Screencast

Arv

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 beräknar[1]

[1] Types 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)

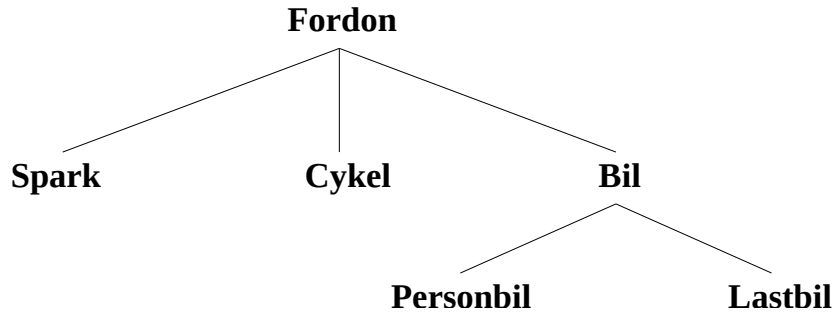
Arv

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 basklass för klasserna `Personbil` och `Lastbil`



Arv (forts)

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

- Klassen `Fordon` sägs vara en *basklass* (äv. 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:

$$B \textit{ är en } A$$

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

Arv (forts)

Ett objekt ur en subklass

– får automagiskt basklassens attribut och metoder och

– kan dessutom lägga till egna

Man säger att subklassen *ärver* basklassens egenskaper.

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

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

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

Metodspecialisering (overriding)

– Ponera en klass A och en klass B som är en subklass till A

– A definierar metoden **`int`** `getX()`

– B definierar också en metod **`int`** `getX()`

– Vi säger nu att B:s `getX()`-metod "override:ar" (är en specialisering av) A:s motsvarande metod

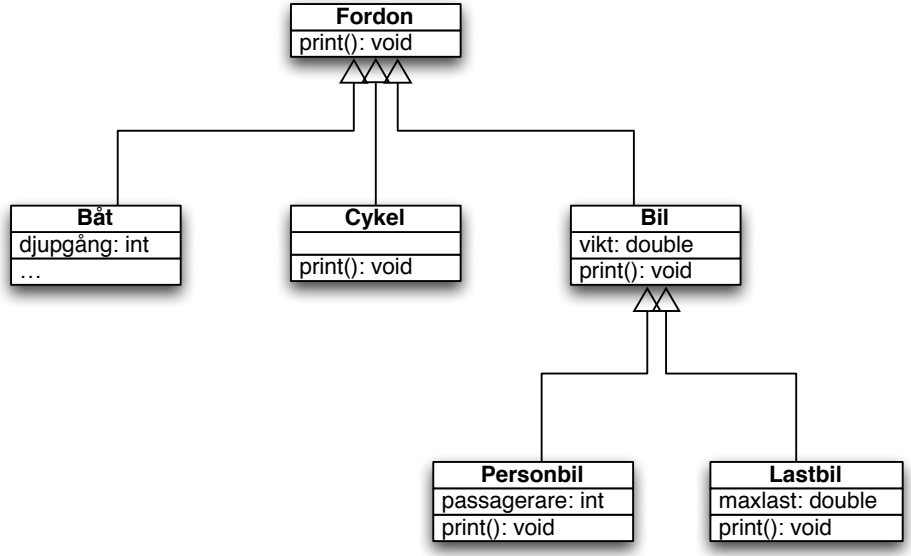
– En subklass kan explicit be om att anropa den *generella* versionen av en metod med nyckelordet **`super`**

Exempel:

```
class A {
  int x = 27;
  int getX() { return x; }
}

class B extends A {
  // Ärver int x automagiskt
  int getX() { return super.getX() + 2; }
}

A a = new A();
a.getX(); // returnerar 27

a = new B();
a.getX(); // returnerar 29

void someMethod(A a) { // Vad skrivs ut här?
  System.out.println("X-värde: " + a.getX());
}
```

Unified Modeling Language (UML)

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



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 konstruktor 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`-operatorn.

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

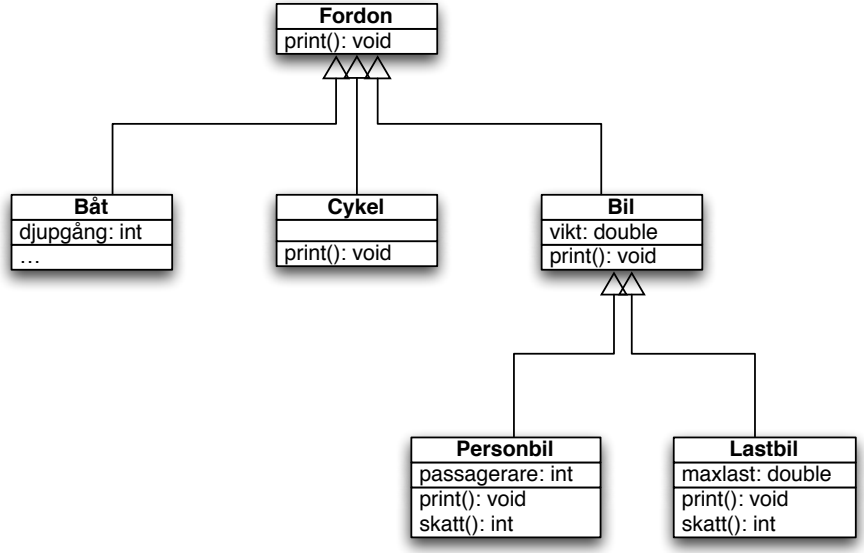
Operatorn **`instanceof`** bör sällan användas! (Varför?)

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



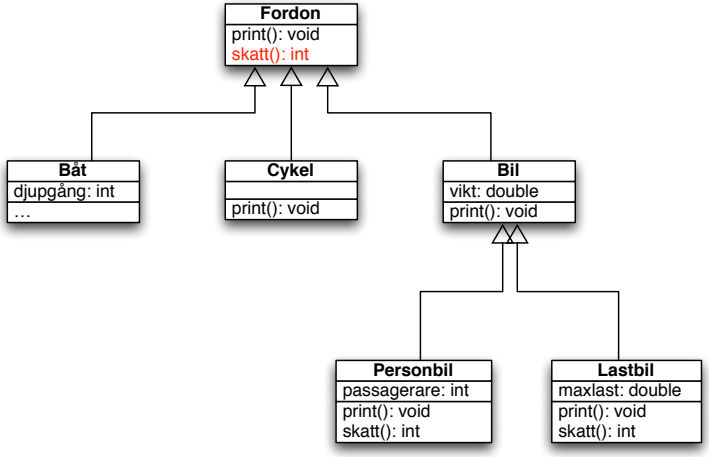
Bilar beskattas men beräkningen görs på olika sätt för person- och lastbilar.

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` är 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 = l;
      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 subklass.

Inget multipelt implementationsarv

Java stöder *inte* multipelt arv mellan klasser.

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

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

Låt C vara en klass som ärver A och har metodsignaturen 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 metodsignaturen m.

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

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

Låt D vara ett interface som äver 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 subklasser 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();
  }
  ...
```