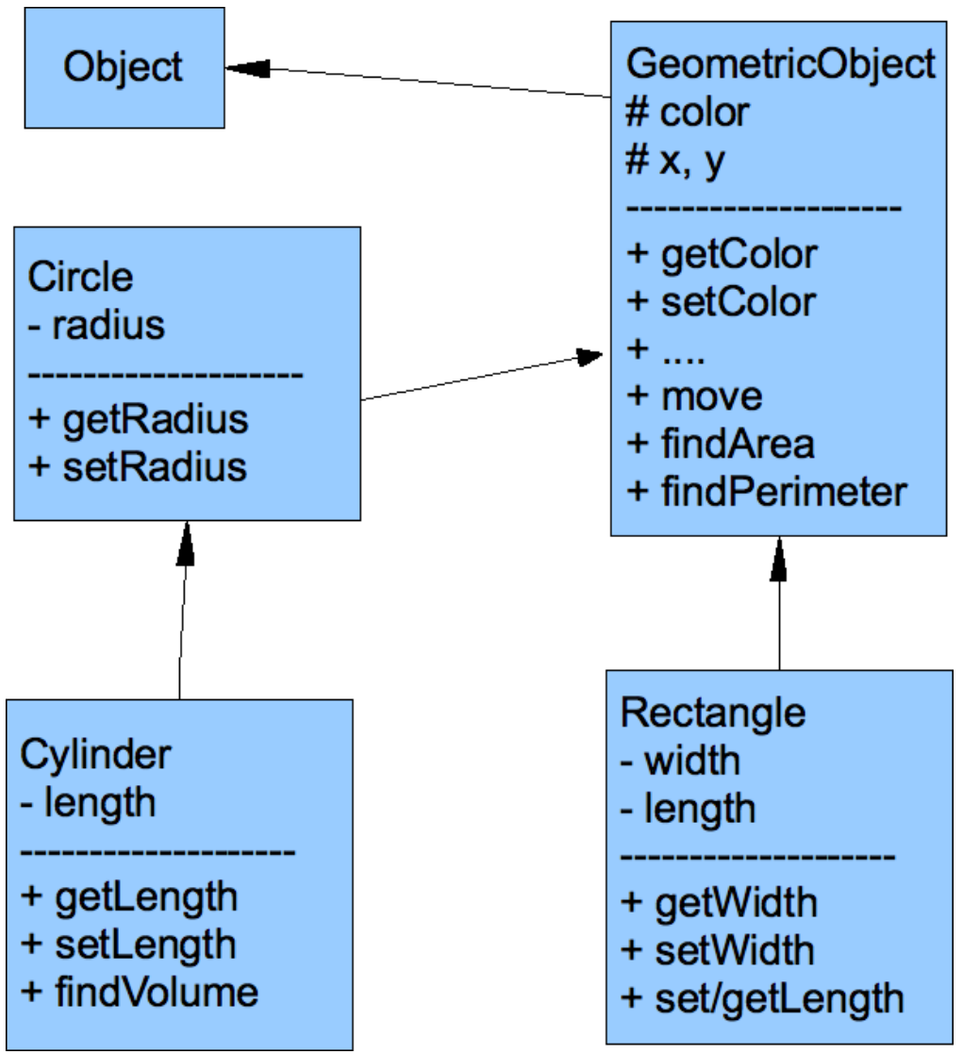


Cirkel klassen som förr:

```
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

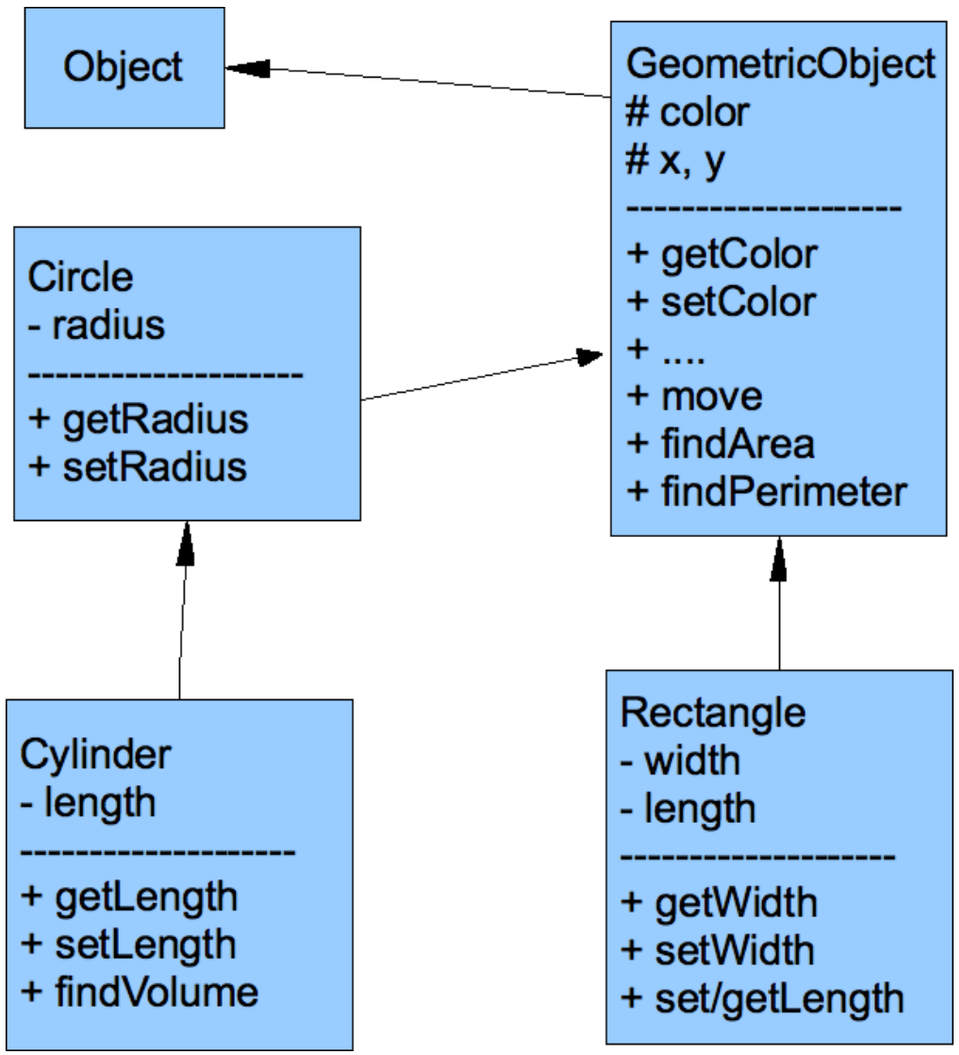
```
class Cylinder extends Circle {

  ...

  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

Idag

Läsanvisning: kap 10 och 11

Idag:

skillnaden mellan klass- och instansvariabler

abstrakta klasser dvs `abstract`

gränssnitt dvs `interface` och `implements`

(att läsa indata dvs input)

Nästa gång: *rekursion*; läsanvisning: kap 19, men bara till sida 812

Javas “`interface`” (Gränssnitt)

“en klass” som bara innehåller abstrakta metoder och konstanter kallas ett interface.

(En abstrakt klass kan alltså även innehålla konkreta metoder och varabler.)

Ett interface används ungefär som en abstrakt klass men nyckelordet är `implements`. Används för att få effekten av multipelt arv.

```
public interface Comparable {
  public int compareTo(Object o);
}

class ComparableCircle
          extends Circle
            implements Comparable {
  ...
```

sök: Java API Comparable

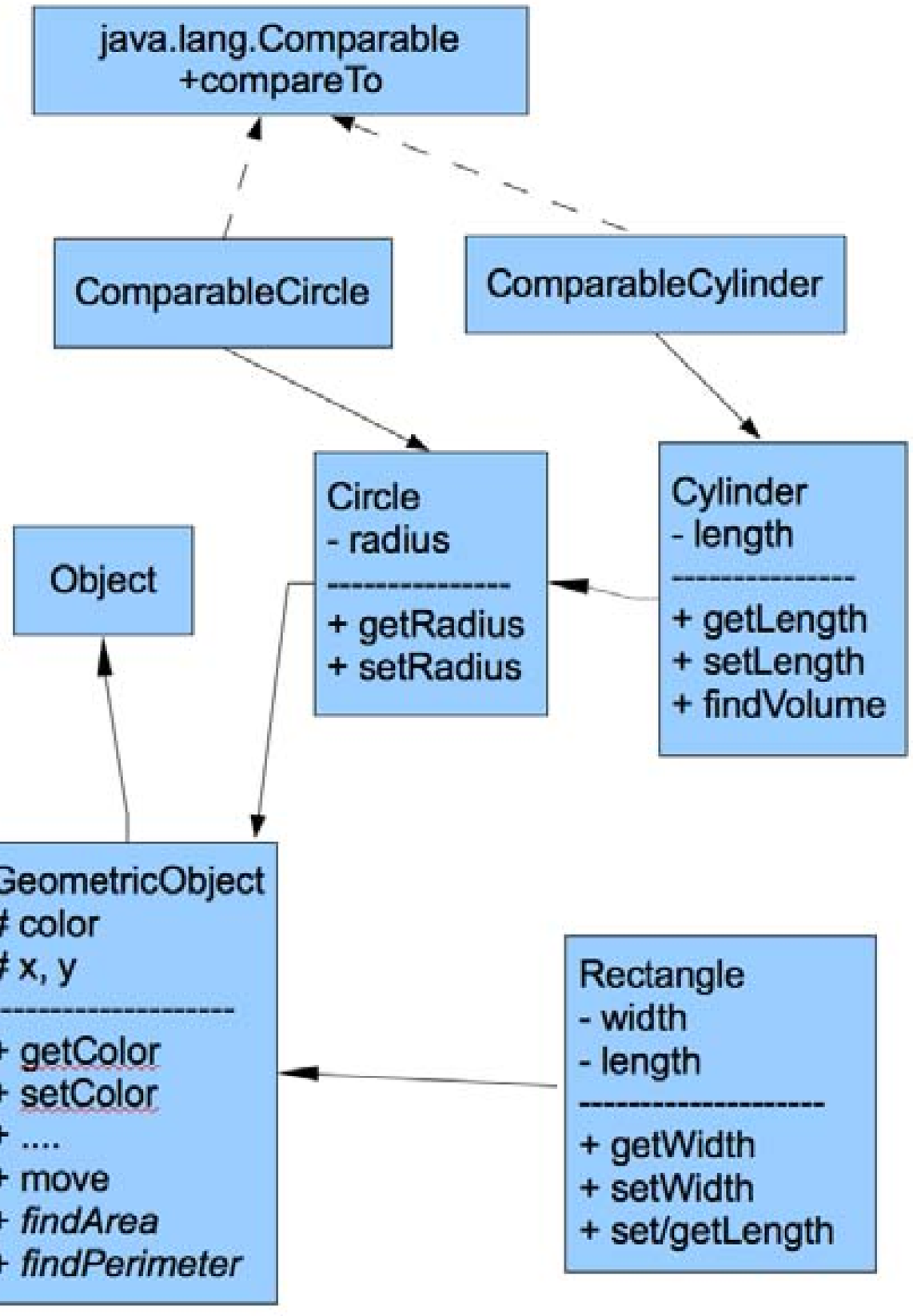
läs som: jag implementerar alla metoder i interfacet **Comparable**

ComparableCircle är en subklass till Circle, som implementerar Comparable interfacet.

ComparableCircle

ComparableCircle som implementerar interfacet Comparable

```
class ComparableCircle extends Circle implements Comparable {
  // Construct a ComparableCircle
  // with specified radius
  public ComparableCircle(double r){
    super(r);
  }
  // Implement the compareTo method
  // defined in Comparable
  public int compareTo(Object o) {
    if (o instanceOf ComparableCircle){
      if (getRadius() > ((Circle)o).getRadius()) {
        return 1;
      } else if (getRadius() < ((Circle)o).getRadius()) {
        return -1;
      } else {
        return 0;
      }
    } else {
      throw new IllegalArgumentException();
    }
  }
}
```


java.lang.Comparable
+compareTo
ComparableCircle
ComparableCylinder
Circle
- radius

+ getRadius
+ setRadius
Cylinder
- length

+ getLength
+ setLength
+ findVolume
Object
GeometricObject
color
x, y

+ getColor
+ setColor
+ ....
+ move
+ findArea
+ findPerimeter
Rectangle
- width
- length

+ getWidth
+ setWidth
+ set/getLength

Lite mera om 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;
```

Idag

Läsanvisning: kap 10 och 11

Idag:

skillnaden mellan klass- och instansvariabler

abstrakta klasser dvs `abstract`

gränssnitt dvs `interface` och `implements`

(att läsa indata dvs input)

Nästa gång: *rekursion*; läsanvisning: kap 19, men bara till sida 812

Sammanfattning av hur man kan läsa indata

Det finns flera sätt att göra detta på:

- argument på kommandoraden
- klassen Scanner
- grafiskt t.ex. klassen JOptionPane
- strömmar, dvs streams (se fortsättningskursen)

Att läsa från kommandoraden

Ofta vill man kunna läsa in parametrar som givits på kommandoraden enligt:

```
$ cat -n fil1 fil2
```

Det är dags att titta lite närmare på **`main`**-metodens specifikation:

```
public static void main(String[] args) { ...
```

Parametern args är ett fält med strängar:

`args[0]` är strängen `"-n"`
`args[1]` är strängen `"fil1"`
`args[2]` är strängen `"fil2"`
`args.length` är `3`

Skriva ut sina argument i args

```
public class PrintArgs {
  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("Inga arg angavs");
    } else {
      for(int i=0; i<args.length;i++) {
        System.out.println("argument " + i +" är " + args[i]);
      }
    }
  } // end main
} // end  PrintArgs
```

```
$ java PrintArgs hi there
argument 0 är hi
argument 1 är there

$ java PrintArgs "hi there"
argument 0 är hi there
```

Scanner

Klassen java.util.Scanner förser oss med inmatningsprimitiver så vi kan läsa "*tokens*" från tangentbordet:

```
Scanner in = new Scanner(System.in);
```

Till scannern kan man ange

- en ström (tex System.in som ovan)
- ett filnamn (new File(<filnamn>))
- en sträng i vilken sökningen sker

Metoder i Scanner:

```
(ersätt X med Int, Double, ..., eller inget)
in.hasNextX() finns det ett X
in.hasNext() finns det någonting
in.nextX() ger nästa X
in.next() ger nästa token som sträng
in.nextLine() ger hela raden som en sträng
```

Scanner – generellt mönster

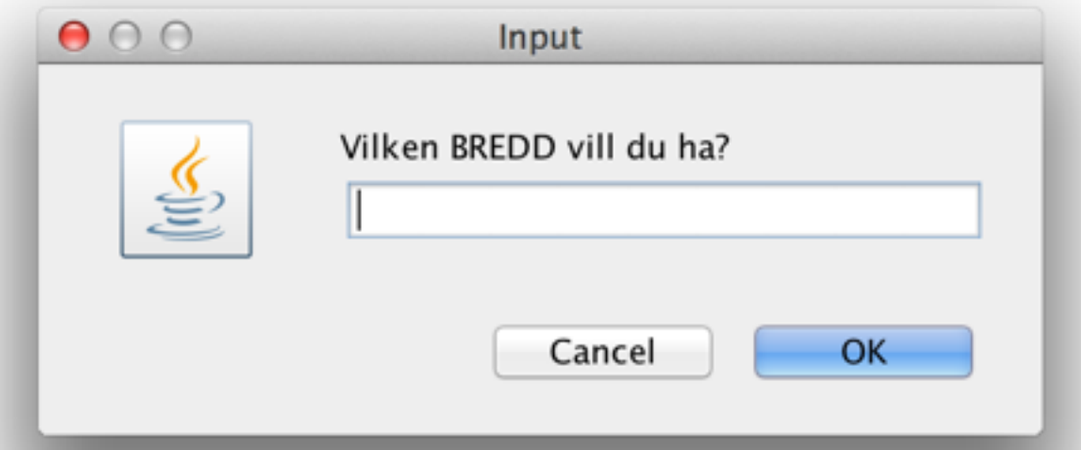
```
import java.util.*;
// breaks the input text into words.
public class ScannerTest {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    while(in.hasNext()) {
      String str = in.next();
      // gör ev något med str, t.ex.
      System.out.println(str);
    }
  }
}
```

Ett annat exempel

```
import java.util.Scanner;
import java.io.*;
public class PrintArgs2 {
  public static void main(String[] args){
    for (int i=0; i<args.length; i++) {
      try {
        Scanner in = new Scanner (new File(args[i]));
        while(in.hasNext()) {
          System.out.println(in.next());
        }
      } catch (FileNotFoundException e) {}
    }
  }
}
```

JOptionPane

Ofta vill man kommunicera med användaren på något grafiskt sätt. Ett lämpligt sätt att göra det på är att använda dialogrutor av olika slag dvs ett tillfälligt fönster med ett meddelande eller en fråga i vilket använd. kan ge ett svar. Enklast är det med JOptionPane.




```
import javax.swing.JOptionPane;

  ...

  // skapa frågefönster
  String ans = JOptionPane.showInputDialog("Vilken BREDD vill du ha?");
  if (ans != null) {
    int bredd = Integer.parseInt(ans.trim());
    ...
  }
```

I Lab 3&4 finns det om uppräkningstyper

En uppräkningstyp:

```
public class Car {

    public enum Direction {
        NORTH, SOUTH, EAST, WEST
    }
    Direction currentDirection = Direction.NORTH;

    public Car(Direction d) {
        currentDirection = d;
    }

}
```

Definierar en ny typ.
Typens namn är **Direction**.
NORTH är ett värde av typen **Direction**.

Man måste använda ett långt namn…

```
... new Car(Car.Direction.NORTH) ...
```

I Lab 3&4 finns det om uppräkningstyper

Nästan samma ***utan*** uppräkningstyper.

Nästan samma effekt utan **enum**.

```
class Car {

    public class Direction {
        public static final int NORTH = 0;
        public static final int SOUTH = 1;
        public static final int WEST = 2;
        public static final int EAST = 3;
    }

    int currentDirection = Direction.NORTH;

    public Car(int d) {
        currentDirection = d;
    }

}


... new Car(Car.Direction.NORTH) ...
```

Måste själv se till att värdena är olika…

Värdena (**NORTH**, **SOUTH** mm.) är av typen **int**.

Bättre att använda uppräkningstyper!