

Klasser och arv (2/2)

Tobias Wrigstad



Doxygen el. JavaDoc

- Doxygen är ett verktyg för att extrahera kommentarer ur kod och generera dokumentation

Hela JavaAPI:et är skapat med hjälp av JavaDoc från koden



MyClass.java

```
/** Description of MyClass
 *
 * @author John Doe
 * @author Jane Doe
 * @version 6.0z Build 9000 Jan 3, 1970.
 */
public class MyClass
{
    /** Description of myIntField */
    public int myIntField;
    /** Description of MyClass()
     *
     * @throws MyException Description of when the exception is thrown
     */

    public MyClass() throws myException
    {
        // Blah Blah Blah...
    }
    /** Description of myMethod(int a, String b)
     *
     * @param a Description of a
     * @param b Description of b
     * @return Description of c
     */
    public Object myMethod(int a, String b)
    {
        Object c;
        // Blah Blah Blah...
        return c;
    }
}
```

```
$ javadoc -d docs MyClass.java
```

```
Generating docs/package-frame.html...
Generating docs/package-summary.html...
Generating docs/package-tree.html...
Generating docs/constant-values.html...
Building index for all the packages and classes...
Generating docs/overview-tree.html...
Generating docs/index-all.html...
Generating docs/deprecated-list.html...
Building index for all classes...
Generating docs/allclasses-frame.html...
Generating docs/allclasses-noframe.html...
Generating docs/index.html...
Generating docs/help-doc.html...
```

Resultatet!

Class MyClass

java.lang.Object
MyClass

```
public class MyClass
extends java.lang.Object
```

Description of MyClass

Field Summary

Fields

Modifier and Type	Field and Description
int	myIntField Description of myIntField

Constructor Summary

Constructors

Constructor and Description

MyClass()
Description of MyClass()

Method Summary

All Methods

Instance Methods

Concrete Methods

Modifier and Type	Method and Description
java.lang.Object	myMethod (int a, java.lang. Description of myMethod(int a, Str

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString,

Field Detail



```
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#ifndef __gc__
#define __gc__
/// The opaque data type holding all the heap data
typedef struct heap_t heap_s;
/// The signature of the trace function
typedef void (*trace_f)(heap_s *h, void *obj);
/// The signature of object-specific trace functions. It will be
/// called for its specific objects, and be given a generic trace
/// function f to be called on each pointer inside obj.
typedef void (*s_trace_f)(heap_s *h, trace_f f, void *obj);
/// Create a new heap with bytes total size (including both spaces
/// and metadata), meaning strictly less than bytes will be
/// available for allocation.
///
/// \param bytes the total size of the heap in bytes
/// \return the new heap
heap_s *h_init(size_t bytes);
/// Delete a heap.
///
/// \param h the heap
void h_delete(heap_s *h);
/// Delete a heap and trace, killing off stack pointers.
///
/// \param h the heap
/// \param dbg_value a value to be written into every pointer into h on the stack
void h_delete_dbg(heap_s *h, void *dbg_value);
/// Allocate a new object on a heap with a given format string.
///
/// Valid characters in format strings are:
/// - 'd' -- for sizeof(int) bytes 'raw' data
/// - '*' -- for a sizeof(void *) bytes pointer value
/// - '\0' -- null-character terminates the format string
///
/// \param h the heap
/// \param layout the format string
/// \return the newly allocated object
```

Project Work Test Implementation

Main Page	Classes	Files	Search
File List	File Members		

gc.h File Reference

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

[Go to the source code of this file.](#)

Typedefs

```
typedef struct heap_t heap_s
    The opaque data type holding all the heap data.

typedef void *(* trace_f)(heap_s *h, void *obj)
    The signature of the trace function.

typedef void *(* s_trace_f)(heap_s *h, trace_f f, void *obj)
```

Functions

```
heap_s * h_init (size_t bytes)
void h_delete (heap_s *h)
void h_delete_dbg (heap_s *h, void *dbg_value)
void * h_alloc_struct (heap_s *h, char *layout)
void * h_alloc_union (heap_s *h, size_t bytes, s_trace_f f)
void * h_alloc_data (heap_s *h, size_t bytes)
size_t h_gc (heap_s *h)
size_t h_avail (heap_s *h)
```

Typedef Documentation

```
typedef void *(* s_trace_f)(heap_s *h, trace_f f, void *obj)
```

The signature of object-specific trace functions. It will be called for its specific objects, and be given a generic trace function f to be called on each pointer inside obj.

doxygen -g gc.doxy

Skapar en fil med
inställningar

emacs gc.doxy

Ändra på input så
att rätt filer pekas ut

doxygen -s gc.doxy

Generera
dokumentationen

Typer

- I Java ger klasser och interface upphov till nya typer. Ex.:

```
interface Stack { ...  
class LinkedStack implements Stack { ...
```

```
LinkedStack s = ...  
s.push(1); // OK  
s.pshu(2); // Not OK  
Stack i = s; // OK
```

- En typ är en etikett som ger en semantisk mening till ett dataobjekt
- I programspråk som använder *statisk typning* har varje uttryck en typ som kan räknas ut vid kompileringstillfället – detta tillåter oss att säga nej till raden `s.pshu(2)` ovan
- Ett typsystem är en syntaktisk metod för att bevisa avsaknaden av vissa klasser av fel genom att klassificera ett programs alla uttryck utefter vilka värden de beräknar¹

¹Types and Programming Languages. B.C. Pierce

Konstruktion av klasser

Aggregering

Klasser kan byggas på redan definierade klasser genom att klassobjekt används som dataattribut när en ny klass beskrivs.

Exempel:

- En klass `PairOfDice` kan konstrueras med två attribut av typen `Die`.
- En klass `CardDeck` kan byggas med hjälp av ett arrayobjekt innehållande 52 objekt av typen `Card`
- En klass `ListMap` kan konstrueras med hjälp av `ListNode`-objekt
- En klass `TrafficSystem` kan sättas ihop med objekt ur klasser som `Lane`, `Light`, `Vehicle` ...

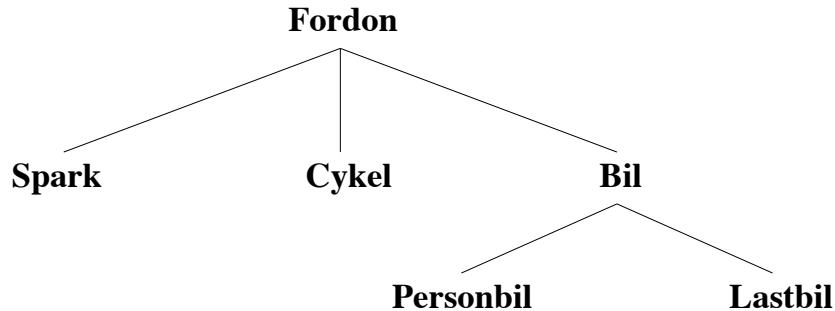
Man brukar säga att detta är en *består av*-relation (has-a)

Arv

En klass kan också byggas som en *specialisering* av en annan klass.

Exempel:

- Klassen StackException som används i Stack-klassen är skriven som en *subklass* till Exception
- Om vi har en klass Fordon så kan vi bygga nya klasser som t.ex. Spark, Cykel och Bil. En Bil kan i sin tur vara basclass för klasserna Personbil och Lastbil



Arv (forts)

I dessa fall säger man att man har en *klasshierarki*

- Klassen Fordon sägs vara en *basklass* (ä. superklass)
- Klasserna Spark, Cykel och Bil är *subklasser* eller *underklasser* till klassen Fordon
- En subklass (t.ex. klassen Bil) kan användas som *basklass* för andra klasser (t.ex. Personbil och Lastbil)

Basklass/superklass/subklass beskriver en klass *relation* till en annan klass(er). Det är *inte* en egenskap hos klassen.

Arv är en *är en*-relation (is a). Man kan pröva om arv är lämpligt genom att försöka sätta in subklassen B och superklassen A i frasen:

B är en A

ex. ”en Bil är ett Fordon”, ”en Människa är ett däggdjur”

Arv (forts)

Ett objekt ur en subclass

- får automatiskt basklassens attribut och metoder och
- kan dessutom lägga till egna

Man säger att subclassen *ärver* basklassens egenskaper.

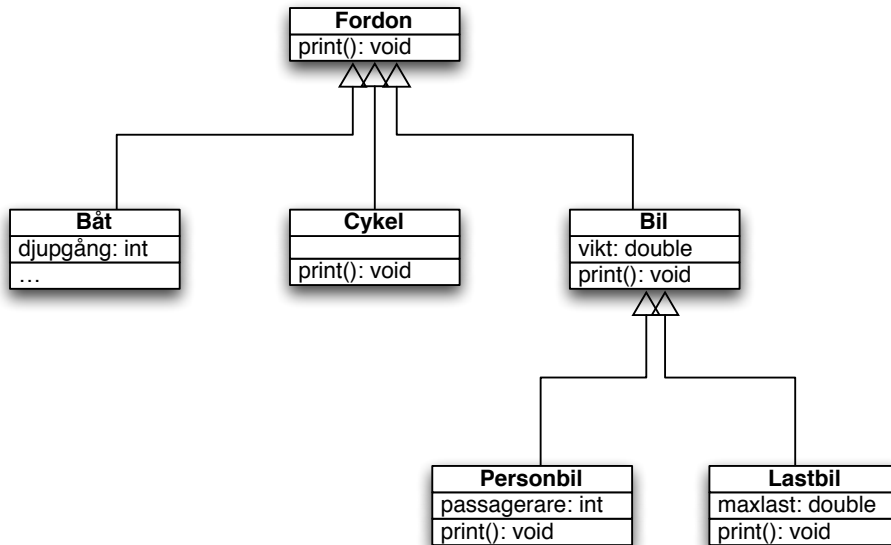
Exempel: Klassen Bil kan ha egenskapen vikt. En Personbil kan tillfoga passagerare medan en Lastbil kan tillfoga maxlast.

En subclass är en *specialisering* av sin superklass (jmf. Bil–Fordon)

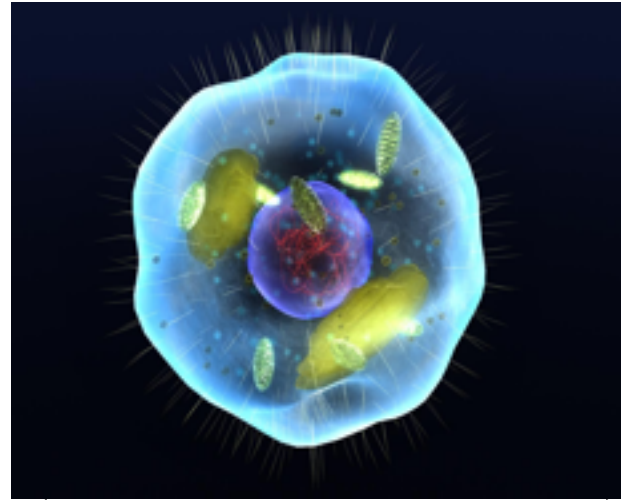
En subclass *specialiserar* ofta metoderna som är ärvda från superklassen (overriding)

Unified Modeling Language (UML)

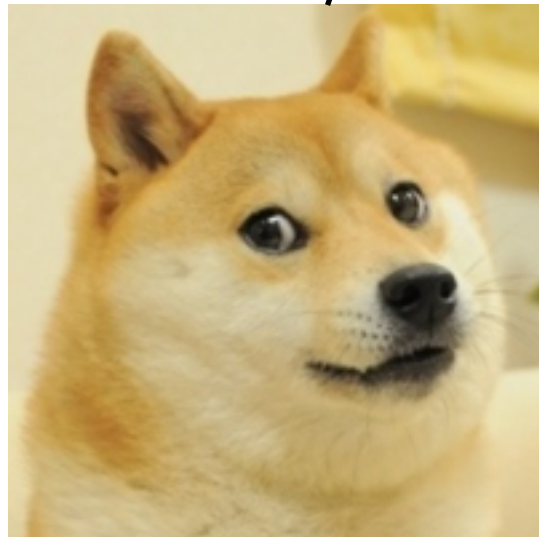
Klasshierarkier kan ritas ut som *klassdiagram* i *UML*.



Animal



is_a



Dog

is_a



Cat



Overriding (Metodspecialisering)

```
abstract class Animal {  
    void makeSound() { System.out.println("Not implemented yet!"); }  
}  
  
class Cat extends Animal {  
    void makeSound() { System.out.println("Meeow!"); }  
}  
  
class Dog extends Animal {  
    void makeSound() { System.out.println("Wuff!"); }  
}  
  
class Animain {  
    public static void main(String[] args) {  
        Animal[] animals = { new Cat(), new Dog(), new Cat() };  
        for (Animal a : animals)  
            a.makeSound();  
    }  
}
```



Overriding exempel 2

```
abstract class Animal {  
    String name = null;  
  
    void setName(final String name) { this.name = name; }  
  
    String getName() { return name; }  
}  
  
class Cat extends Animal { ... }  
  
class Dog extends Animal {  
    String[] okDogNames = { "Amanda", "Prima" , "Tassie" };  
  
    void setName(final String name) {  
        for (String ok : okDogNames) {  
            if (ok.equals(name)) { super.setName(name); return; }  
        }  
    }  
}
```



En bättre hund

```
class Dog extends Animal {  
    String[] okDogNames = { "Amanda", "Prima" , "Tassie" };  
  
    void makeSound() { System.out.println("Wuff!"); }  
  
    boolean isOkName(final String name) {  
        for (String ok : okDogNames) {  
            if (ok.equals(name)) { return true; }  
        }  
        return false;  
    }  
  
    void setName(final String name) {  
        if (isOkName(name)) super.setName(name);  
    }  
}
```



Overriding av abstrakta metoder

```
abstract class Animal {  
    String name = null;  
  
    void setName(final String n) { if (isOkName(n)) this.name = n; }  
  
    abstract boolean isOkName(final String name);  
}  
  
class Cat extends Animal {  
    boolean isOkName(final String name) { return true; }  
}  
  
class Dog extends Animal {  
    String[] okDogNames = { "Amanda", "Prima" , "Tassie" };  
  
    boolean isOkName(final String name) {  
        for (String ok : okDogNames) {  
            if (ok.equals(name)) { return true; }  
        }  
        return false;  
    }  
}
```



Exempel på inkapsling och arv

```
abstract class Animal {
    String name = null;

    void setName(final String n) { if (isOkName(n)) this.name = n; }

    abstract boolean isOkName(final String name);
}

class Cat extends Animal {
    bool isOkName(final String name) { return true; }
}

class Dog extends Animal {
    String[] okDogNames = { "Amanda", "Prima" , "Tassie" };

    bool isOkName(final String name) {
        // cheat!
        this.name = "Captain Beefheart!";
    }
}
```



Exempel på inkapsling och arv

```
abstract class Animal {  
    private String name = null;  
  
    void setName(final String n) { if (isOkName(n)) this.name = n; }  
  
    abstract boolean isOkName(final String name);  
}  
  
class Cat extends Animal {  
    bool isOkName(final String name) { return true; }  
}  
  
class Dog extends Animal {  
    String[] okDogNames = { "Amanda", "Prima" , "Tassie" };  
  
    bool isOkName(final String name) {  
        // cheat!  
        this.name = "Captain Beefheart!";  
    }  
}
```



Fordonshierarkin i kod

```
public class Fordon {  
    public void print() {  
        System.out.print("Fordon");  
    }  
}  
  
public class Spark extends Fordon {}  
  
public class Cykel extends Fordon {  
    public void print() {  
        System.out.print("Cykel");  
    }  
}
```

```
public class Bil extends Fordon {  
    double vikt;  
  
    public Bil(double vikt) {  
        this.vikt = vikt;  
    }  
  
    public void print() {  
        System.out.print("Bil med vikt " + vikt);  
    }  
}
```

Användning

Koden

```
public static void pr(String s) { System.out.print(s); }

public static void main(String [] args) {
    Fordon f = new Fordon();
    Spark s = new Spark();
    Cykel c = new Cykel();
    Bil b = new Bil(1.2);
    pr("f: "); f.print(); pr("\n");
    pr("s: "); s.print(); pr("\n");
    pr("c: "); c.print(); pr("\n");
    pr("b: "); b.print(); pr("\n");
}
```

ger utskriften

f: Fordon

s: Fordon

c: Cykel

b: Bil med vikt 1.2

Flera subklasser

```
public class Personbil extends Bil {  
    int passagerare = 0;  
  
    public Personbil(double vikt) {  
        this(vikt, 5); // Anrop till den andra konstruktorn  
    }  
  
    public Personbil(double vikt, int passagerarAntal) {  
        super(vikt); // Anrop till superklassens konstruktör gör så här  
        passagerare = passagerarAntal;  
    }  
  
    public void print() {  
        System.out.print("Person");  
        super.print();  
        System.out.print(" och platsantal " + passagerare);  
    }  
}
```

Fortsättning av main

Koden

```
...
Personbil pb1 = new Personbil(1.5);
Personbil pb2 = new Personbil(0.9,4);
pr("pb1: "); pb1.print(); pr("\n");
pr("Vikt b : " + b.vikt + "\n");
pr("Vikt pb1: " + pb1.vikt + "\n");
f = b;
pr("f: "); f.print(); pr("\n");
f = pb1;
pr("f: "); f.print(); pr("\n");
pb1 = (Personbil) f;
pr("pb1: "); pb1.print(); pr("\n");
```

ger resultatet

```
pb1: PersonBil med vikt 1.5 och platsantal 5
Vikt b: 1.2
Vikt pb1: 1.5
```


f: Bil med vikt 1.2

f: PersonBil med vikt 0.9 och platsantal 4

pb1: PersonBil med vikt 0.9 och platsantal 4

Typomvandling

Typomvandling (cast) kan göras och förändrar kompilatorns syn på ett värdes typ (men inte värdet!).

```
pb1 = f;  
pb1 = (Personbil) c;  
pr("Vikt: " + f.vikt);  
pb1 = (Personbil) f; // om f ej refererar Personbil
```

Man kan fråga om ett värde har en viss typ med instanceof-operatör.

```
if (f instanceof Personbil) {  
    pb1 = (Personbil) f;  
} else {  
    ...  
}
```

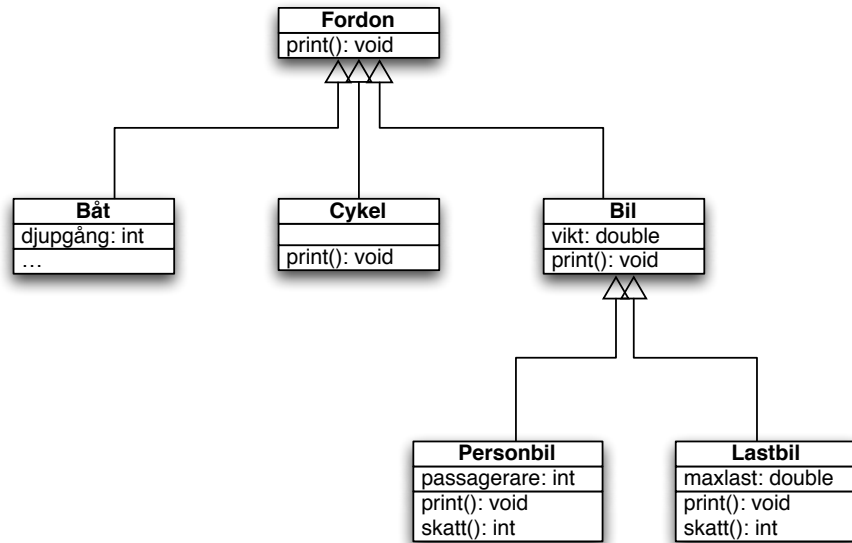
Operatör **instanceof** bör sällan användas! (Varför?)

Sammanfattning

Om klassen S är en subklass till klassen B och om s är deklarerad som en referens till S och b till B så gäller

- Ett objekt ur klassen S har alla attribut och metoder som klassen B har
- Om S deklarerar ett attribut som finns i B *överskuggas* B:s attribut.
- En referens får referera objekt ur deklarerad klass *och dess subklasser*.
- För att uttrycket b.x (eller b.x()) skall vara tillåten måste attributet (eller metoden) x finnas deklarerad i B eller någon superklass till B.
- När man anger ett attribut a via punktnotation p.a så är det hur p är *deklarerad* som avgör vilket attribut som väljs.
- När man anropar en metod m via p.m() så är det typen på det element som p verkligen refererar som avgör vilken metod som väljs (så kallad *dynamisk* eller *sen* bindning).

Metoder som bara finns i vissa klasser



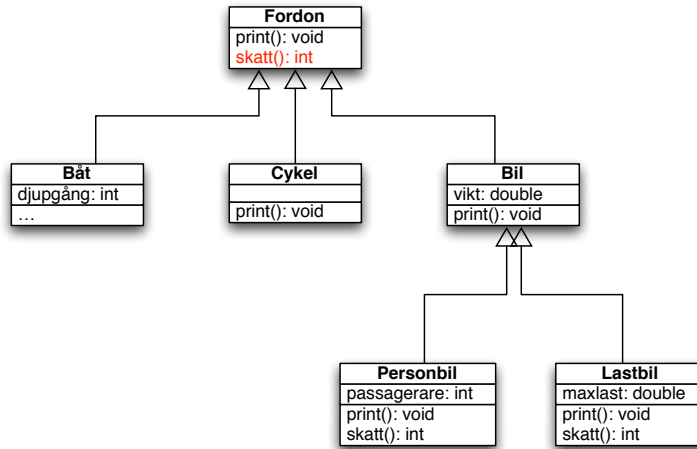
Bilar **beskattas** men beräkningen görs på olika sätt för person- och lastbilar.

Användning av skatt-metoden

```
Bil b = new Lastbil(3., 2.);  
...  
if (...) {  
    b = new Personbil(1.2, 5);  
}  
double s;  
  
s = b.skatt(); // Illegalt! Skatt inte finns i b  
  
s = ((Lastbil) b).skatt(); // Riskabelt!  
  
if (b instanceof Lastbil) { // OK men klumpigt och oflexibelt  
    s = ((Lastbil) b).skatt();  
} else if (b instanceof Personbil) {  
    s = ((Personbil) b).skatt();  
} else {  
    s = 0;  
}
```

Generalisera—skatt() åt alla

Definiera dessutom en skatt-metod i klassen Bil eller, troligen bättre, i klassen Fordon:



```
public class Fordon {
    public void print() {
        System.out.print("Fordon");
    }

    public double skatt() {
        return 0;
    }
}
```

Då kan metoden anropas för alla objekt av typen Fordon eller dess underklasser. (Test: är det vettigt att alla fordon kan beskattas?)

Generellt: Utnyttja den dynamiska bindningen istället för **instanceof**!

Konstruktorer

1. Metoder som bara får köras i samband med att ett objekt skapas
2. Måste anropas om de finns
3. Garanterar att objekten är valida så fort de blir "synliga" (=går att referera)
4. Ett tillfälle då överlagring inte är en bad smell

Konstruktorer: exempel

```
class Person {  
    String name;  
    Personnummer ssn;  
    Person(String name, Personnummer ssn) {  
        assert(name != null && ssn != null);  
        this.name = name;  
        this.ssn = ssn;  
    }  
}
```

```
    Person(String name, String ssn) {  
        assert(name != null && ssn != null);  
        this.name = name;  
        this.ssn = new Personnummer(ssn);  
    }  
}
```

...


```

class Personnummer { // Ignoring last four digits for brevity
    int year;
    int month;
    int day;

    Personnummer(int y, int m, int d) {
        setYear(y); // sets year if 0 <= y <= system year (or exception)
        setMonth(m); // sets month if 1 <= m <= 12 (or exception)
        setDay(d); // sets day if d is valid for current month (or exception)
    }

    // s has format YYYYMMDD-NNNN
    Personnummer(String s) {
        this(extractInt(s, 0, 4), extractInt(s, 4, 2), extractInt(s, 6, 2));
    }

    int extractInt(String s, int from, int length) {
        return Integer.parseInt(s.substring(from, length));
    }
}

```

Konstruktorer och arv

Konstruktorer ärvs *inte* (varför?)

När ett objekt ur en subklass skapas så sker följande:

1. Instansvariablerna får sina defaultvärden
2. En konstruktor för superklassen anropas. Man använder `super` för att specificera vilken av basklassens konstruktor som skall användas. Om `super` inte används anropas basklassens parameterlösa konstruktor som då måste finnas (implicit eller explicit).
3. Eventuella initieringsuttryck evalueras och tilldelas respektive instansvariabler
4. Satserna i subklassens konstruktor exekveras

Använd **super** i stället för att upprepa kod! Notera att **super**-anropet måste stå först i konstruktorn. (Varför?)

```
class Person {  
    String name;  
    Person(String name) { this.name = name; }  
}
```

```
class Student extends Person {  
    University uni;  
    Student(String name, University uni) {  
        super(name);  
        this.uni = uni;  
    }  
}
```

```
class PhDStudent extends Student {  
    Person advisor;  
    PhDStudent(String name, Person advisor) {  
        this(name, advisor.getUniversity(), advisor);  
    }  
    PhDStudent(String name, University uni, Person advisor) {  
        super(name, uni);  
        this.advisor = advisor;  
    }  
}
```

Javas klassbibliotek

Hela Java-miljön bygger på arv och klasshierakier

Exempel:

1. Klasserna för undantag: Throwable med underklasserna Error och Exception med underklasserna ...
2. Grafiska komponenter: Component med underklasser Button, Checkbox, Container, ... där t.ex. Container har underklasserna Panel, Window ...
3. Avbildningar: AbstractMap med bl.a. underklassen HashMap
4. Samlingsklasserna: Collection med bl a underklassen och List som bl a har underklasserna Vector och LinkedList

(Map, Collection och List är egentligen *interface* och inte klasser)

Rotklassen Object

I Java är klassen Object en s.k. rotklass – en superklass till alla klasser. En referens till Object får således referera vilket objekt som helst.

Kan utnyttjas för att göra generella ”samlingsklasser” (listor, tabeller ...):

```
class ListNode {
    Object info;
    ListNode next;
    ListHead(Object i, ListNode n) {
        info = i;
        next = n;
    }
}

public class List {
    ListNode head;
    public void insertFirst(Object o) {
        head = new ListNode(o, head)
    }
    ...
}
```

Från och med Java 1.5 används dock oftast ”generics” i stället.

Klassen Object forts

Några metoder i klassen Object:

- `Object clone()`
- **`boolean`** `equals(Object o)`
- `String toString()`

Ofta finns det anledning att omdefiniera dessa i en subclass.

Inget multipelt implementationsarv

Java stöder *inte* multipelt arv mellan klasser.

Låt A vara en klass som med metodssignaturen m och en instansvariabel i .

Låt B vara en klass som ärver A och har metodssignaturen m' och en instansvariabel i' .

Låt C vara en klass som ärver A och har metodssignaturen m'' och en instansvariabel i'' .

Låt D vara en klass som äver av *både* B och C .

Vad händer om $m \equiv m'$ eller $m \equiv m''$ eller $m' \equiv m''$?

Vad händer om $i \equiv i'$ eller $i \equiv i''$ eller $i' \equiv i''$?

```
D d = new D();  
d.m();  
d.i = ...;
```

Multipelt gränssnittsarv

Java stöder multipelt arv mellan interface.

Låt A vara ett interface med metodssignaturen m .

Låt B vara ett interface som ärver A med metodssignaturen m' .

Låt C vara ett interface som ärver A med metodssignaturen m'' .

Låt D vara ett interface som ärver av *både* B och C .

Om $m \equiv m'$ eller $m \equiv m''$ eller $m' \equiv m''$ uppstår ingen konflikt eftersom ingen *metodkropp* finns som kan skilja sig!

Samma-lika? Identitet och ekvivalens!

Täcks på en egen screencast. *Se den!*

```
String a = "Hello";  
...  
String b = "Hel" + "lo";  
...  
String lo = "lo";  
String c = "Hel" + lo;
```

```
if (a == b) { ... }  
if (a == c) { ... }  
if (b == c) { ... }
```

```
if (a.equals(b)) { ... }  
if (a.equals(c)) { ... }  
if (b.equals(c)) { ... }
```

Inkapsling och Arv

För en utökad diskussion om inkapsling, se screencast.

Åtkomstmodifikatorer

- **private** Endast åtkomligt från klassens egna metoder
- **public** Åtkomligt för alla
- **protected** Åtkomligt från egna subclasser och klasser i samma paket
- *package* (saknar nyckelord) Om ingen skyddsnivå anges så ges alla klasser i samma paket åtkomsträttighet

Som tumregel, använd alltid **private** – skydda superklassen från basklassernas beteende. Använd endast någon av de andra om det finns en vettig anledning (med minst ett bra exempel på användande).

Inkapsling och klassinvarianter

Först nu kan vi vara ”säkra” på att klassinvarianterna i Person och Personnummer inte bryts.

```
public class Person {  
    private String name;  
    private Personnummer ssn;  
  
    ...  
  
public class Personnummer {  
    private int year;  
    private int month;  
    private int day;  
    private int fourLast[4]; // does it leak?  
  
    int[] getFourLast() {  
        return fourLast;  
    }  
  
    ...  
}
```

Inkapsling och klassinvarianter – förbättrad

Först nu kan vi vara ”säkra” på att klassinvarianterna i Person och Personnummer inte bryts.

```
public class Person {  
    private String name;  
    private Personnummer ssn;
```

```
    ...
```

```
public class Personnummer {  
    private int year;  
    private int month;  
    private int day;  
    private int fourLast[4]; // does it leak? No it doesn't!
```

```
    int[] getFourLast() {  
        return fourLast.clone();  
    }  
    ...
```