Programmering för Språkteknologer II

Markus Saers
markus.saers@lingfil.uu.se
Rum 9-2040
stp.lingfil.uu.se/~markuss/ht09/pst2

Innehåll

Arv
Polymorfi
Abstrakta klasser
Gränssnitt
Iteratorer
Stackar

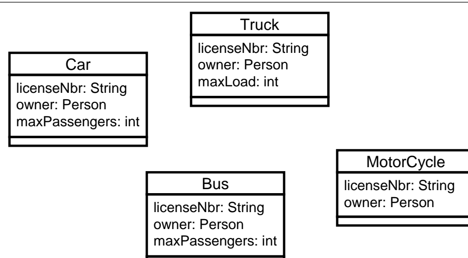
Arv

Exempel från kap. 9 i boken
Olika fordon
- `Car`, `Truck`, `Bus`, `MotorCycle`

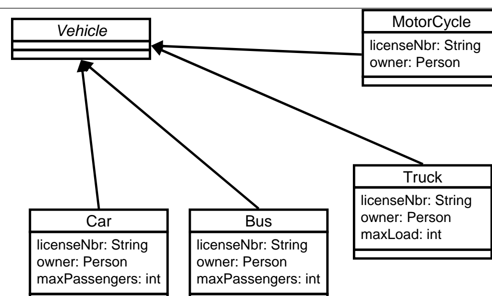
Fordon

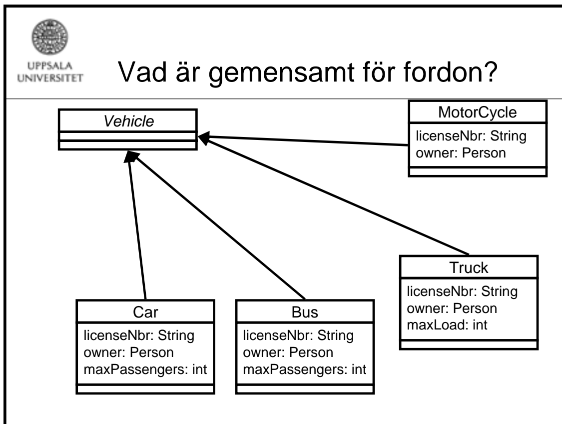
```
public class Car {
    private String licenseNbr;
    private Person owner;
    private int maxPassangers;
}
public class Truck {
    private String licenseNbr;
    private Person owner;
    private int maxLoad;
}
public class Bus {
    private String licenseNbr;
    private Person owner;
    private int maxPassengers;
}
public class MotorCycle {
    private String licenseNbr;
    private Person owner;
}
```

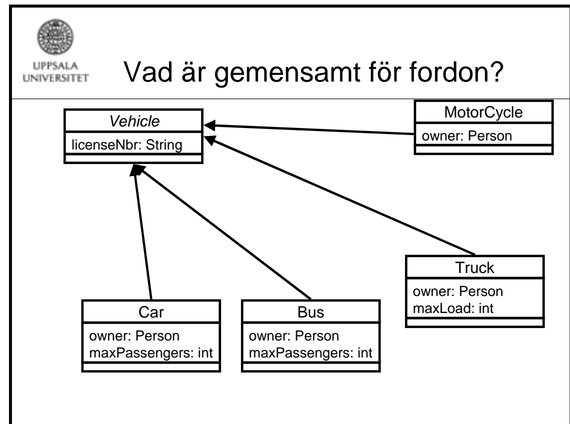
UML – Unified Modeling Language

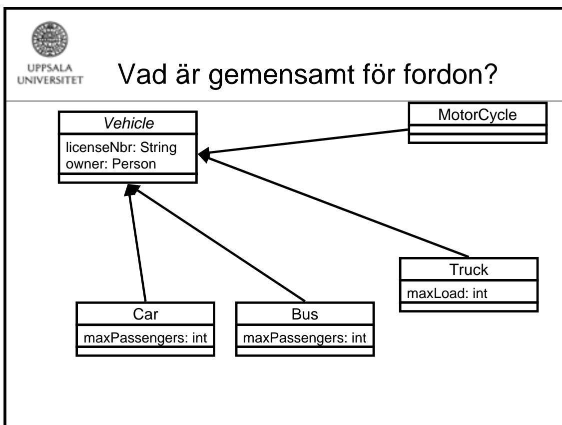


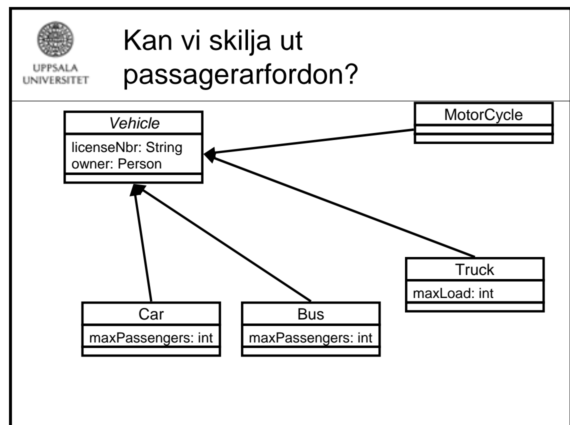
Hur kan vi säga att alla är fordon?

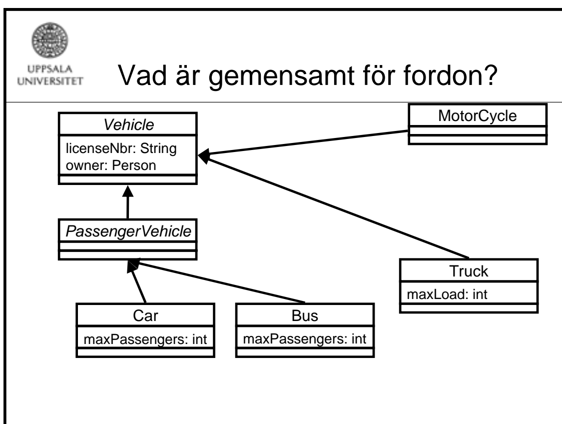


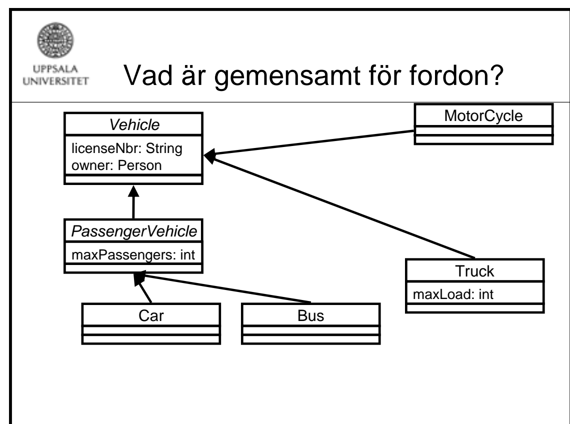

UPPSALA UNIVERSITET
Vad är gemensamt för fordon?
Vehicle
MotorCycle
licenseNbr: String
owner: Person
Truck
licenseNbr: String
owner: Person
maxLoad: int
Car
licenseNbr: String
owner: Person
maxPassengers: int
Bus
licenseNbr: String
owner: Person
maxPassengers: int


UPPSALA UNIVERSITET
Vad är gemensamt för fordon?
Vehicle
licenseNbr: String
MotorCycle
owner: Person
Truck
owner: Person
maxLoad: int
Car
owner: Person
maxPassengers: int
Bus
owner: Person
maxPassengers: int

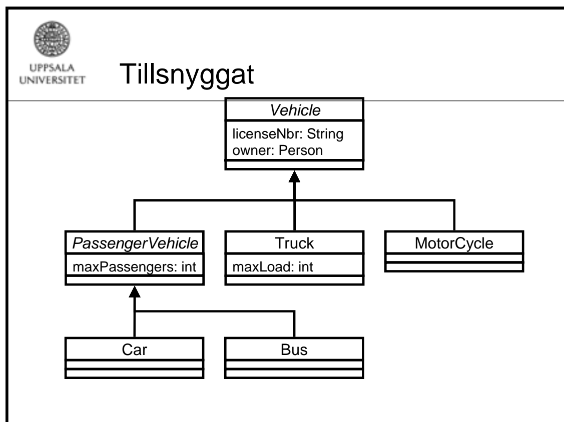

UPPSALA UNIVERSITET
Vad är gemensamt för fordon?
Vehicle
licenseNbr: String
owner: Person
MotorCycle
Truck
maxLoad: int
Car
maxPassengers: int
Bus
maxPassengers: int


UPPSALA UNIVERSITET
Kan vi skilja ut passagerarfordon?
Vehicle
licenseNbr: String
owner: Person
MotorCycle
Truck
maxLoad: int
Car
maxPassengers: int
Bus
maxPassengers: int


UPPSALA UNIVERSITET
Vad är gemensamt för fordon?
Vehicle
licenseNbr: String
owner: Person
MotorCycle
PassengerVehicle
Truck
maxLoad: int
Car
maxPassengers: int
Bus
maxPassengers: int


UPPSALA UNIVERSITET
Vad är gemensamt för fordon?
Vehicle
licenseNbr: String
owner: Person
MotorCycle
PassengerVehicle
maxPassengers: int
Truck
maxLoad: int
Car
Bus

Tillsnyggat



I Java?

- Klasser i kursivstil är abstrakta
  - Deklareras med ordet **`abstract`** i Java
- Pilen betyder att en klass ärver från den klass den pekar på
  - I Java säger man att klassen ”bygger ut” en existerande klass
  - Deklareras med order **`extends`**

Från början

```
public class Car {
  private String licenseNbr;
  private Person owner;
  private int maxPassangers;
}
public class Truck {
  private String licenseNbr;
  private Person owner;
  private int maxLoad;
}
public class Bus {
  private String licenseNbr;
  private Person owner;
  private int maxPassengers;
}
public class MotorCycle {
  private String licenseNbr;
  private Person owner;
}
```

Nu

```
public abstract class Vehicle {
  private String licenseNbr;
  private Person owner;
}
public abstract class PassengerVehicle
extends Vehicle {
  private int maxPassangers;
}
public class Car extends PassengerVehicle
{
}
public class Truck extends Vehicle {
  private int maxLoad;
}
public class Bus extends PassengerVehicle
{
}
public class MotorCycle extends Vehicle {
}
```

Fördelar med arv

- Förändringar kan göras där de hör hemma
  - Om vi ville lägga in ett beskattningsvärde på alla fordon kan vi göra det i `Vehicle`
- Modelleringen blir ”korrekt”
  - …eller har förutsättningen att bli det
  - Sekundärmetoder kan läggas högt upp i arvshierarkin
  - Abstrakta klasser kan bli en bra plattform att bygga vidare på
- Man kan spara några knapptryckningar
  - Egentligen inte relevant…

Sekundärmetoder?

- Metoder som beror på andra metoder
- Leder oss till abstrakta metoder, ärvda metoder och polymorfi
  - Polymorfi = mångformighet

Vi tittar på beskattning!

- Vad vill vi beskatta?
  - Antal passagerarplatser
    - Skatt per passagerarplats
  - Hög lastkapacitet
    - Inte aktuellt för småfordon

Skattesatser

- Passagerarplatsskatten
  - Första platsen: gratis
  - Därutöver: 10 kr/plats
- Lastkapacitetskatten
  - Under 760 kg: gratis
  - Därefter 10 öre/kg

I Java

```
    public abstract class Vehicle {
   private String licenseNbr;
   private Person owner;
   public double getPassengerTax() {
      double tax = (getMaxPassengers() - 1) * 10;
      if (tax < 0) tax = 0;
      return tax;
   }
   public double getLoadTax() {
      double tax = (getMaxLoad() - 760) * 0.1;
      if (tax < 0) tax = 0;
      return tax;
   }
   public double getTax() {
      return getPassengerTax() + getLoadTax();
   }
   public abstract int getMaxPassengers();
   public abstract int getMaxLoad();
}
```

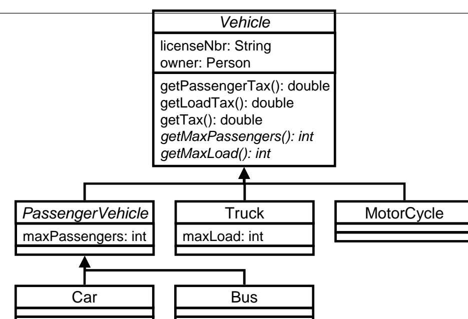
Vad har hänt?

- Vi kan räkna ut skatten för alla fordon
- Fordon är nu tvungna att implementera de två abstrakta metoderna
  - int getMaxPassengers()
  - int getMaxLoad()

Abstrakt/konkret

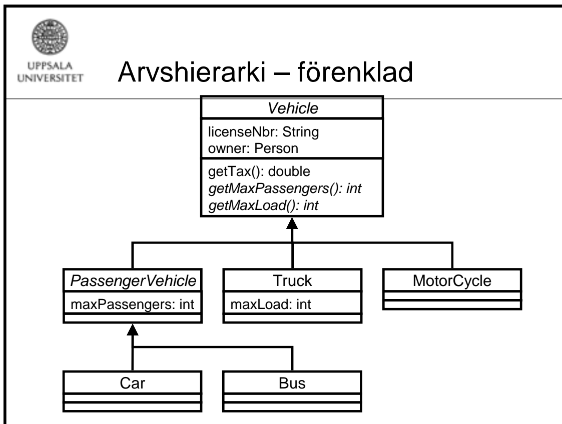
- Det går inte att skapa abstrakta objekt
- Så fort det finns abstrakta metoder måste klassen vara abstrakt
- Motsatsen till abstrakt är konkret
  - Abstrakt är det markerade fallet och deklareras med `abstract`
  - Konkret är normalfallet och behöver inte deklareras

Arvshierarki



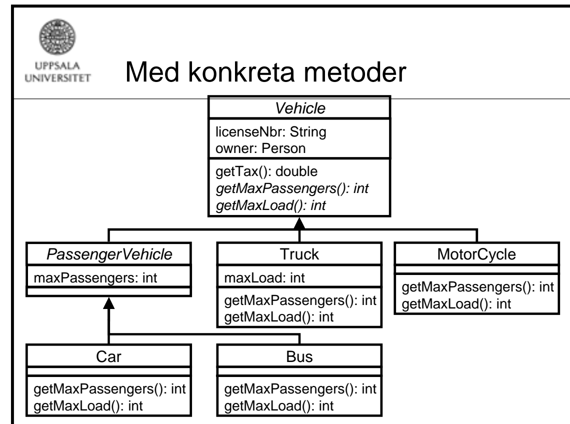
UPPSALA UNIVERSITET

Arvshierarki – förenklad



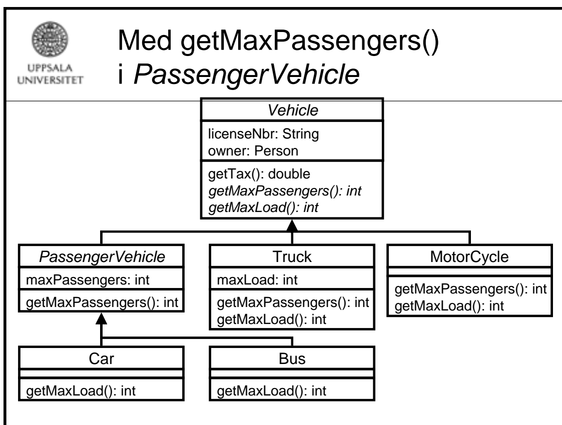
UPPSALA UNIVERSITET

Med konkreta metoder



UPPSALA UNIVERSITET

Med getMaxPassengers() i *PassengerVehicle*



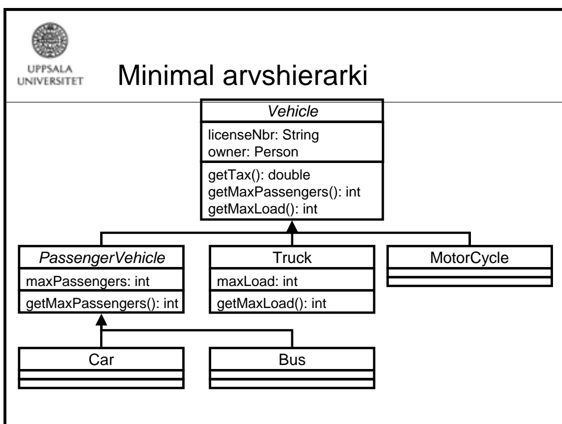
UPPSALA UNIVERSITET

Polymorfi 1: Närmast löven gäller!

```
    public abstract class Vehicle {
  private String licenseNbr;
  private Person owner;
  public double getPassengerTax() {
     return (getMaxPassengers() - 1) * 10;
  }
  public double getLoadTax() {
     return (getMaxLoad() - 760) * 0.1;
  }
  public double getTax() {
     return getPassengerTax() + getLoadTax();
  }
  public int getMaxPassengers() {
     return 1;
  }
  public int getMaxLoad() {
     return 1;
  }
}
```

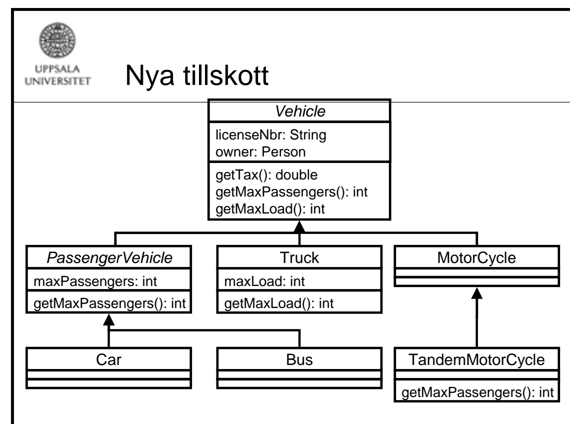
UPPSALA UNIVERSITET

Minimal arvshierarki



UPPSALA UNIVERSITET

Nya tillskott



UPPSALA UNIVERSITET

Vad finns i en bil?

Car	*PassengerVehicle*	*Vehicle*
	maxPassengers: int	licenseNbr: String owner: Person
	getMaxPassengers(): int	getTax(): double getMaxPassengers(): int getMaxLoad(): int

UPPSALA UNIVERSITET

Vad finns i en bil?

Car	*PassengerVehicle*	*Vehicle*
	maxPassengers: int	licenseNbr: String owner: Person
	getMaxPassengers(): int	getTax(): double getMaxPassengers(): int getMaxLoad(): int

UPPSALA UNIVERSITET

Vad finns i en bil?

Car	*PassengerVehicle*	*Vehicle*
	maxPassengers: int	licenseNbr: String owner: Person
getTax(): double getMaxPassengers(): int getMaxLoad(): int	getTax(): double getMaxPassengers(): int getMaxLoad(): int	getTax(): double getMaxPassengers(): int getMaxLoad(): int

UPPSALA UNIVERSITET

Motorklubbar

```
      import java.util.*;
public class MotorClub {
   private LinkedList<Vehicle> members;
   public MotorClub() {
      members = new LinkedList<Vehicle>();
   }
   public void add(Vehicle v) {
      members.add(v);
   }
         public static void main(String[] args)
{
      MotorClub kak = new MotorClub();
      kak.add(new Car(”KINGOFSWEDEN”, 4));
      kak.add(new Truck(”ABC321”, 2400));
            MotorClub ha = new MotorClub();
      ha.add(new MotorCycle(”HAH666”));
      ha.add(new MotorCycle(”HIH666”));
      ha.add(new MotorCycle(”HOH666”));
   }
}
```

UPPSALA UNIVERSITET

Polymorfi 2

Variabler och värden kan ha olika typer!

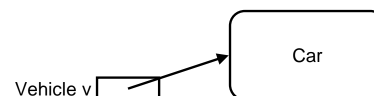
Värdet måste vara en specialisering av variabelns typ

```
LinkedList<Vehicle> a =
   new LinkedList<Vehicle>();
a.addLast(new Car(”AAA111”, 4));
System.out.println(a.getLast.getTax());
```

UPPSALA UNIVERSITET

Polymorfi 2

```
     Vehicle v = new Car(”AAA111”,
4);
```



Konstruktorer i arvshierarkier

```
      public abstract class Vehicle {
   private String licenseNbr;
   private Person owner;
   public Vehicle(String l, Person o) {
      licenseNbr = l;
      owner = o;
   }
}
      public abstract class PassengerVehicle extends
Vehicle {
   private int maxPassangers;
   public PassengerVehicle(String l, Person o, int mp) {
      super(l, o);
      maxPassengers = mp;
   }
}
      public class Car extends PassengerVehicle {
   public Car(String l, Person o, int mp) {
      super(l, o, mp);
   }
   public Car(String l, Person o) {
      this(l, o, 4);
   }
}
```

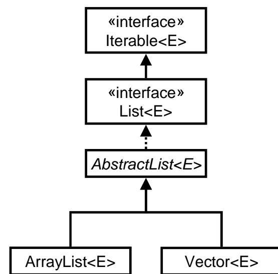
Abstrakta klasser och gränssnitt

Vad ska man använda och när?

Fallstudie: Listor i **`java.util`**

- **`Iterable<E>`** (gränssnitt)
- **`List<E>`** (gränssnitt)
- **`AbstactList<E>`** (abstrakt klass)
- **`ArrayList<E>`** (klass)
- **`Vector<E>`** (klass)

Listor



Listor

Gränssnittet **`List`** avgör vad listor ska klara av

Den abstrakta klassen **`AbstractList`** ger implementationer för de flesta metoderna i **`List`** utom de mest grundläggande

Exakt hur de grundläggande funktionerna implementeras avgörs av specialiseringarna av **`AbstractList`**

- **`ArrayList`**
- **`Vector`**

Bra uppdelning gränssnitt/abstrakt klass

Frågor?

Varför är listor intressanta?

Vi kan ha en godtycklig mängd element av en viss typ samlade på ett ställe

Vi kan behandla dem en och en

- Vi kan vara säkra på att ”besöka” varje element i listan exakt en gång
- Gränssnittet **`Iterable`**

Iterable<T>

Tillhandahåller en **Iterator<T>** – ett objekt som kan användas för att löpa igenom en samling element

Klasser som implementerar gränssnittet måste ha metoden
Iterator<T> iterator()

Iterator<T>

Innehåller allt som behövs för att löpa igenom och ändra en samling värden

```
T next()
boolean hasNext()
void remove()
```

Vi fokuserar på **next()** och **hasNext()**

Loopar igen

Grundläggande form:

```
String[] v = {"a", "b", "c"};
int i = 0;
while (i < v.length) {
   String s = v[i];
   // Gör något med s
   i++;
}
```

Loopar

- Tre komponenter
  - Initialisering (peka på första elementet)
  - Steg (avancera till nästa element)
  - Stoppvillkor (finns det fler element?)
- Använder ett index
  - Ofta inte intressant i sig när man löper igenom en vektor
  - Värdet i vektorn mer intressant

Iterator

- Tre komponenter
  - Initialisering
    - konstruktor
  - Steg (ta ut aktuellt värde och avancera)
    - **next()**
  - Stoppvillkor
    - hasNext()

Loopar över samlingar

```
    List<String> v =
  new ArrayList<String>();
// v fylls med värden

Iterator<String> i = v.iterator();
while (i.hasNext()) {
   String s = i.next();
   // Gör något med s
}
```

Kortform

```
    List<String> v =
  new ArrayList<String>();
// v fylls med värden

for (String s : v) {
   // Gör något med s
}
```