Stockholms universitet

Programmering 2

Föreläsning 1: Objektorienterad programmering

Isak Samsten, VT22

Stockholms universitet

Introduktion

2022-03-22 /Isak Samsten

Stockholms universitet

Primitiva värden och objekt

- I Java finns både primitiva värden:
  - `int`, `double`, `long`, `char` osv
- och referensvärden:
  - Arrayer och objekt

Stockholms universitet

Värdesemantik

- Beteende där värden kopieras när de tilldelas, skickas som parametrar eller returneras
  - Alla primitiva värden i Java följer värdesemantik
  - När en variabel tilldelas kopieras dess värde
  - Modifikation påverkar inga andra värden

Stockholms universitet

Exempel: primitiva värden

```
public class Test {

    public static void main(String[] args) {
        int a = 10;
        int b = 11;
        swap(a, b);
        System.out.println(a + " " + b);
    }

    private static void swap(int a, int b) {
        int tmp = a;
        a = b;
        b = tmp;
    }
}
```

Stockholms universitet

Referenssemantik

- Beteende där variabler inte sparar objektet utan para en ”pekare” till objektet
  - När en variabel tilldelas en annan variabel ändras enbart pekaren och inte objektet
  - Båda variablerna pekar på samma objekt
  - Modifikation av den ena pekaren påverkar den andra

Stockholms universitet

Exempel: referensvärden

```
public static void main(String[] args) {
    int[] a = new int[] {10};
    int[] b = new int[] {11};
    swap(a, b);
    System.out.println(a[0] + " " + b[0]);
}

private static void swap(int[] a, int[] b) {
    int tmp = a[0];
    a[0] = b[0];
    b[0] = tmp;
}
```

Stockholms universitet

Klasser och objekt

- Klasser agerar ”ritning” för hur objekts skapas
- Objekt ”kapslar in” data och beteende

Stockholms universitet

Objekt

- Objekt är entiteter som kapslar in (”encapsulate”)
  - Data: variabler i objektet
  - Beteenden: metoder i objektet
- Man interagerar med metoder som ändrar och hämtar data inne i objektet
- Konstruktor: `Object o = new Object();`
- Anropa metoder: `o.hashCode();`

Stockholms universitet

Klasser

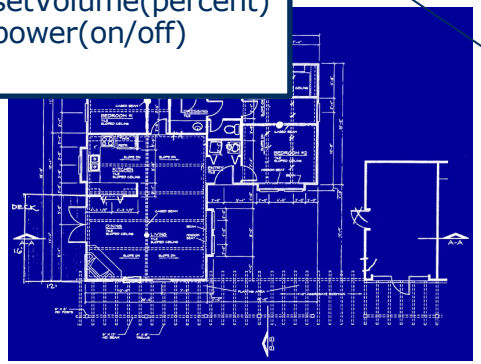
- Klasser representerar (i Java):
  - Program (om det finns en main-metod)
  - En ”ritning” för nya objekt
- Objektorienterad programmering (OOP)
  - Program som utför sitt beteende genom att **objekt** interagerar
  - Abstraktion: objekt och klasser agerar som abstraktioner genom att separera koncept och (implementations)detaljer

Ritningsanalogi

Smartphone (klass)
Data
- currentCall
- volume
- batteryLevel

Beteende
- makeCall(to)
- setVolume(percent)
- power(on/off)

Skapar



Smartphone #1
Data
- currentCall = ”wife”
- Volume = 24
- batteryLevel = 33

Beteende
- makeCall(to)
- setVolume(percent)
- power(on/off)

Smartphone #2
Data
- currentCall = ”friend”
- Volume = 88
- batteryLevel = 99

Beteende
- makeCall(to)
- setVolume(percent)
- power(on/off)

Smartphone #3
Data
- currentCall = ”SOS”
- Volume = 50
- batteryLevel = 0.2

Beteende
- makeCall(to)
- setVolume(percent)
- power(on/off)

Stockholms universitet

Ritningsanalogi

- Klassen (dvs. ritningen) beskriver hur objekt skapas
- Varje objekt har sin egen ”kopia” av data (variabler) och metoder
  - Metoderna ändrar enbart sitt objekts data

Stockholms universitet

Program

- Ett program som använder klasser för att skapa objekt. **`Test`** är en program som använder objekt av klassen **`ArrayList`**

```
public class Test {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("hello");
    }
}
```

Stockholms universitet

Instansvariabler

- En instansvariabel är en variabel som är del av ett objekts tillstånd
  - Varje objekt har en egen kopia av varje instansvariabel

Inkapsling

- Genom att dölja implementationsdetaljer från användaren av en klass krävs abstraktion
  - Skiljer den interna vyn (instansvarialer) från den externa vyn
  - ”Skyddar” objektets data

```
class TextBox {
    private String text;
    public String getText() {
        return text;
    }
}
```

Samma vy,

men olika data

```
class TextBox {
    private char[] text;
    public String getText() {
        return String.valueOf(text);
    }
}
```

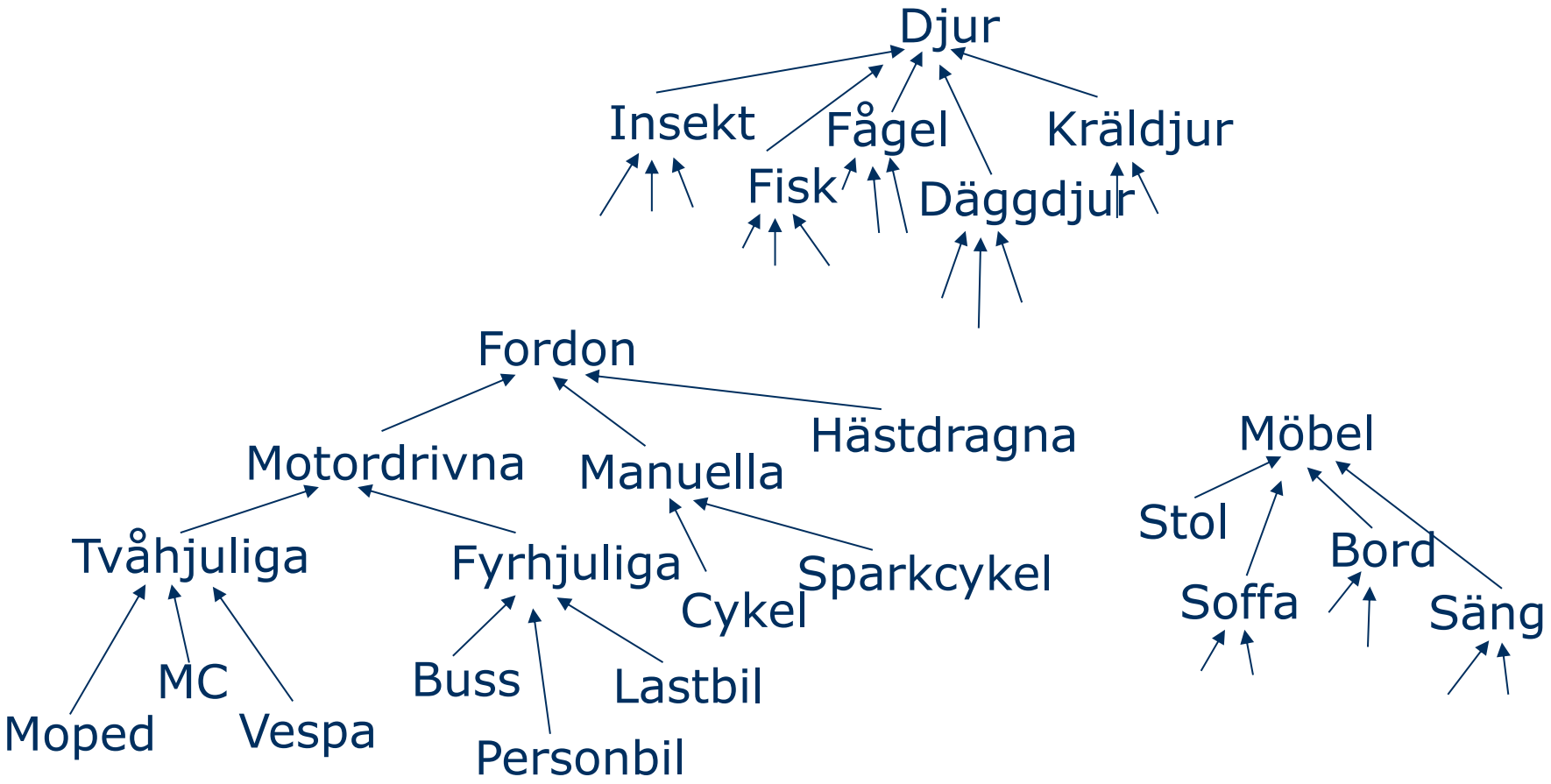
Stockholms universitet

Fördelar med inkapsling

- Abstraktion mellan objekt och användare
- Skyddar intern data från att användas utan ”lov”
- Vi kan ändra implementation men ha kvar den externa vyn
- Vi kan kontrollera att ett objekt uppfyller vissa villkor: en `TextBox` får ha max 142 tecken

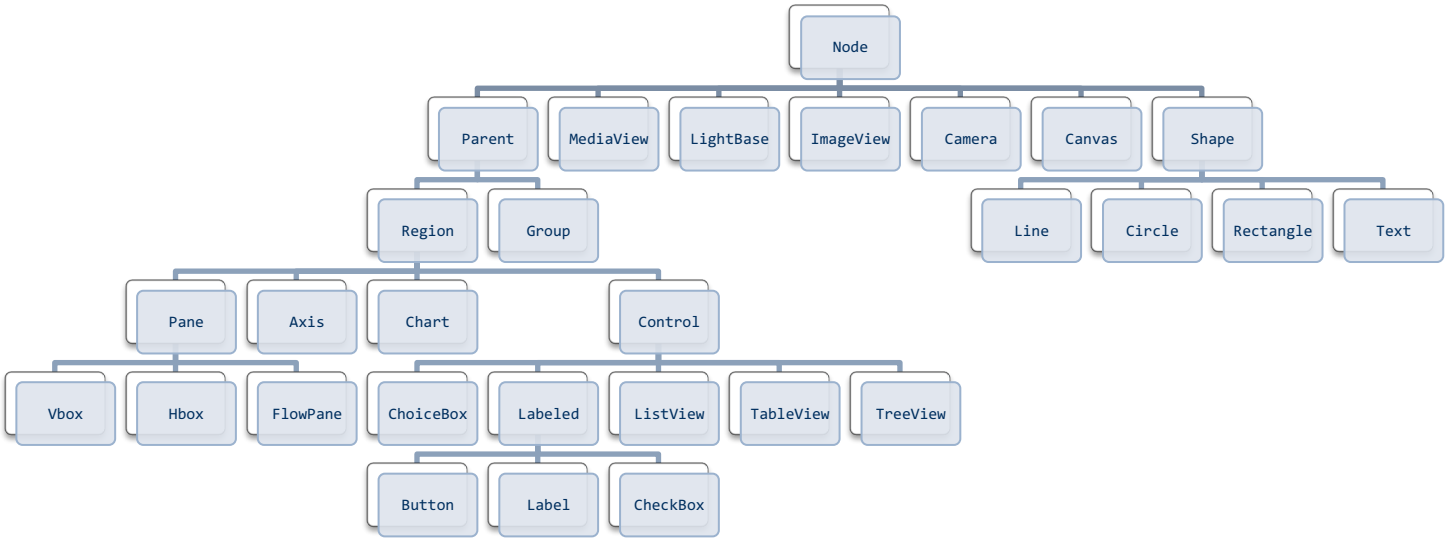
Begreppshierarkier

Stockholms universitet

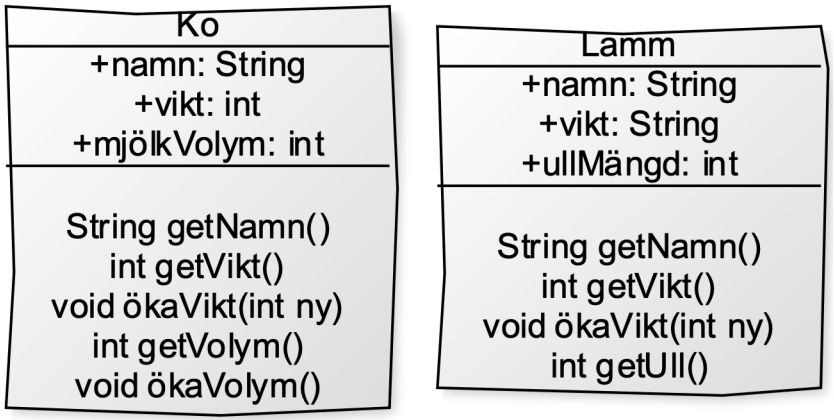


Stockholms universitet

Del av `Node`-hierarkin i JavaFX



Separata klasser



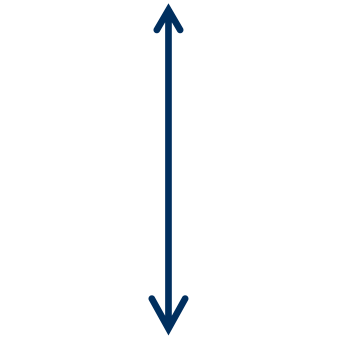

```
Ko rosa = new Ko("Rosa", 400, 25);
Lamm lambi = new Lamm("Lambi", 77, 4);
```

```
rosa: klass Ko
namn: ”Rosa”
vikt: 400
mjölkVolym: 25
```

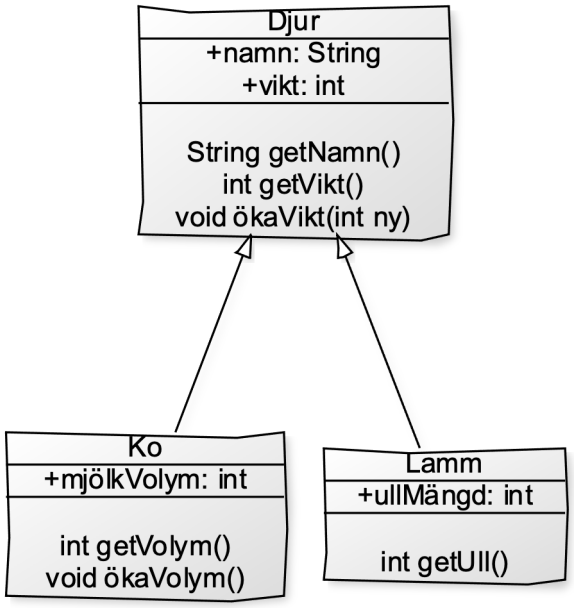
```
lambi: klass Lamm
namn: ”Lambi”
vikt: 77
ullMängd: 4
```

Klasshierarki

Generalisering



Specialisering



rotklass (högst upp)
superklass
basklass
supertyp

is-a
är-en
extends

subklass
härledd klass
lövklass (längst ner)

Arv i Java

```java
public class Djur {
    private String namn;
    private int vikt;
    public String getNamn() {
        return namn;
    }
    public int getVikt() {
        return vikt;
    }
    public void ökaVikt(int x) {
        vikt += x;
    }
}
```

Lamm **är-ett** Djur

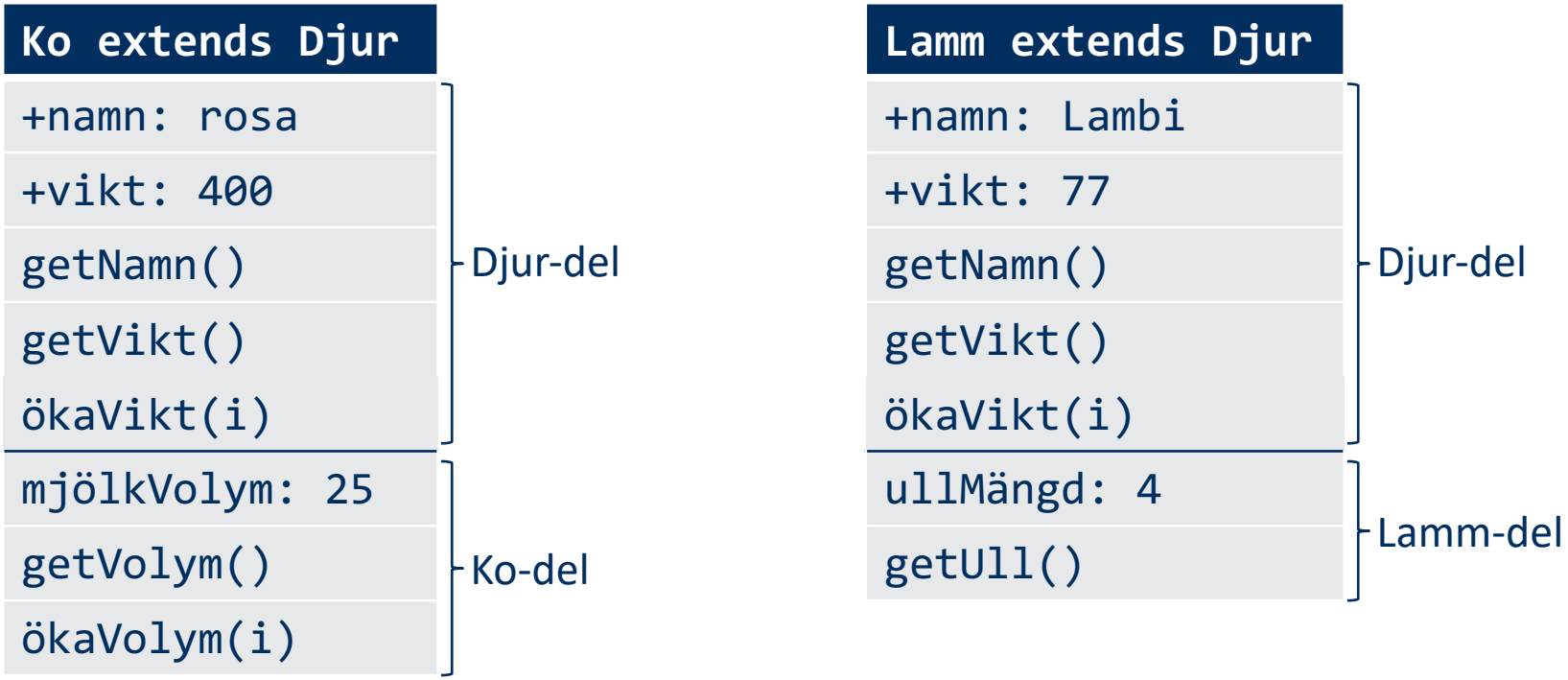
Ko **är-ett** Djur

```java
class Lamm extends Djur {
    private int ullMängd;
    public int getUll() {
        return ullMängd;
    }
}
```

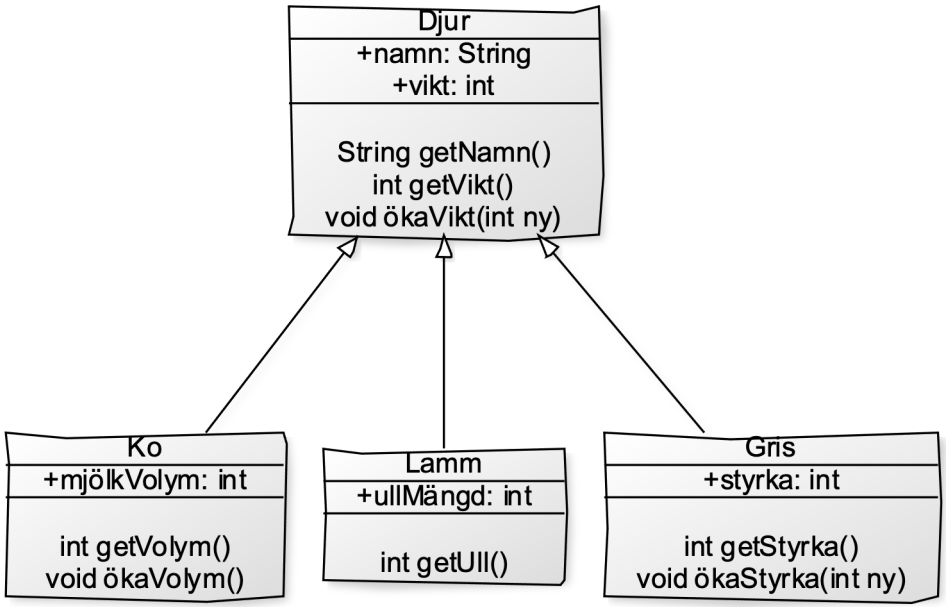
```java
class Ko extends Djur {
    private int mjölkVolym;
    public int getVolym() {
        return mjölkVolym;
    }
    public void ökaVolym(int x) {
        mjölkVolym += x;
    }
}
```

```
Ko rosa = new Ko("Rosa", 400, 25);
Lamm lambi = new Lamm("Lambi", 77, 4);
```

Stockholms universitet



Vi kan enkelt lägga till fler klasser

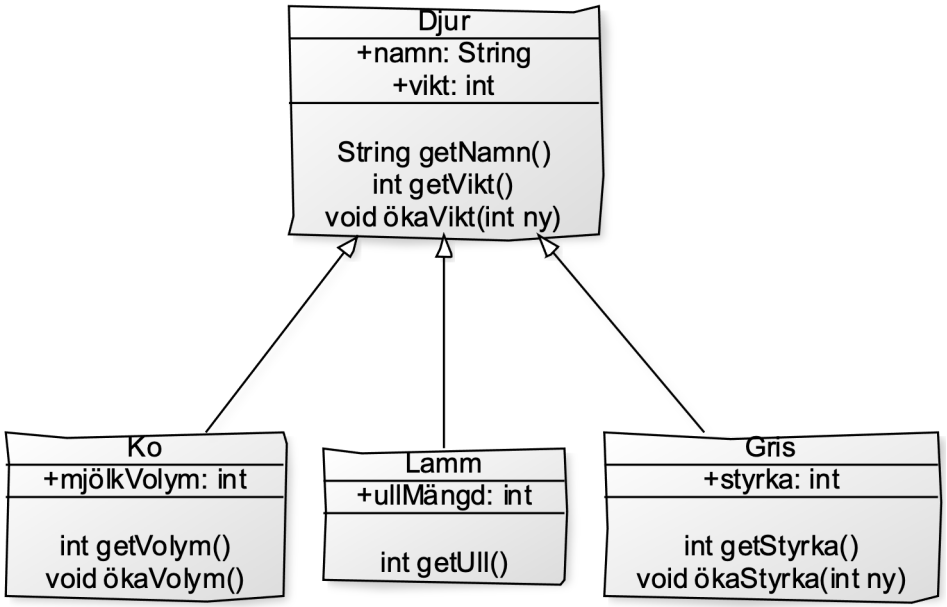



```java
class Gris extends Djur {
    private int styrka;
    public int getStyrka() {
        return styrka;
    }

    public void ökaStyrka(int x) {
        styrka += x;
    }
}
```

Stockholms universitet

Referenstyper

- Nu finns flera olika referenstyper:
  - `Ko`: kan referera till Ko-objekt
  - `Lamm`: kan referera till Lamm-objekt
  - `Gris`: kan referera till Gris-objekt
  - `Djur`: kan referera till alla subklasser av `Djur`: `Ko`, `Lamm` och `Gris` eller bara `Djur`
- Alla subklasser (av exempelvis Djur) innehåller alltid en del av superklassen:
  - Ex. en Ko är alltid ett Djur och har en djur-del




```
Ko ko = new Ko("Rosa", 117, 3);
Lamm lamm = new Lamm("Bäbä", 37);
Gris gris = new Gris("Edgar", 88, 21);
```

```
Djur d1, d2, d3;
d1 = new Ko("Rosa", 117, 3);
d2 = new Lamm("Bäbä", 37);
d3 = new Gris("Edgar", 88, 21);
```

Stockholms universitet

Samlingar av subklasser

```
ArrayList<Djur> allaDjur = new ArrayList<>();
allaDjur.add(new Ko("Rosa", 117, 3));
allaDjur.add(new Lamm("Bäbä", 37));
allaDjur.add(new Gris("Edgar", 88, 21));
allaDjur.add(new Ko("Blända", 195, 4));

Djur[] djurArr = new Djur[25];
djurArr[0] = new Ko("Blända", 195, 4);
djurArr[1] = new Gris("Edgar", 88, 22);
```

```
Ko ko = new Ko("Rosa", 117, 3);
Lamm lamm = new Lamm("Bäbä", 37);
Gris gris = new Gris("Edgar", 88, 21);
```

```
Djur d1, d2, d3;
d1 = new Ko("Rosa", 117, 3);
d2 = new Lamm("Bäbä", 37);
d3 = new Gris("Edgar", 88, 21);
```

Referenstyp och objekttyp

- Referenstyp (statisk) och objektstyp (dynamisk)
  - `i = d1.getVolym(); // kompileringsfel!`
- Kompilatorn kontrollerar semantiken
- Referensen `d1` är (statiskt) deklarerad som `Djur`
  - Klassen Djur har ingen metod `getVolym()`
  - Alltså: kompileringsfel!
- Via en `Djur`-referens ”syns” alltså bara objektets djur-del

```
ArrayList<Djur> allaDjur = new ArrayList<>();
allaDjur.add(new Ko("Rosa", 117, 3));
allaDjur.add(new Lamm("Bäbä", 37));
allaDjur.add(new Gris("Edgar", 88, 21));
allaDjur.add(new Ko("Blända", 195, 4));
```

Stockholms universitet

Samlingar

- Definieras med referenstyp men kan innehålla subklasser av denne
- allaDjur har *referenser* till olika Djur men vid iteration har kan vi bara ”se” djur-delen

```
for (Djur djur : allaDjur) {
    System.out.println(djur.getNamn());
    // vi kan ju inte anropa
    // djur.getVolym(), för vi vet inte
    // om djur är en Ko
}
```

```java
Ko ko = new Ko("Rosa", 117, 3);
Lamm lamm = new Lamm("Bäbä", 37);
Gris gris = new Gris("Edgar", 88, 21);
```

```java
Djur d1, d2, d3;
d1 = new Ko("Rosa", 117, 3);
d2 = new Lamm("Bäbä", 37);
d3 = new Gris("Edgar", 88, 21);
```

```java
public static String getDjurString(Djur djur) {
    return djur.getNamn() + ": " + djur.getVikt();
}
```

```java
public static String getKoString(Ko ko) {
    return ko.getNamn() + " " + ko.getVolym();
}
```

Referenstyper kontrolleras vid kompilering

- Vi kan alltid ersätta en superklass med en subklass:
  - `d2 = gris;`
  - `getDjurString(d1);`
  - `getDjurString(lamm);`
- Motsatsen gäller dock inte:
  - `gris = d3; // kompileringsfel`
  - `getKoString(d1); // kompileringsfel`

```
Ko ko = new Ko("Rosa", 117, 3);
Lamm lamm = new Lamm("Bäbä", 37);
Gris gris = new Gris("Edgar", 88, 21);
```

```
Djur d1, d2, d3;
d1 = new Ko("Rosa", 117, 3);
d2 = new Lamm("Bäbä", 37);
d3 = new Gris("Edgar", 88, 21);
```

```
public static String getDjurString(Djur djur) {
    return djur.getNamn() + ": " + djur.getVikt();
}
```

```
public static String getKoString(Ko ko) {
    return ko.getNamn() + " " + ko.getVolym();
}
```

Typkovertering

- Vi kan dock ”konvertera” eller ändra vy på en referenstyp – det kallas att *casta.*
- Exempel: om vi vet att `d1` i ett givet ögonblick pekar mot ett objekt av klassen `Ko` kan vi ändra referensens vy:
  - `Ko k = (Ko) d1;`
  - `k.getVolym();`
  - `((Ko) d1).getVolym();`
- Om referensen **inte** är en av typen som vi ändrar vy till får vi ett fel vid körningen av programmet.
  - `Ko k = (Ko) d2;`
  - Det kompilerar dock utan fel!

Stockholms universitet

Säker typkonvertering

- Ett objekts typ kan testas med relationsoperatorn **`instanceof`**
  - `referensuttryck` **`instanceof`** `KlassNamn`
- Exempelvis:

```
for (Djur djur : allaDjur) {
    if (djur instanceof Ko) {
        ((Ko) djur).ökaVolym(10);
    } else if (djur instanceof Gris) {
        ((Gris) djur).ökaStyrka(10);
    } else {
        djur.ökaVikt(10);
    }
}
```

Stockholms universitet

Konstruktorer

- En subklass konstruktor **måste** anropa super-klassens konstruktor det första som händer.
  - Görs med `super(…)`

```
class Djur{
    private String namn;
    private int vikt;
    public Djur(String namn, int vikt){
        this.namn = namn;
        this.vikt = vikt;
    }
}
```

```
class Ko extends Djur {
    private int mjölkVolym;
    public Ko(String n, int v, int m) {
        super(n, v);
        mjölkVolym = m;
    }
}
```

Implicit super-anrop

- Default konstruktorn är tom, dvs. gör inget

```java
public class Tom {
    private int i = 0;
    public Tom() { }
}
```

=

```java
public class Tom {
    private int i = 0;
}
```

```java
class Djur{
    private String namn;
    private int vikt;
    public Djur(String namn, int vikt){
        this.namn = namn;
        this.vikt = vikt;
    }
}
```

```java
class Gris extends Djur{
    private int styrka;
    public Gris(String n, int v, int styrka){
        super(n, v);
        this.styrka = styrka;
    }
}
```

Stockholms universitet

Polymorfism

- Möjligheten att använda samma kod men med olika objekt som har olika beteenden
- `System.out.println(Object o)` can print any type of object (using `Object#toString`)

```
for (Djur djur : allaDjur) {
    System.out.println(djur.getNamn());
}
```

Stockholms universitet

Rotklassen: `Object`

- Alla objekt i Java ärver av klassen `Object`
- Det innebär att alla objekt har metoderna
  - `hashCode()`
  - `toString()`
  - `equals(Object o)`
  - `getClass()`
  - Och några fler…
- Det är också därför `System.out.println` kan skriva ut alla typer av objekt – eftersom vi kan överskugga `toString()`

Stockholms universitet

Polymorfa parametrar

- Vi har redan sett ett exempel på polymorfa typ parametrar

```
public static String getDjurString(Djur djur) {
    return djur.getNamn() + ": " + djur.getVikt();
}
```

- Vi kan skriva ut namnet och vikten för olika typer av djur! Med samma kod!

```java
class Djur {
    private String namn;
    protected int vikt;
    public Djur(String namn, int vikt) {
        this.namn = namn;
        this.vikt = vikt;
    }
}
```

```java
class Ko extends Djur {
    private int mjölkVolym;
    public Ko(String n, int v, int m) {
        super(n, v);
        mjölkVolym = m;
    }
    public int getVikt() {
        return vikt + mjölkVolym;
    }
}
```

Stockholms universitet

Synlighetsmodifierare

- I java kan vi ha `public`, `private` och `protected` klasser och instansvariabler.
- `protected` anger att instansvariabler och metoder får användas av subklasser, men inte av utomstående objekt (om inte dessa finns i samma paket)

Stockholms universitet

Beräkna värdet av djur ☺

- Anta att vi vill kunna beräkna värdet av djur och skriva en metod `summaVärde(List<Djur> list)`
- Varje djurs värde beräknas annorlunda
  - Lamm: `ullMängd * ullMängd`
  - Gris: `vikt * 3 + styrka`
  - Ko: `vikt * mjölkVolym`
- Alla djur-subklasser ska alltså ha en metod `värde()`

```
public int värde() {
    // returnera värdet på djuret
}
```

```java
class Gris extends Djur{
    private int styrka;
    public Gris(String n, int v, int styrka){
        super(n, v);
        this.styrka = styrka;
    }
    public int värde() {
        return getVikt () * 3 + styrka;
    }
}
```

```java
class Ko extends Djur {
    private int mjölkVolym;
    public Ko(String n, int v, int m) {
        super(n, v);
        mjölkVolym = m;
    }
    public int värde() {
        return getVikt() * mjölkVolym;
    }
}
```

```java
class Lamm extends Djur {
    private int ullMängd;
    public Lamm(String namn, int ullMängd) {
        super(namn, 40);
        this.ullMängd = ullMängd;
    }
    public int getVärde() {
        return ullMängd * ullMängd;
    }
}
```

Stockholms universitet

Måste finnas i subklassen

- Metoden värde måste finnas i subklassen
- Eftersom värdet beräknas **olika** för de olika subklasserna

Beräkna summan av lista med djur

```
public static int summaVärde(ArrayList<Djur> allaDjur) {
    int summa = 0;
    for (Djur d : allaDjur) {
        if (d instanceof Ko) {
            summa += ((Ko) d).värde();
        } else if (d instanceof Lamm) {
            summa += ((Lamm) d).värde();
        } else if (d instanceof Gris) {
            summa += ((Gris) d).värde();
        } else {
            // oj här blev det fel!
        }
    }
    return summa;
}
```

- Fungerar men otroligt icke-flexibelt och icke-objektorienterad lösning!
- Alla våra subklasser har ju metoden värde – varför alla dessa test?
- Och värre: vi måste uppdatera koden om vi (eller någon annan) lägger till fler typer av djur! Det finns ett bättre sätt!

Överskuggning (polymorfism)

Stockholms universitet

- Vi deklarerar metoden värde i superklassen Djur
- När vi har en metod med exakt samma signatur (namn, parametrar och returvärde) i både superklass och subklass omdefinierar (överskuggar/override) subklassen superklassens metoden
- Nu är metoden deklarerad i superklassen men vi ger olika beteenden i subklasserna!

```
class Djur {
    private String namn;
    protected int vikt;
    public Djur(String namn, int vikt) {
        this.namn = namn;
        this.vikt = vikt;
    }
    public int värde() {
        return -1;
    }
}
```

Polymorfism eller dynamisk bindning

```
public static int summaVärde(ArrayList<Djur> allaDjur) {
    int summa = 0;
    for (Djur d : allaDjur) {
        summa += d.värde();
    }
    return summa;
}
```

Stockholms universitet

Överskuggning

- Bra! Nu kan vi enkelt lägga till fler Djur utan att vi behöver ändra implementationen när vi summera värdet på en lista med djur!
- Metoden som är implementerad så långt ner i arvshierarkin som möjligt anropas alltid
  - i vårt fall i Ko, Lamm och Gris.
- Vilken metod som anropas avgörs vid anropstillfället (och alltså inte när vi kompilerar koden – vi vet ju inte alla möjligt djur-implementationer).
  - Detta kallas dynamisk bindning eller polymorfism (mångformighet)
- Om vi inte överskuggar metoden i en subklass kommer Djur implementation användas
  - Problem: värdet blir -1!

Stockholms universitet

Utökning med ny klass

- Säg att vi vill lägga till en ny klass:

```
class Häst extends Djur{
    private int antalVinster;
    public int värde(){
        return antalVinster * getVikt();
    }
}
```

- Koden för att beräkna summan fungerar fortfarande
- Att på detta sätt lägga till nya klasser utan att behöva ändra i den gamla koden är något som gör objektorienterad programmering mycket kraftfullt och en stor fördel!

Abstrakta klasser och metoder

- Att implementera metoden `värde()` i klassen Djur att returnera `-1` är fel.
- Det vi vill göra är att deklarera att metoden måste finnas, men inte vad den gör
- Detta görs med abstrakta klasser och abstrakta metoder
- Abstrakta metoder deklareras utan implementation
  - Det tvingar subklasser att implementera metoden
- Endast abstrakta klasser kan ha abstrakta metoder

```
abstract class Djur {
    private String namn;
    protected int vikt;
    public Djur(String namn, int vikt) {
        this.namn = namn;
        this.vikt = vikt;
    }
    public abstract int värde();
}
```

Abstrakta klasser och metoder

Stockholms universitet

- Abstrakta metoder kan användas trots att de inte är implementerade!
  - Anropet binder dynamiskt till den subklass som implementerar värdet!
- Man kan inte skapa instanser av abstrakta klasser. Dvs. vi kan inte göra `new Djur(…) // kompileringsfel`
  - Metoden `värde()` är ju inte implementerad!

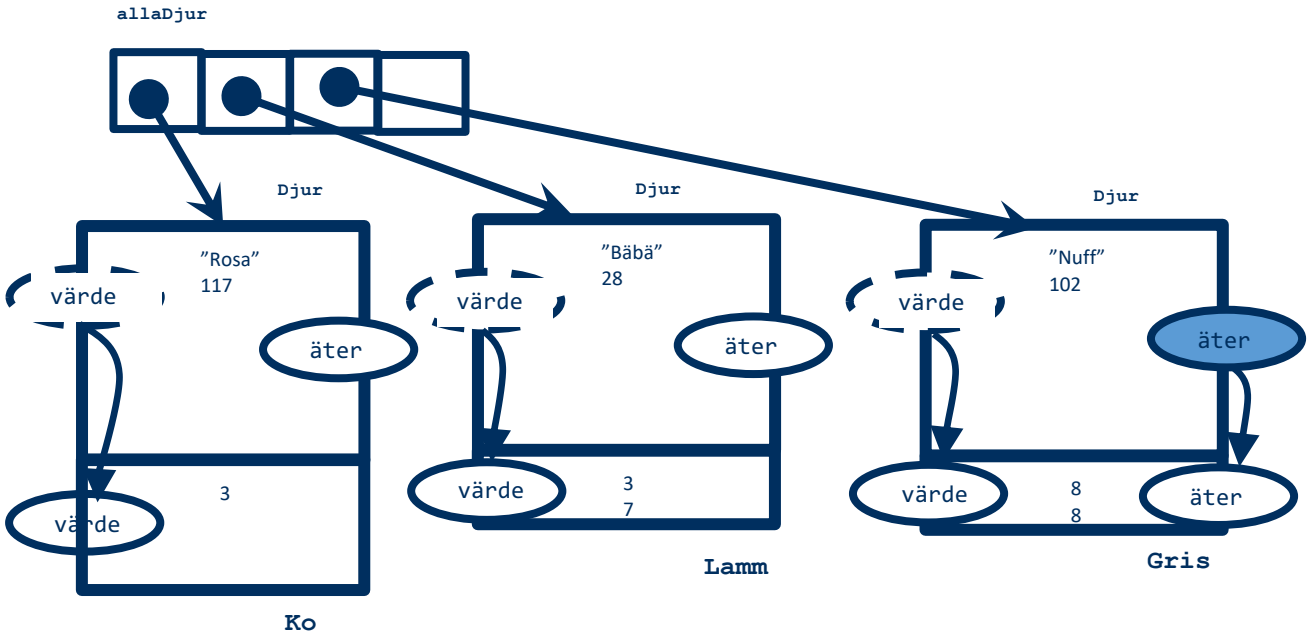
```
abstract class Djur {
    private String namn;
    protected int vikt;
    public Djur(String namn, int vikt) {
        this.namn = namn;
        this.vikt = vikt;
    }
    public abstract int värde();
    public String toString() {
        return namn + " " + vikt + " " + värde();
    }
}
```

Annan överskuggning

- Vi har ett standard beteende för djur
  - När de äter ökar deras vikt
- Men, vissa subklasser har olika beteenden:
  - När en Ko äter ökar också mjölkvolymen
- Subklasser som är nöjda med den metod som är deklarerad i superklassen behöver inte göra nåt.
  - Subklasser som vill göra på annat sätt kan överskugga metoden.

```
class Djur {
    private String namn;
    protected int vikt;
    public Djur(String namn, int vikt) {
        this.namn = namn;
        this.vikt = vikt;
    }
    public void äter(int mat) {
        this.vikt += mat;
    }
}
```

```
class Ko extends Djur {
    private int mjölkVolym;

    public Ko(String n, int v, int m) {
        super(n, v);
        mjölkVolym = m;
    }
    public void äter(int mat) {
        super.äter(mat);
        mjölkVolym++;
    }
}
```


allaDjur
Djur
”Rosa”
117
värde
äter
3
värde
Ko
Djur
”Bäbä”
28
värde
äter
värde
3
7
Lamm
Djur
”Nuff”
102
värde
äter
värde
8
8
äter
Gris

Stockholms universitet

Annoteringar

- I Java kan man annotera metoder, instansvariabler, klasser med mera
  - De skrivs som `@Annotering`
- Dessa annoteringar kan användas för att kompilatorn ska ge varningar eller genererar kod
- En användbar annotering är `@Override`
  - Den hjälper kompilatorn att kontrollera så att en metod verkligen överskuggar en annan metod

Stockholms universitet

Annoteringar: exempel

```
class Djur {
    private String namn;
    protected int vikt;
    public Djur(String namn, int vikt) {
        this.namn = namn;
        this.vikt = vikt;
    }
    public void äter(int mat) {
        this.vikt += mat;
    }
}
```

```
class Ko extends Djur {
    private int mjölkVolym;

    public Ko(String n, int v, int m) {
        super(n, v);
        mjölkVolym = m;
    }
    @Override
    public void äter(int mat) {
        super.äter(mat);
        mjölkVolym++;
    }
}
```

Returtyp hos överskuggade metoder

- Returtypen hos en överskuggade metod behöver inte vara samma som i superklassens metod
  - Den kan vara en subklass av returtypen
- Man kan också deklarera att en subklass returnerar en subtyp

```
abstract class Djur{
    public abstract Djur nyttBarn();
}
```

```
class Ko extends Djur{
    @Override
    public Ko nyttBarn() {
        return new Ko();
    }
}
```

```
class Gris extends Djur {
    @Override
    public Gris nyttBarn() {
        return new Gris();
    }
}
```

Stockholms universitet

Returtyp hos överskuggade metoder

- Om vi har ett objekt av typen Djur så kommer returtypen vara Djur (men konkret exempelvis en Ko eller en Gris)
- Om vi har en subklass kommer returtypen vara av den av subklassen deklarerade typen.

```
Ko ko1 = d.nyttBarn();    // kompileringsfel
Djur d1 = d.nyttBarn();
d1 instanceof Ko           // true
Ko ko2 = k.nyttBarn();
```

Förbjuda överskuggning

- En klass kan förbjuda att överskugga hela klassen eller bara vissa metoder

```
class Ko extends Djur {
    private int mjölkVolym;
    public Ko(String n, int v, int mv) {
        super(n, v);
        this.mjölkVolym = mv;
    }
    public final int getVolym() {
        return mjölkVolym;
    }
    public final void ökaVolym(int x) {
        mjölkVolym += x;
    }

}
```

Kompileringsfel

```
final class Värde {
    int värde;
    Värde(int värde) {
        this.värde = värde;
    }
    public int getVärde() {
        return värde;
    }
}
```

```
class DubbelVärde extends Värde {
    int värde2;
    public DubbelVärde(int värde2) {
        this.värde2 = värde2;
    }
}
```

Stockholms universitet

Gränssnitt (interfaces)

- Gränssnitt (interface) är en **helt** abstrakt klass som enbart kan:
  - Deklarera abstrakta publika metoder
  - Deklarera konstanta variabler
  - Deklarera default-metoder
  - Deklarera statiska metoder
- De kan också ärva av andra gränssnitt

Stockholms universitet

Interface

```
interface Färgbar extends Comparable<Färgbar> {
    int SVART=1, VIT=-1, GUL=2, GRÖN = -2;
    int getFärg();
    void setFärg(int färg);

    default int inverteraFärg() {
        return -getFärg();
    }

    static int svart() {
        return SVART;
    }

    default int compareTo(Färgbar other) {
        return other.getFärg() - getFärg();
    }
}
```

Stockholms universitet

Interface

- Alla metoder som inte är default är abstrakta och måste implementeras av klasser som implementerar ett interface
- Default metoder får enbart anropa metoder som definieras i gränssnittet eller dess supertyp

Stockholms universitet

Interface: certifiering

- Interface deklarerar alltså ett antal metoder som en klass lovar att implementera
- Arv ger en ”är-en”-relation **och** delning av kod
- Interface ger en ”’är-en” relation utan delning av kod
- Likt ”roller” eller certifiering
  - Jag är certifierad kirurg; jag kan operera
  - Jag är certifierad `Färgbar`; jag kan `setFärg` och `getFärg`

Stockholms universitet

Interfaces

- En klass kan deklarera att den implementerar ett interface

```
class Bil implements Färgbar{
    private String regnr, märke;
    private int färg;
    public String getRegnr() { return regnr; }
    public String getMärke() { return märke; }
    public int getFärg(){ return färg; }
    public void setFärg(int f) { färg = f; }
}
```

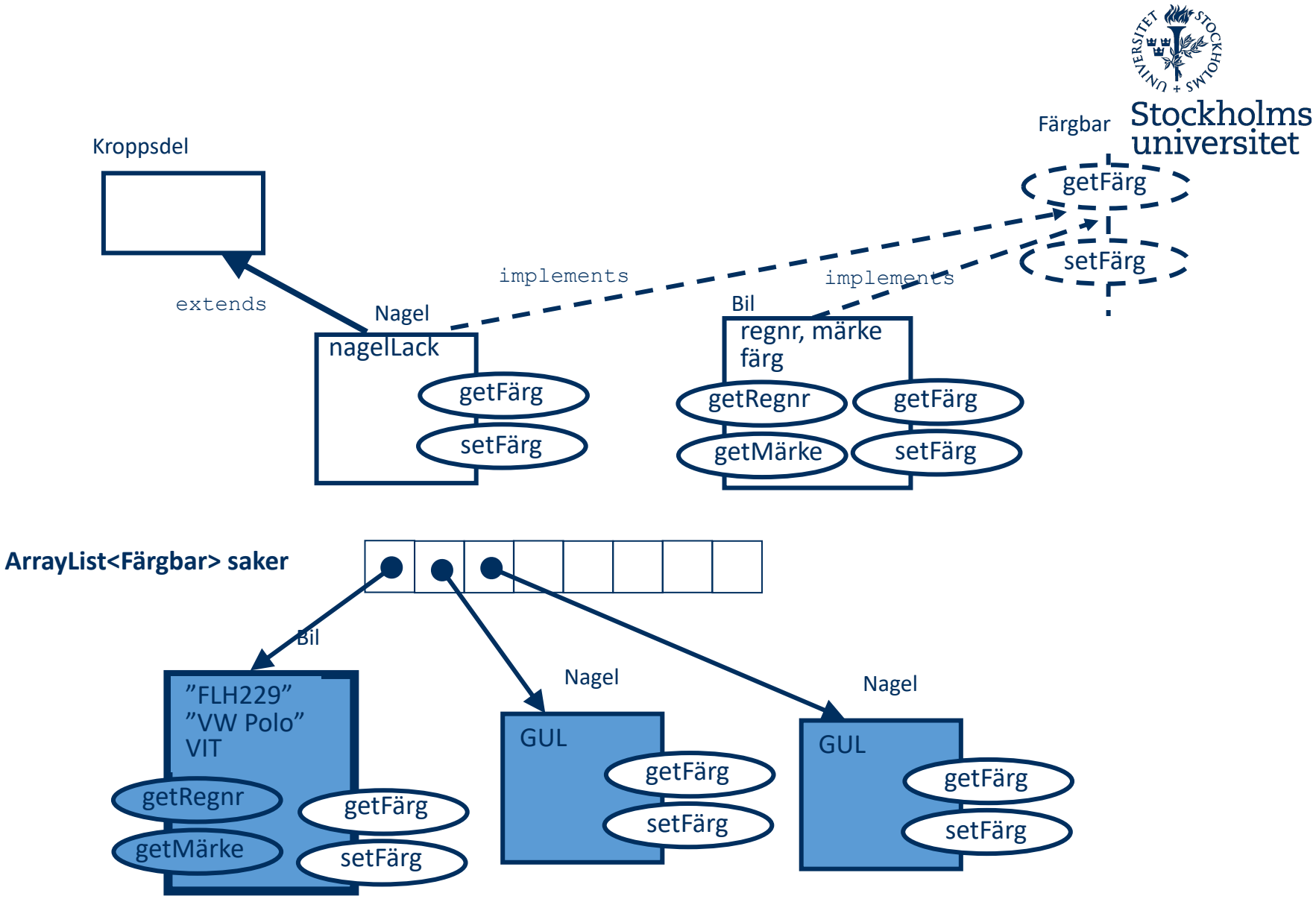
```
class Kroppsdel{ .....}

class Nagel extends Kroppsdel implements Färbar{
    private int nagelLack = GUL;
    public void setFärg(int lack) { nagelLack = lack; }
}
```

Stockholms universitet

Interface som referenstyp

- Interfacet kan sedan används som referenstyp

```
Bil b = new Bil("FLH229", "VW Polo", Färgbar.GUL);
Nagel[] naglar = new Nagel[10];
for(int i=0; i<10; i++) {
    naglar[i] = new Nagel();
}
ArrayList<Färgbar> saker = new ArrayList<>();
saker.add(b);
for(int i=0; i<10; i++) {
    saker.add(nagel[i]);
}
for(Färgbar saken : saker) {
    saken.setFärg(Färgbar.SVART);
}
for (Färgbar saken : saker) {
    saken.setFärg(saken.inverteraFärg());
}
```


Kroppsdel
extends
Nagel
nagelLack
getFärg
setFärg
implements
Färgbar
getFärg
setFärg
implements
Bil
regnr, märke
färg
getRegnr
getFärg
getMärke
setFärg
ArrayList<Färgbar> saker
Bil
”FLH229”
”VW Polo”
VIT
getRegnr
getFärg
getMärke
setFärg
Nagel
GUL
getFärg
setFärg
Nagel
GUL
getFärg
setFärg

Stockholms universitet

Nästa föreläsning

- Hur kan man byta beteende dynamiskt?
- Är det verkligen nödvändigt att alltid ha en ”är-en” relation eller finns det andra typer av relationer mellan objekt?