Konstruktorer till Hus

Var noga med att se till att objekten aldrig kan ta in felaktig data.

```
public class Hus {
  private double längd;
  private double bredd;
  private int antalVåningar;
  // Konstruktorer
  public Hus() {}
  public Hus(double l, double b, int v) {
    sättLängd(l); sättBredd(b); sättAntalVåningar(v);
  }
  // Instansmetoder
  public void sättLängd(double l) {
    if (l > 0)
      längd = l;
    else
      throw new IllegalArgumentException("Negativ längd");
  }
  etc.
}
```

Initiering av objekt:

1. Alla instansvariabler som deklareras i klassen sätts till sina defaultvärden (0 eller motsvarande).
2. En konstruktor för superklassen anropas.
3. De instansvariabler i klassen som har explicita initieringsuttryck initieras till dessa värden.
4. Satserna i subklassens konstruktor exekveras.

Varje klass ansvarar bara för sina egna variabler. Det betyder att klass C bara har sin gången variabel att ansvara för, resten av variablerna ansvarar superklassen för. Försök således inte att påverka superklassens variabler i subklassen utan använd dig av `super(...)` istället.

```
public class Bostadshus extends Hus {
  boolean tilläggsisolerat;
  // Konstruktorer
  Bostadshus(boolean isol) {
    // super() anropas automatiskt
    tilläggsisolerat = isol;
  }
  Bostadshus() {
    // super() anropas automatiskt
    tilläggsisolerat = true;
  }
  Bostadshus(double l, double b, int v, boolean isol) {
    super(l, b, v);        // måste ligga först
    tilläggsisolerat = isol;
  }
  // Instansmetoder
  public void isolera() {
    tilläggsisolerat = true;
  }
}
```

Polymorpha uttryck är uttryck av tre olika slag.

1. Överlagring - anropa en metod. För att veta exakt vilken metod man anropar måste man kika på parameterlistan. Olika parametrar ger olika metoder.
2. Generiska enheter - listor med olika slag. Snackar man bara om en lista har man inte röjt hela sanningen utan man måste även veta typen för att ha koll på vad man har att göra med.
3. Referens.metod() - en referens anropar en metod och vi har subklasser inblandade. Hår kan man inte - vid kompilation - veta vad som kommer hända. Först när man kör metoden kommer man veta ty det är då man skickar datan till metoderna. Det är detta (nr 3) som är **dynamisk bidning**.

```
Hus[] ha = new Hus[100];
ha[0] = new Hus(40, 25, 4);
ha[1] = new Bostadshus(40, 25, 4, true);
ha[2] = new Flerfamiljshus(40, 25, 4, true, 10);
etc.

for (int i=0; i<ha.length; i++)
  if (ha[i] != null)
    System.out.println("Ett " + ha[i].getClass().getName()
                       + " med ytan " + ha[i].yta());
```

Eller

```
for (Hus h : ha)
  if (h != null)
    System.out.println("Ett " + h.getClass().getName()
                       + " med ytan " + h.yta());
```

```
Ett Hus med ytan 4000.0
Ett Bostadshus med ytan 4000.0
Ett Flerfamiljshus med ytan 3800.0
```

Arrays är effektivare om man gör enkla saker och vet exakt storlek, innehåll och så vidare. När man däremot ska ha något som ska kunna ändras, lägga till, dra ifrån, osv - då är det en lista som är att föredra.

```
List<Hus> hl = new ArrayList<>();
hl.add(new Hus(40, 25, 4));
hl.add(new Bostadshus(40, 25, 4, true));
hl.add(new Flerfamiljshus(40, 25, 4, true, 10));

for (Hus h : hl)
  if (h != null)
    System.out.println("Ett " + h.getClass().getName()
                       + " med ytan " + h.yta());
```

Abstrakta saker

Hus skulle kunna ses som en abstrakt klass. Det finns inget som bara är ett "hus", det måste specificeras ytterligare (Villam, lägenhet, hotell...)

Man får inte skapa objekt till abstrakta klasser, bara göra referenser till dem. **Inga NEW av abstrakta klasser!**

```
public abstract class Hus {
  som tidigare
}
Hus h1 = new Hus();   // FEL!! Hus är en abstrakt klass
Hus h2 = new Flerfamiljshus();   // OK
```

Om det finns ett djur så vill vi ha en metod som kan rita ett djur. Vi kan dock inte rita `djur` utan behöver först specificera vilket djur det rör sig om.

Icke abstrakta subklasser måste implementera alla metoder i den abstrakta superklassen. Däremot behöver inte abstrakta subklasser implementera alla metoder i den abstrakta superklassen.
Även abstrakt klass får ha icke abstrakta metoder och den får ha instansvariabler.

```
public abstract class Djur {
  public abstract void rita();  // abstrakt metod
  övriga variabler om metoder
}

public class Tiger extends Djur {
  @Override
  public void rita() {
    här ligger satser som ritar en tiger
  }
  övriga variabler om metoder
}


Djur d = ...; // refererar till något djur, t.ex. en Tiger
d.rita();     // kan alltid utföras
```


Interface

Ett interface kan ses som en abstrakt superklass som bara innehåller abstrakta metoder. Inga instansvariabler eller annat bös. Du får bara en mass krav på vad du måste implementera.

Du behöver inte skriva @Override när du implementerar metoderna från interfacet. Interface gör detta av sig självt!

```
public interface Skrivbar {
  void skriv();
}

public class Punkt implements Skrivbar {
  public double x;
  public double y;
  public Punkt(double xx, double yy) {
    x = xx; y = yy;
  }
  public Punkt() {}         // defaultkonstruktor
  @Override
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
  public void skriv() {
    System.out.println(toString());
  }
}
```

Det är helt okej att implementera flera interfaces. Du lovar helt enkelt bara att du kan ännu fler saker. Däremot får du bara "extenda" en superklass.

```
public interface Ritbar {
   void rita();
}

public class Punkt implements Skrivbar, Ritbar {
  som tidigare
  public void skriv() {
    System.out.println(toString());
  }
  public void rita() {
    här ligger satser som ritar en punkt
  }
}


public abstract class GrafiskFigur {
  deklarationer av variabler om metoder
}

public class Punkt extends GrafiskFigur
                   implements Skrivbar, Ritbar {
  som tidigare
}


public abstract class Djur implements Ritbar {
  deklarationer av variabler om metoder
}
```

Man får INTE skapa objekt av interfacetyp. Precis som man inte får skapa objekt av abstrakta klasser.

Det är dock helt okej att skapa variabler av interfacetyp. Man kommer dock inte åt de andra grejerna i klasserna, bara det som interfacet talar om.

```
Skrivbar sk = new Skrivbar();  // FEL!!

// Men följande är OK
Punkt p = new Punkt(1.0, 2.5);
Tiger ti = new Tiger();
Skrivbar sk;
Ritbar ri1, ri2;
sk = p;
ri1 = p;
ri2 = ti;
ri2.rita();  // dynamisk bindning
```

Ett interface kan ärva andra interface.

```
public interface Presenterbar extends Ritbar, Skrivbar {
  void visa();
}
```

Du får lägga konstanta variabler i ett interface.

```
public interface Lagringsbar {
  int blockStorlek = 1024;    // automatiskt final och static
  void lagra();
}
Ritbar[] r = new Ritbar[100];
r[0] = new Tiger();
r[1] = new Punkt();
r[2] = new Mus();


for (int i=0; i<r.length; i++)
  if (r[i] != null)
    r[i].rita();
```

Java 8 - bös

I ett interface får du numera lägga till metoder så tillvida att de är markerade med `default`.

Det liknar mer och mer en abstrakt metod.

Back to Java 7
Jämförelser

`Comparable` innehåller bara en enda metod och den heter `compareTo`.

compareTo ska ta en parameter vilken talar om typen (därav `<Klockslag>`) man vill jämföra med.

Comparable är bra om man vill jämföra och sortera saker och ting och du får inte använda `sort` på något som inte är comparable med sig självt.

Man får bara jämföra två saker åt gången.

Exempel Cirkel

Nyhet i Java 8: default-metoder (defender methods, virtual extension methods)

```
public interface Skrivbar {
  void skriv() default {
    System.out.println(toString());
  }
}
```

Nu OK:

```
public class Punkt implements Skrivbar {
  public double x;
  public double y;
  public Punkt(double xx, double yy) {
    x = xx; y = yy;
  }
  public Punkt() {}          // defaultkonstruktor
  @Override
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

Default-metoder

- kränglar till reglerna för interface
- gör det möjligt att utöka ett interface utan att befintliga klasser som implementerar det behöver påverkas.
- gäller bara om ingen implementering skett på "normalt" sätt (egen implementering eller arv från superklass)
- upphävs om de omdefineras i ett subgränssnitt (även om de där inte är default-metoder)
- ger oklarheter när en viss metod finns i mer än ett gränssnitt vilka implementeras av en klass eller utökas i ett subgränssnitt. (Kan lösas med hjälp av **super**.)

Jämförelser

```
public class Klockslag implements Comparable<Klockslag> {
  int tim;
  int min;
  public int compareTo(Klockslag k) {
    if  (tim < k.tim || (tim == k.tim && min < k.min))
      return -1;
    else if (tim > k.tim || (tim == k.tim && min > k.min))
      return 1;
    else
      return 0;
  }
  @Override
  public String toString() {
    return String.format("%02d:%02d", tim, min);
  }
}

int i = kl.compareTo(k2);
if (i < 0)
  System.out.println(kl + " kommer före " + k2);
else if (i == 0)
  System.out.println("Samma klockslag");
else
  System.out.println(kl + " kommer efter " + k2);
```

```
public class Cirkel {
  // instansvariabler
  double x, y;     // mittpunktens koordinater
  double radie;
  // instansmetoder
  public void sättRadie(double r) {  // ändrar radien
    if (r >= 0)
      radie = r;
    else
      throw new IllegalArgumentException("Negativ radie");
  }
  public double area() {             // beräknar arean
    return Math.PI * radie * radie;
  }
  public double omkr() {             // beräknar omkretsen
    return 2 * Math.PI * radie;
  }
}
```

```
import java.util.*;     // innehåller bl.a. Comparator
public class JfrCirkel implements Comparator<Cirkel> {
  public int compare(Cirkel a, Cirkel b) {
    if (a.radie < b.radie)
      return -1;
    else if (a.radie > b.radie)
      return 1;
    else
      return 0;
  }
}

JfrCirkel jfr = new JfrCirkel();   // skapa en jämförare
if (jfr.compare(c1,c2) < 0)
  JOptionPane.showMessageDialog(null, "Den första cirkeln är minst");
```

Några metoder i klassen java.util.Arrays

`toString(a)`	ger en textversion av fältet `a`. Om komponenterna är objekt anropas `toString` för varje komponent.
`equals(a,b)`	ger **`true`** om fälten `a` och `b` är lika, dvs. har lika många komponenter och motsvarande komponenter är lika (`a[i].equals(b[i])` om komponenterna är objekt)
`sort(a)`	sorterar komponenterna i fältet `a`
`sort(a, c)`	som ovan, men använder en jämförare `c`
`sort(a,i1,i2)`	sorterar komponenterna nr `i1` till `i2-1` i fältet `a`
`sort(a,i1,i2,c)`	som ovan, men använder en jämförare `c`
`binarySearch(a,k)`	söker efter komponenten `k` i fältet `a`,
	ger `k`:s index i fältet om `k` finns, -1 annars
`binarySearch(a,k,c)`	som ovan, men använder en jämförare `c`
`asList(a)`	ger möjlighet att hantera fältet `a` som en lista

Några metoder i klassen java.util.Collections

`sort(l)`	sorterar elementen i listan `l`.
`sort(l, c)`	som ovan, men använder en jämförare `c`.
`binarySearch(l,e)`	söker efter elementet `e` i listan `l`,
	ger `k`:s index i listan om `k` finns, -1 annars.
`binarySearch(l,e,c)`	som ovan, men använder en jämförare `c`.
`max(l)`	ger det största elementen i listan `l`.
`max(l,c)`	som ovan, men använder en jämförare `c`.
`min(l)`	ger det minsta elementen i listan `l`.
`min(l,c)`	som ovan, men använder en jämförare `c`.