

Typhierarkier del 1

Gränssnitt, ärvning mellan gränssnitt, ärvning mellan klasser

Hur används *hierarkier* för att modellera *nära relaterade typer*?

Nu:

**De viktigaste
begreppen**

**Pragmatisk
användning av
hierarkier, ärvning**

Nästa:

Paus från ärvning

**Programmering av
grafiska gränssnitt
i Java**

Lite senare:

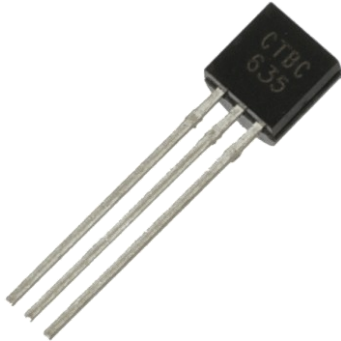
**Hierarkier och
ärvning del 2**

**Vad händer på ett
djupare plan?**

Fler begrepp...

Motivering 1: Verkligheten

- Vi har pratat om klasser av objekt



Tydliga
olikheter
→
separata
klasser

- Men om objekten är "**ganska lika**"?

- **En** klass: Generella **motorfordon**?

Bra: Vi kan **generalisera**...
"Detta gäller alla motorfordon"

- **Tre** klasser: Bilar, bussar och lastbilar?

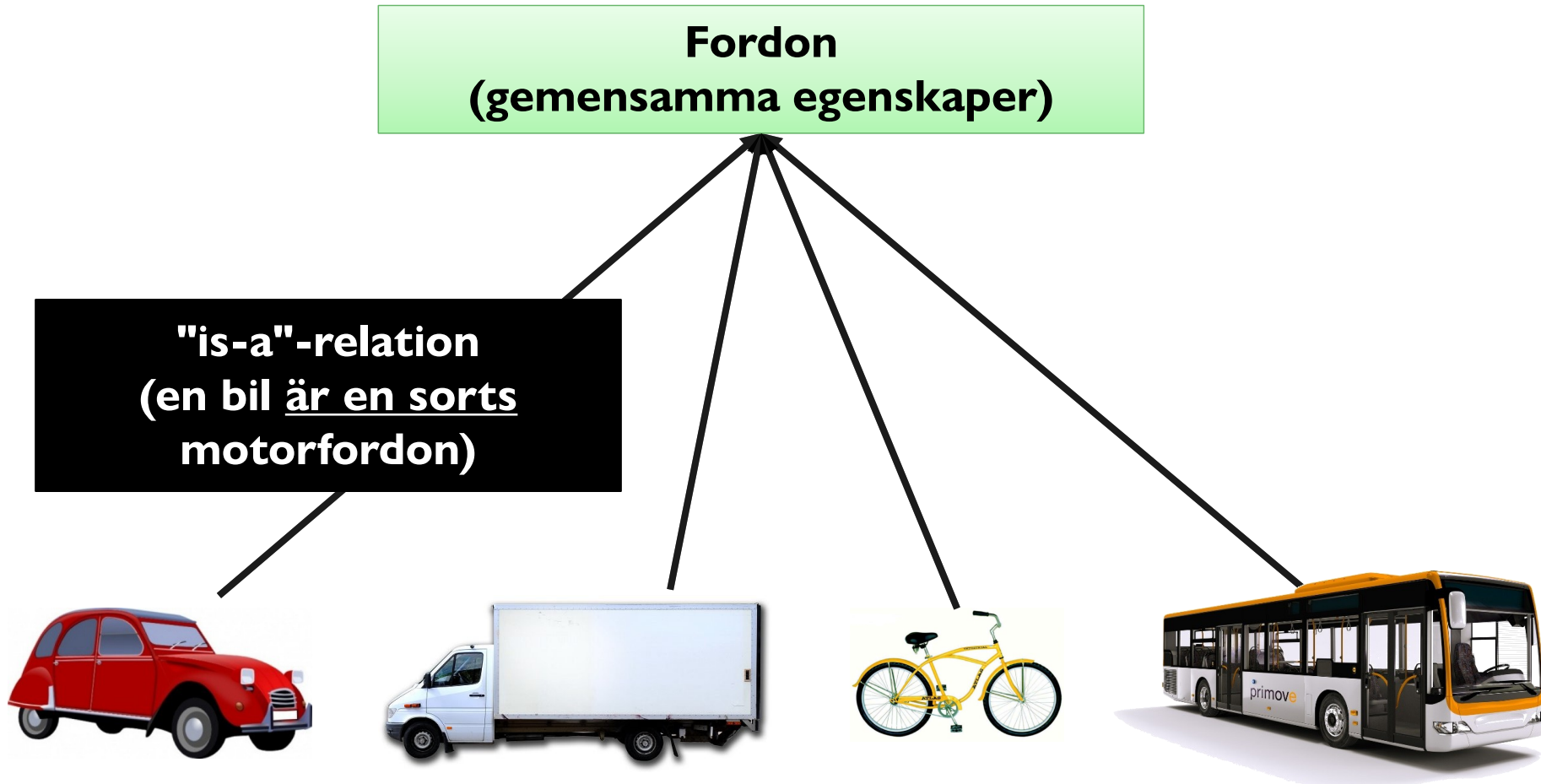
Bra: Vi kan vara **specifika**...
"Detta gäller bara bussar"

- **Fem** klasser: Bil, lätt lastbil, tung lastbil, minibuss, buss?



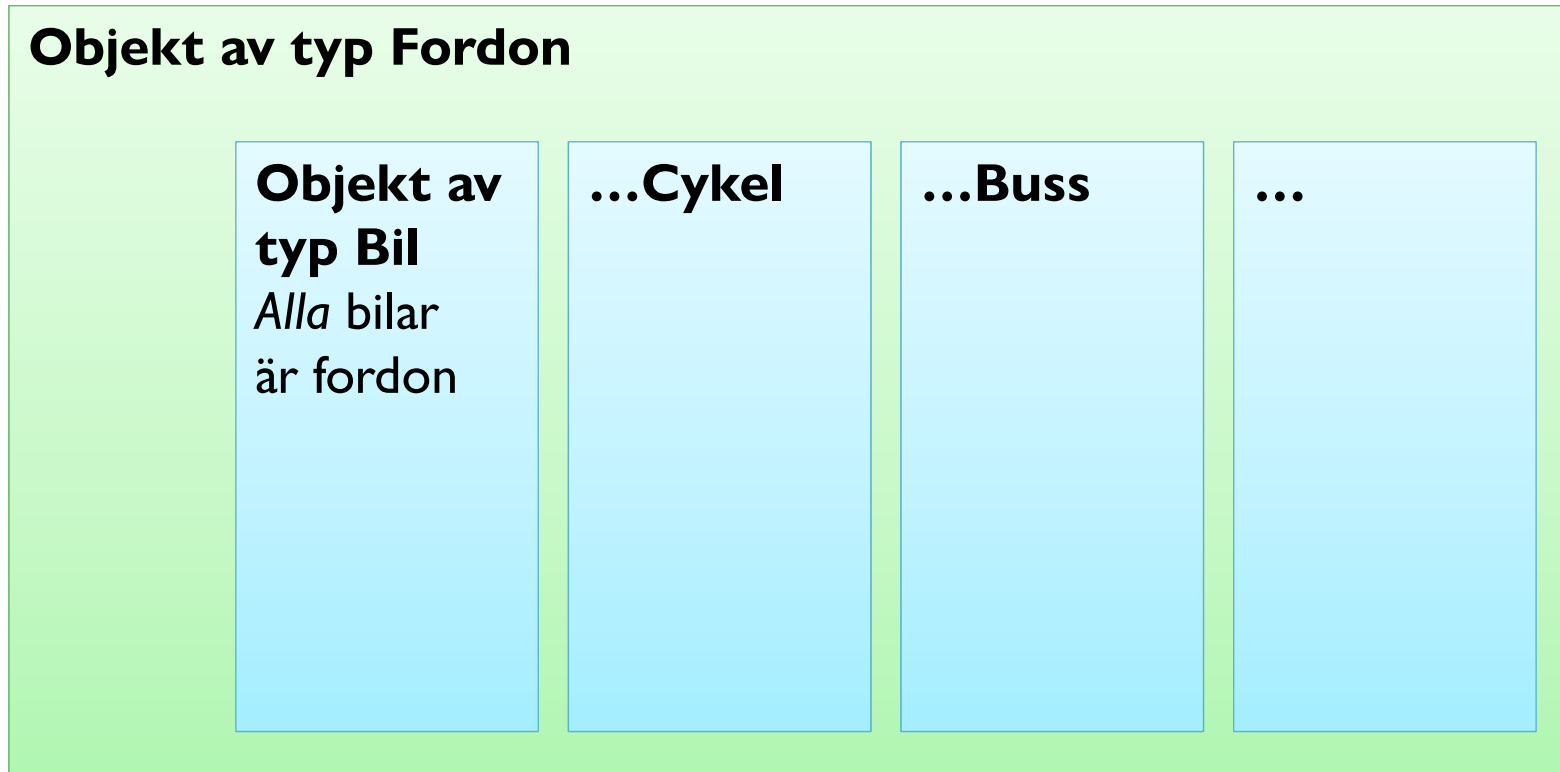
Hierarkier av klasser

- Vi kan ha **hierarkier** med **både** generella och specifika klasser!



Hierarkier och delmängder

- En bil är ett fordon...



Med denna hierarki:
En Bil kan aldrig vara en Buss, är alltid ett Fordon

Motivering 2: Datastrukturer

Liknande datatyper (1)

- Flera sätt att lagra **listor**!

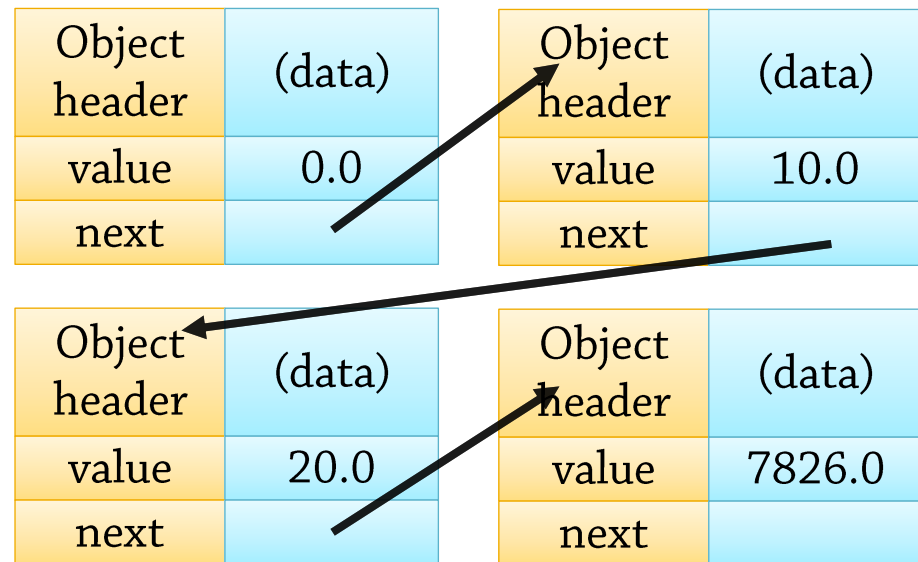
"Linjärt" i minnet...

Object header	(data)
pos 0	0.0
pos 1	10.0
pos 2	20.0
...	...
237000	7826.0

Snabbt att slå upp element 92000:
Adress = liststart + 92000 * elementstorlek

Långsamt att stoppa in ett element:
Måste flytta alla efterföljande element

Länkad lista...



Långsamt att slå upp element 92000:
Gå till start, följ pekare 92000 gånger

Snabbt att stoppa in ett element:
Ändra två pekare

Liknande datatyper (2)

- **Många** olika implementationer: `LinkedList`, `ArrayList`, `SkipList`, ...
 - Mycket **gemensamt** – t.ex. vilka **metoder** som **finns**
 - `int size();`
 - `Object get(int index);`
 - `void set(int index, Object element);`
 - Stora skillnader i **kod** (metoder, fält) → **olika klasser**

LinkedList

```
size() : int  
get(int pos) : Object  
add(Object element) : Object  
set(int pos, Object element) : void  
insert(...) : void
```

ArrayList

```
size() : int  
get(int pos) : Object  
add(Object element) : Object  
set(int pos, Object element) : void
```

SkipList

...

```
size() : int  
get(int pos) : Object  
add(Object element) : Object  
set(int pos, Object element) : void
```

Liknande datatyper (3)



- Om vi måste ange **exakt typ** för parameter, returvärde:

```
public void quicksort(ArrayList someList) {  
    // Kräver ArrayList!  
  
    // Men skulle fungera med många  
    // klasser som har size(), get(), set(),  
}
```

```
public ArrayList getNames() {  
    // Lovar att returnera ArrayList!  
    // Anropare kan förlita sig på detta  
    // ➔ måste fortsätta med samma typ...  
}
```

Duck Typing

Python

```
class ArrayList:  
    def extend(self, list):  
        ...  
  
class LinkedList:  
    def extend(self, list):  
        ...  
  
def add_lists(list1, list2):  
    list1.extend(list2)  
    return list1
```

Även om vi har en ArrayList-klass:

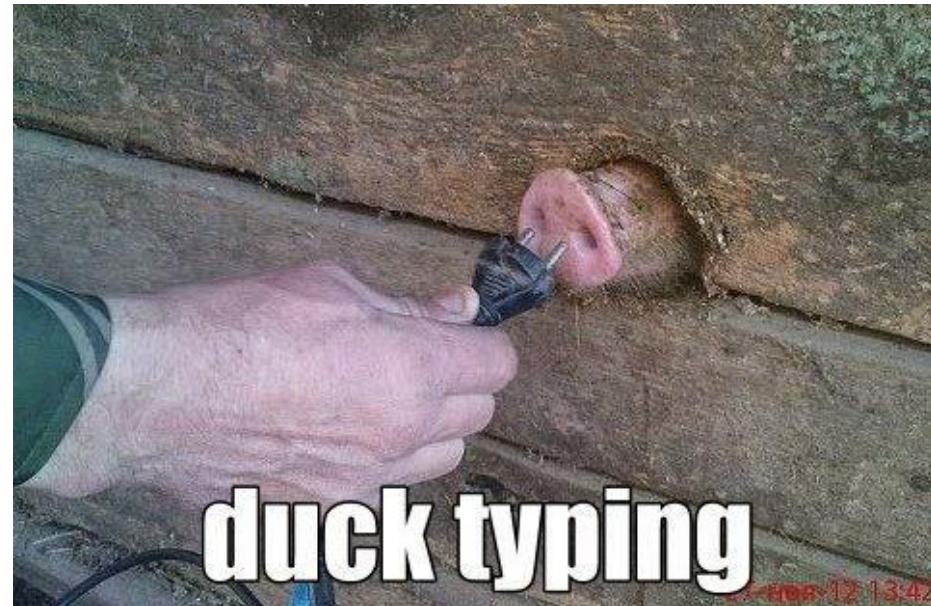
Parametertyp anges inte – kontrollera vid körning om objektet har extend()!

Duck Typing:

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck

– James Whitcomb Riley

När jag ser ett objekt med get(), set(), extend(), då kallar jag det objektet en lista



Java

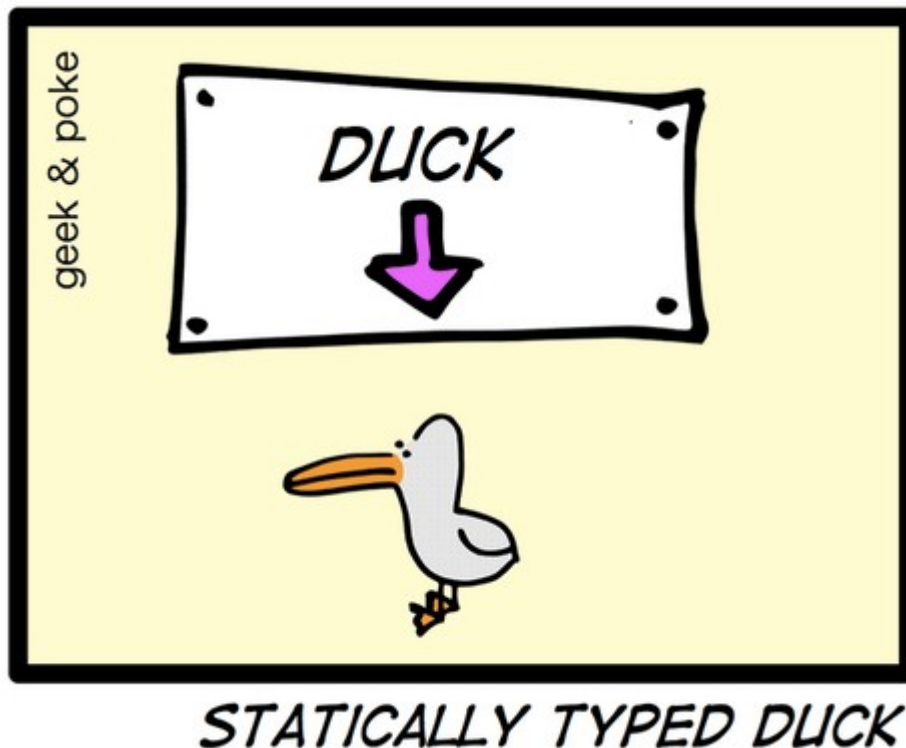
```
public class ArrayList
{ ... definierar addAll() ... }

public class LinkedList
{ ... definierar addAll() ... }

public static ArrayList
addLists(ArrayList list1, ArrayList list2)
{
    list1.addAll(list2);
    return ret;
}
```

Måste ange parametertyp

Behöver annat sätt att ange "ArrayList eller LinkedList eller ..."



- Lösning: Vill ha hierarkier även här

Objekt av typ **List**

ArrayList

LinkedList

SkipList

...

Enkla typhierarkier i Java: Gränssnitt (interface)

- Ett interface är specifikation utan implementation

```
/** An ordered collection (also known as  
    a sequence). Elements in the list are  
    indexed beginning at 0 and ... */
```

```
public interface List {
```

```
    /** Returns the number of elements  
        in this list. */
```

```
    int size();
```

```
    Object get(int index);
```

```
    void add(Object element);
```

```
    void set(int index, Object element);
```

```
}
```

Metodsignaturer och dokumentation

anger *krav* för alla
som påstår sig vara av typ List
(Koppling till abstrakta datatyper!)

Metodsignaturer:

Returtyp, namn, parametrar
Inga fält, ingen metodkod...

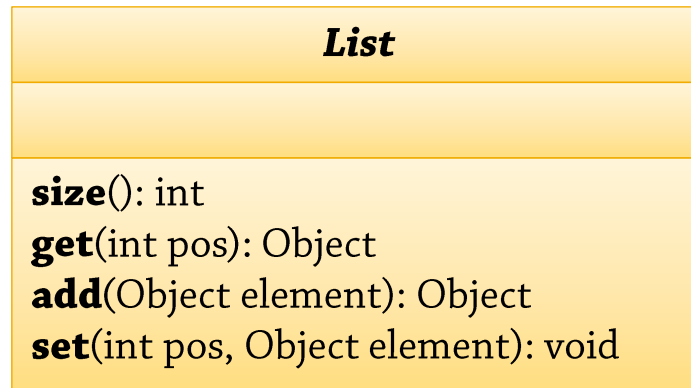
Metoder i ett gränssnitt

är alltid (underförstått):
public – tillgängliga för alla
abstract – ej implementerade här

- Gränssnitt kan inte instansieras
 - **new** List()?
Det finns ingen kod för dess metoder!

Interface: Hierarkier 1

- Möjliggör en hierarki av datatyper!



List:
"Detta ska alla
listor klara"

Säger inte hur!

LinkedList

ArrayList

SkipList

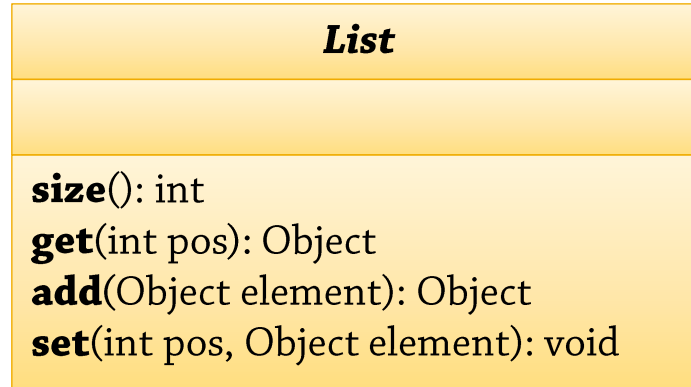
...

...

LinkedList: Vanlig klass –
som uppfyller List-specifikationen

Interface: Hierarkier 2

■ Terminologi:



List är
supertyp
till **LinkedList**



LinkedList

ArrayList

SkipList

LinkedList är
subtyp
till List

supertyp = supertype
subtyp = subtype

Interface: Vad vinner vi?

```
void quicksort(ArrayList someList) {  
    // Kräver ArrayList!  
  
    // Men skulle fungera med många  
    // klasser som har size(), get(), set()  
}
```



```
void quicksort(List someList) {  
    // Kan ta emot ArrayList, SkipList,  
    // ...!  
}
```

Kräv mindre!

```
ArrayList getNames() {  
    // Lovar att returnera ArrayList!  
    // Anropare kan förlita sig på detta  
    // ➔ måste fortsätta med samma typ...  
}
```



```
List getNames() {  
    // Lovar bara List:  
    // Vi kan fritt byta vilken typ av lista  
    // som returneras!  
}
```

Lova mindre!

Interface 2: Implementation

- En Java-klass kan implementera ett gränssnitt

```
public class ArrayList {  
    private Object[] elements;  
    private int length;  
  
    public int size() { return length; }  
    public Object get(int index) { return elements[index]; }  
    public void add(Object element) { ... }  
    public void set(int index, Object element) { ... }  
    ...  
  
    public void shuffle() { ... }  
}
```

Ett löfte att uppfylla
vad gränssnittet lovar ("kontrakt")

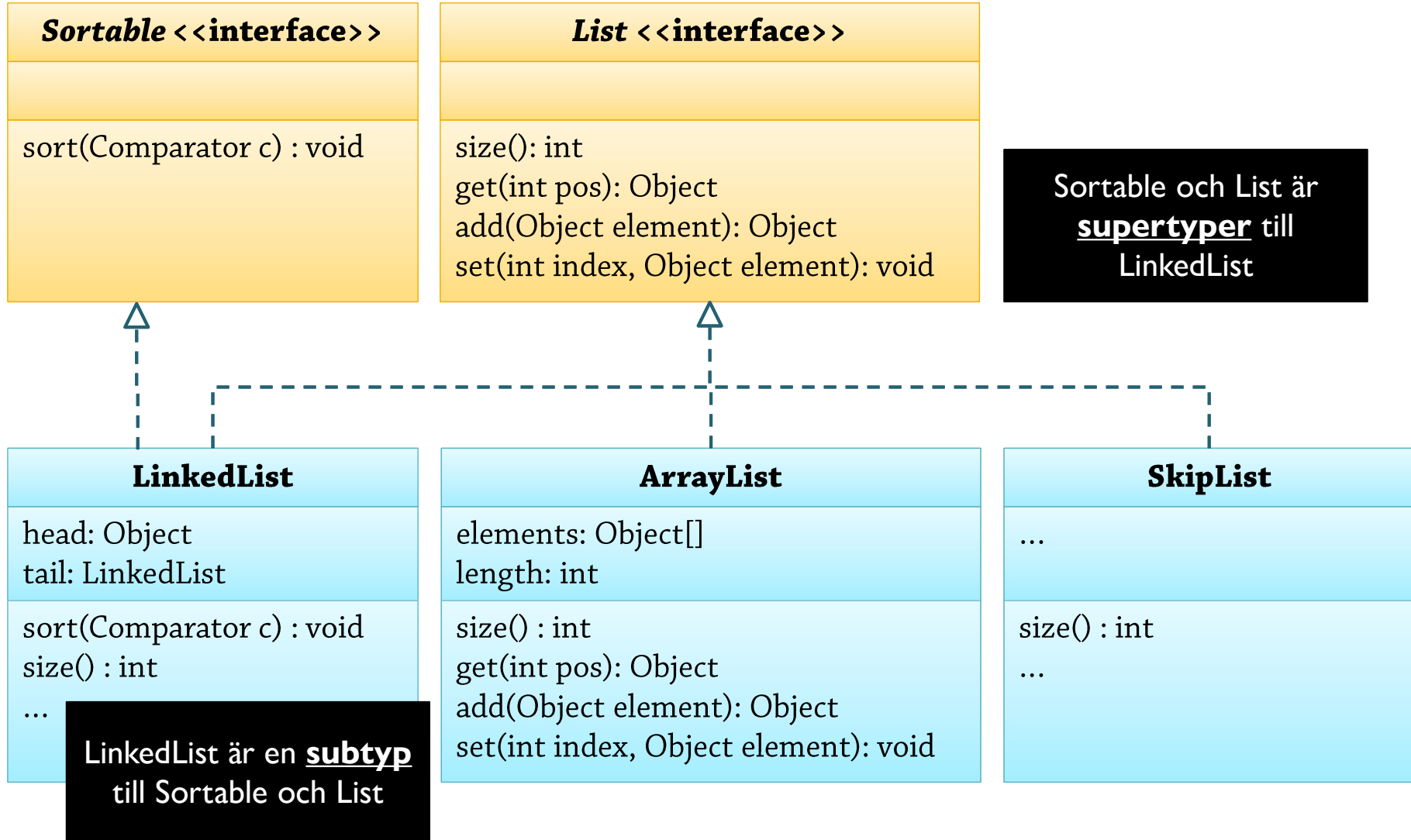
Kompilatorn verifierar att
metoderna från List finns
Programmeraren verifierar att
kontraktet i övrigt är uppfyllt

Kan ha fler metoder än gränssnittet
(håller ändå vad den lovar)

implementera = implement

Interface 3: Typhierarki

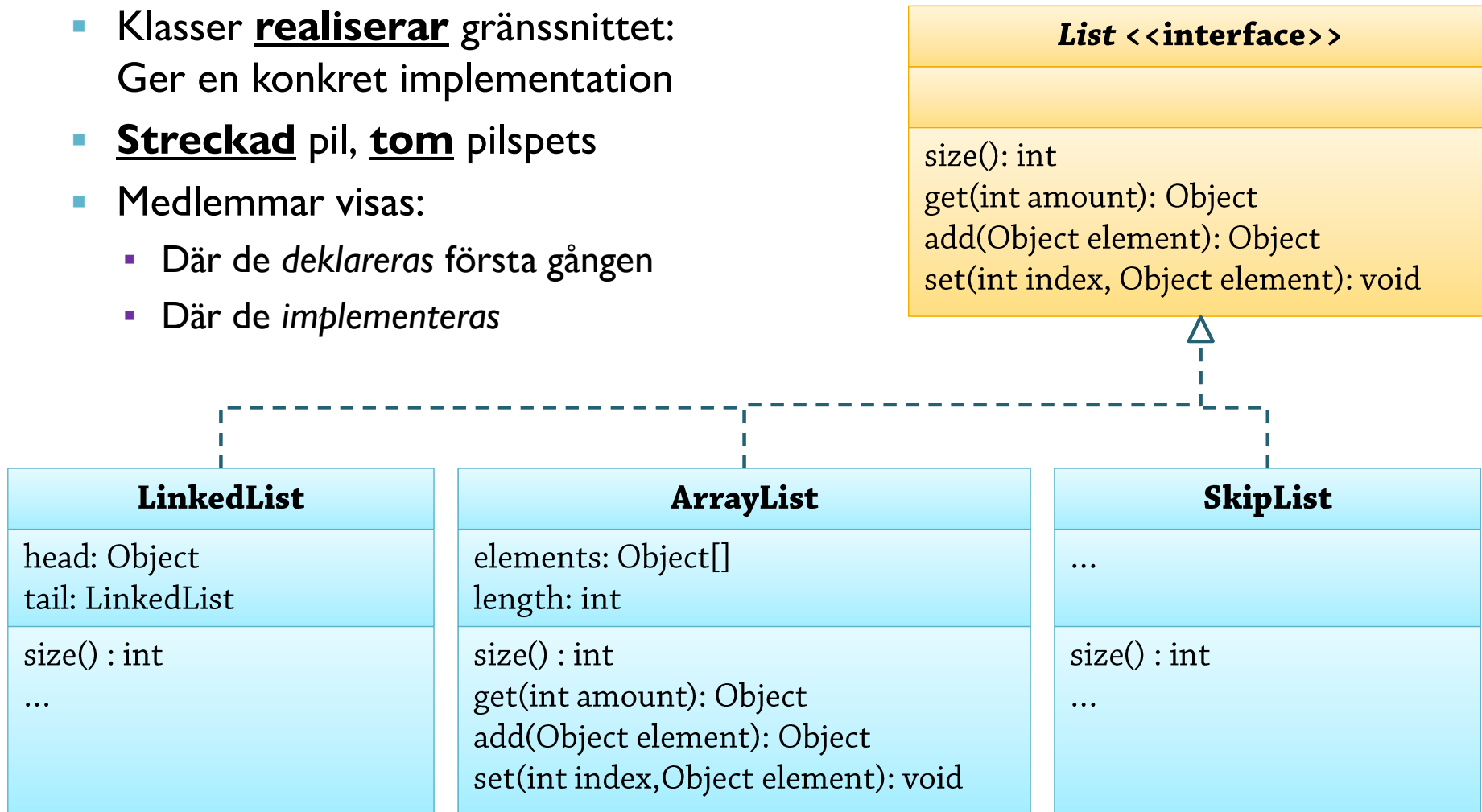
- En klass kan implementera **flera** gränssnitt



UML: Klassdiagram med gränssnitt

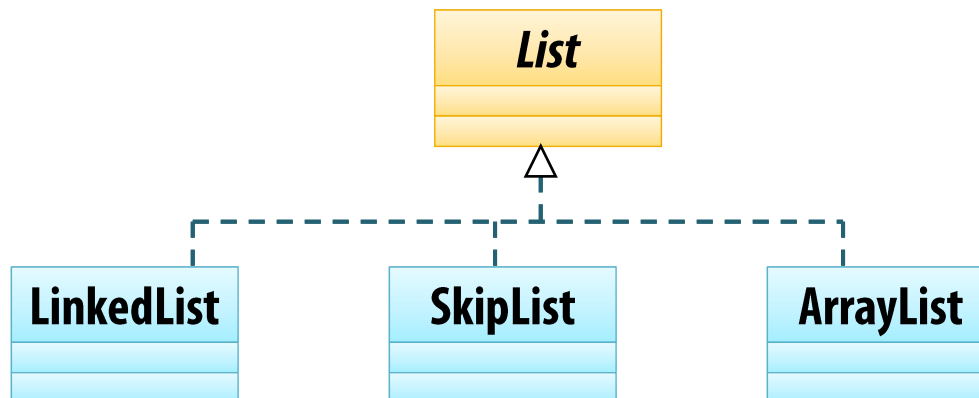
■ UML-diagram: Klassrelationen realisering

- Ett gränssnitt är bara en *beskrivning*
- Klasser **realiserar** gränssnittet:
Ger en konkret implementation
- **Streckad** pil, **tom** pilspets
- Medlemmar visas:
 - Där de *deklarerar* första gången
 - Där de *implementeras*



Om man har flera "klassvarianter"
med gemensam bas:

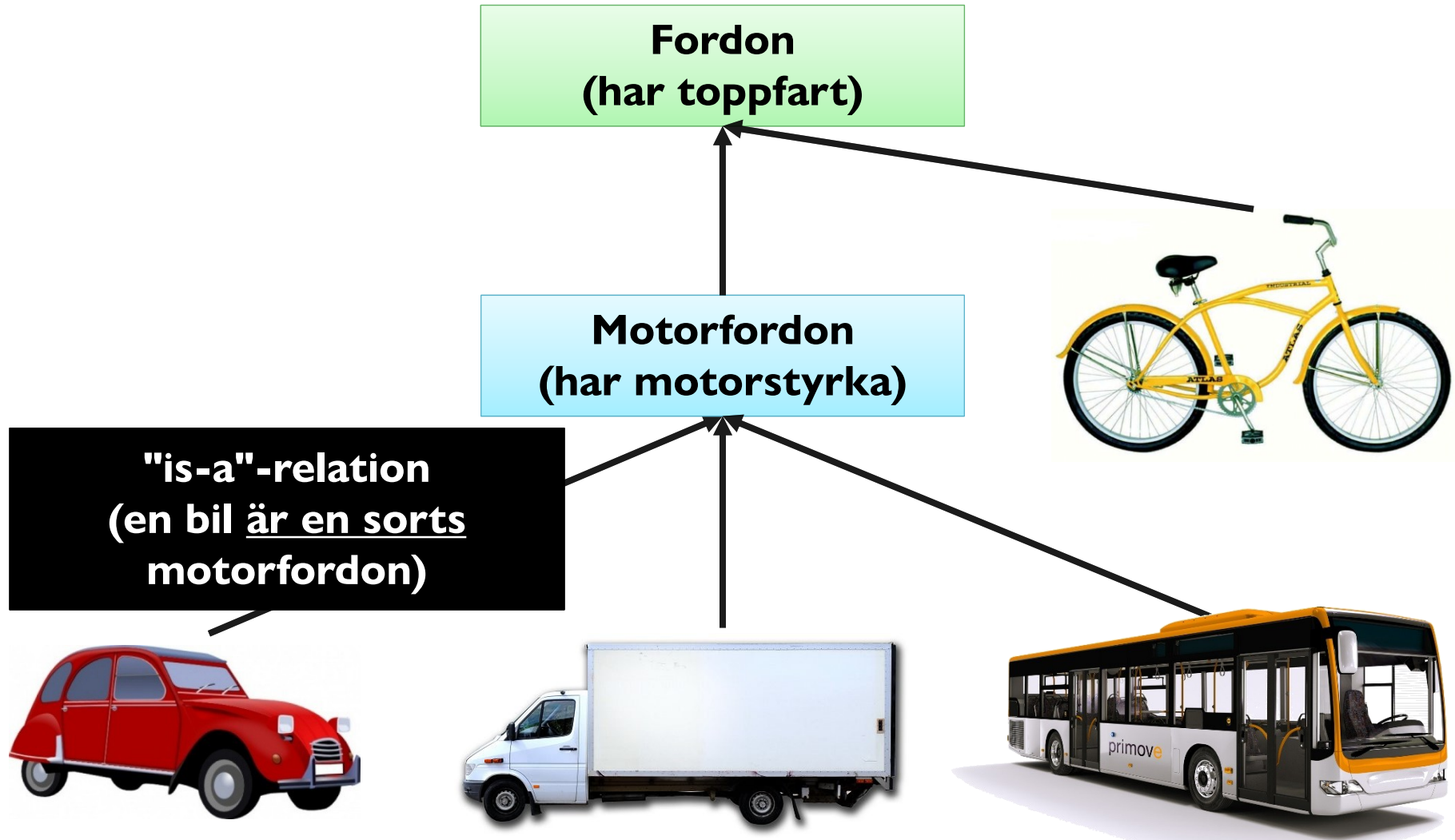
Oftast bör man ha ett gränssnitt,
så den gemensamma basen kan användas



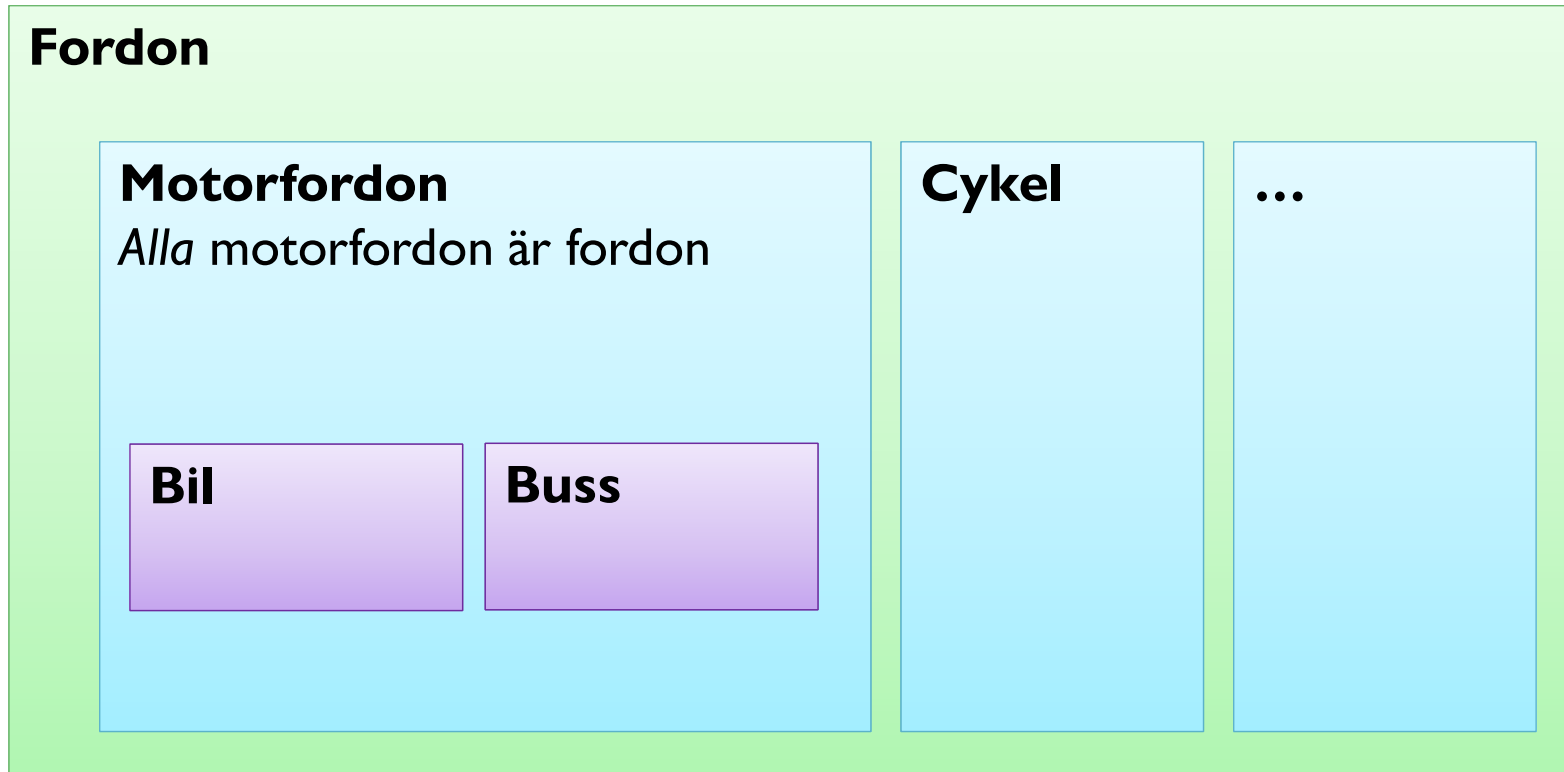
Ärvning mellan gränssnitt

Hierarkier av klasser

- Hierarkier kan ha flera nivåer



- Fortfarande "alla x är y":



Interface-ärvning 1: Hierarkier

- Vi såg typhierarkier i två nivåer

Gränssnitt anger
funktionalitet:
Vad som är
gemensamt för
alla listor

List <<interface>>

size(): int
get(int pos): Object
add(Object element): Object
set(int index, Object element): void



LinkedList

...

...

ArrayList

...

...

SkipList

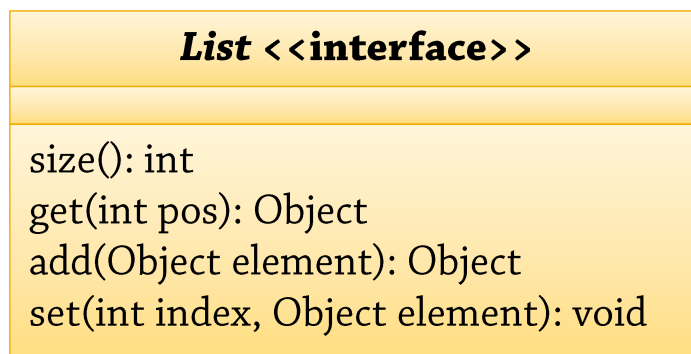
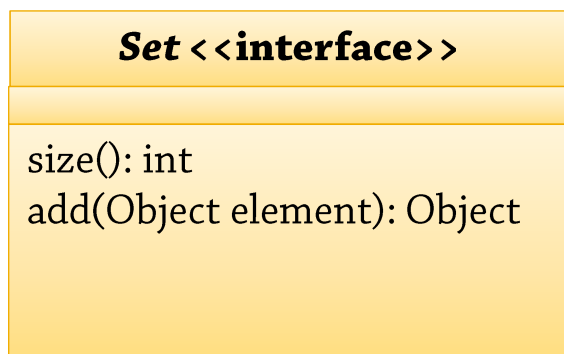
...

...

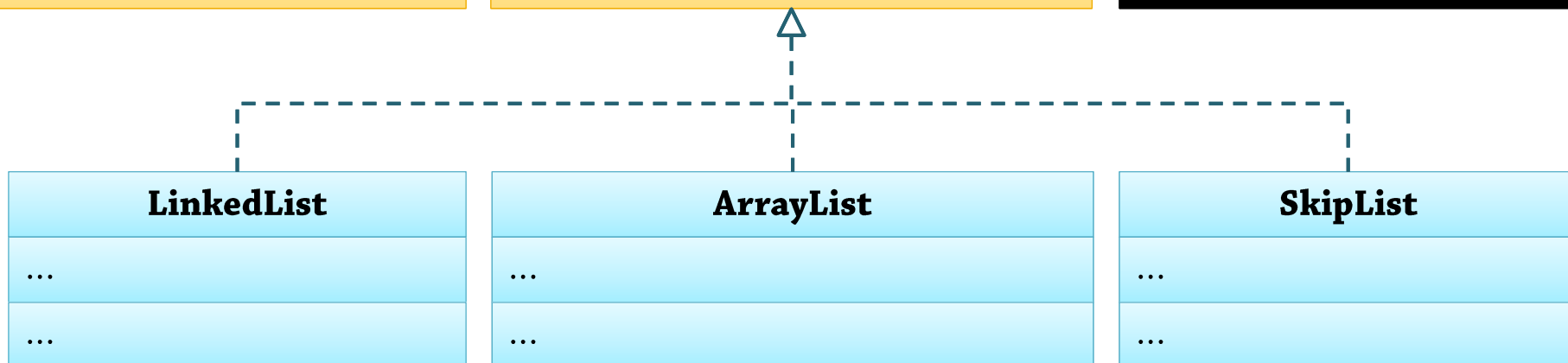
Klasser ger oss
implementationer

Interface-ärvning 2: Gemensamt

- Men **List** har också något gemensamt med **Set**...

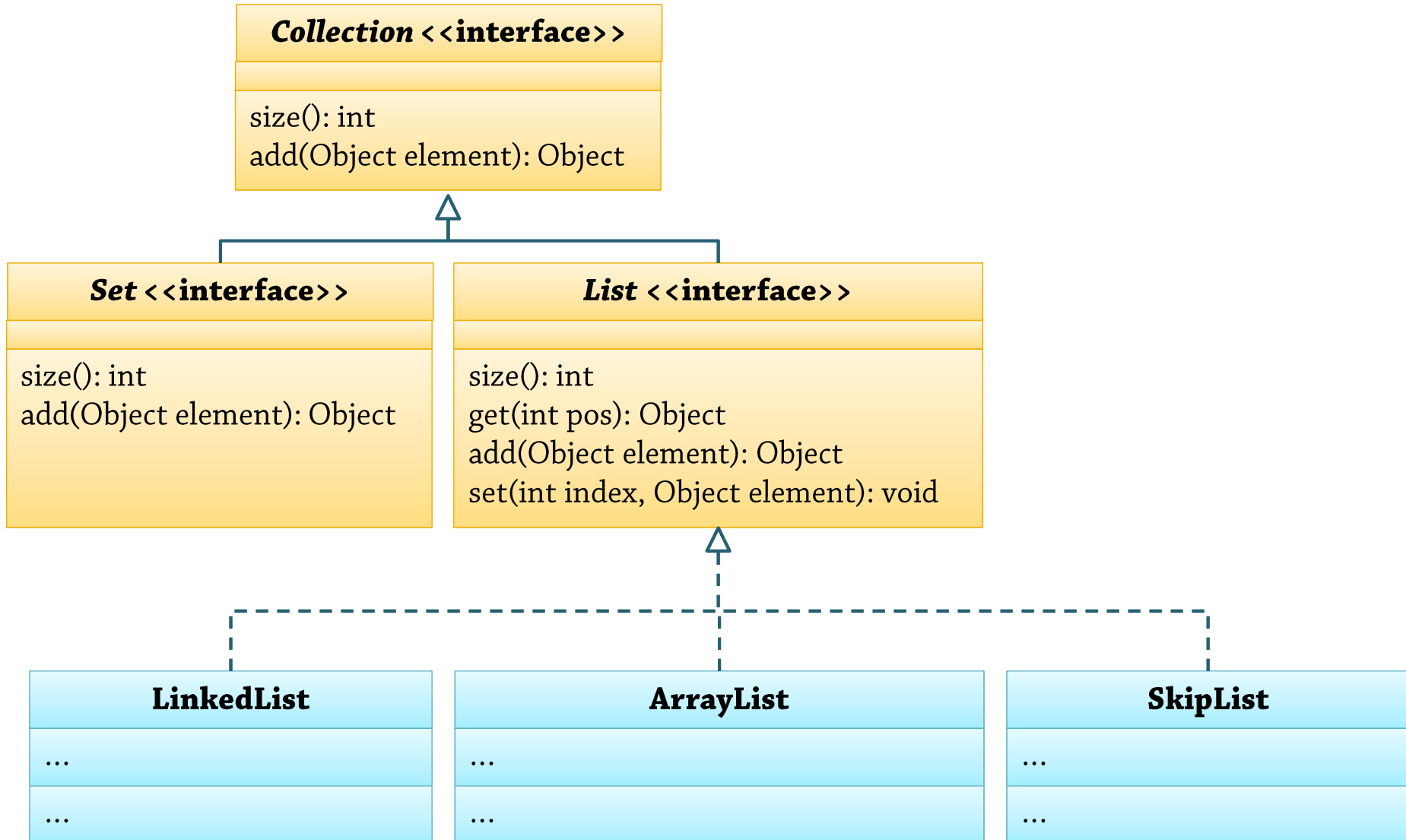


Hur deklarerar vi
en variabel som kan
peka på en lista *eller*
en mängd?



Interface-ärvning 3: En nivå till

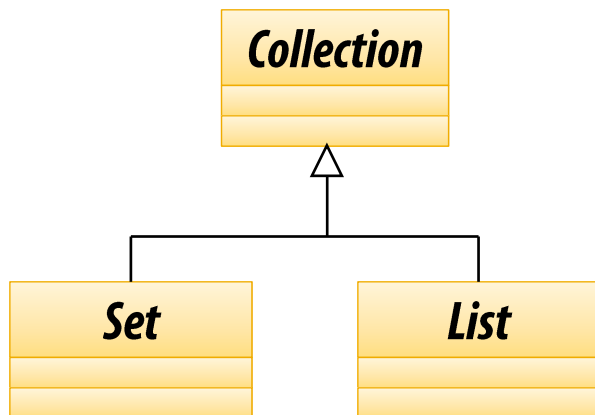
- Genom ytterligare en nivå!



Interface-ärvning 4

■ Detta är interface-ärvning

- **public interface** Collection {
 void add(Object obj);
 boolean contains(Object obj);
}
- **public interface** List {
 void set(int index, Object obj);
 ...
}



Collection är ett superinterface till List

List utökar eller ärver från Collection
List är ett underinterface till Collection

List extends or inherits from Collection
List is a subinterface of Collection

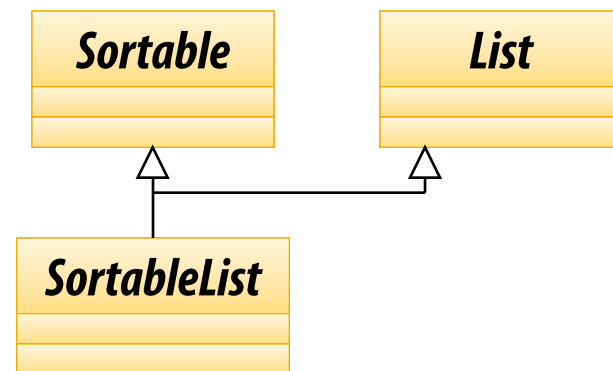
Ärver kontrakt/löften:
Collection lovar att ha
en add()-metod, ...

Adderar egna löften:
List lovar också ...

Interface-ärvning 5: Multipel ärvning

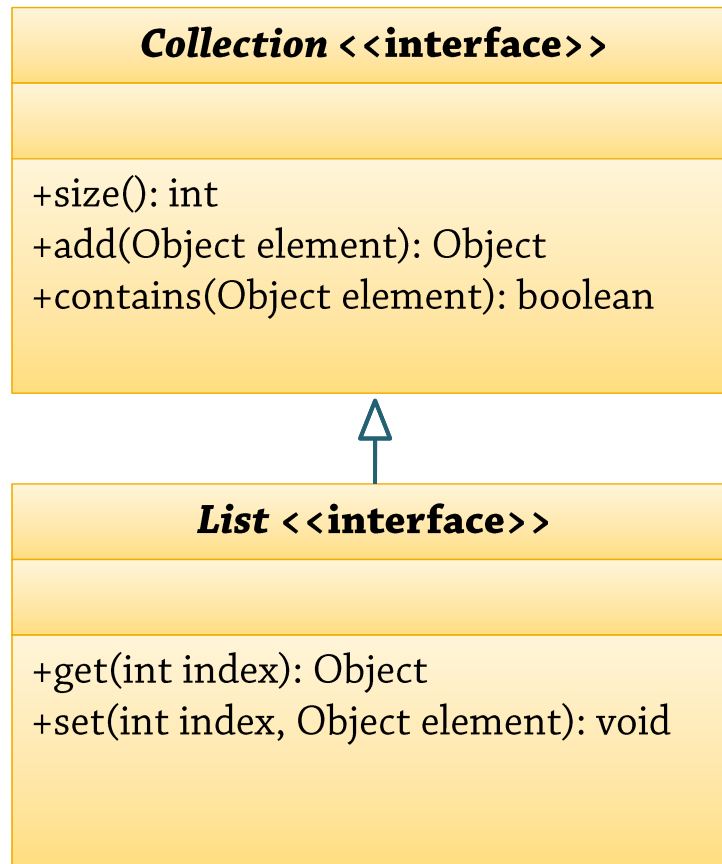
- **Multipel** ärvning tillåts för gränssnitt

- **public interface** *SortableList* {
 // Kombinerar supertypernas löften / kontrakt
 // Inget nytt adderat
}
- **public interface** *NetworkSocket* {
 // Kombination av två supertyper
 // Plus: ytterligare en metod krävs
 public *NetworkAddress* getRemoteHost();
}



multipel ärvning = multiple inheritance

- Klassrelation: **Generalisering** (ärvning)
 - List **specialiserar** Collection (lägger till nya krav, nya metoder)
 - Collection **generaliserar** List
 - Vanlig linje, tom pilspets



Ärvning av implementation

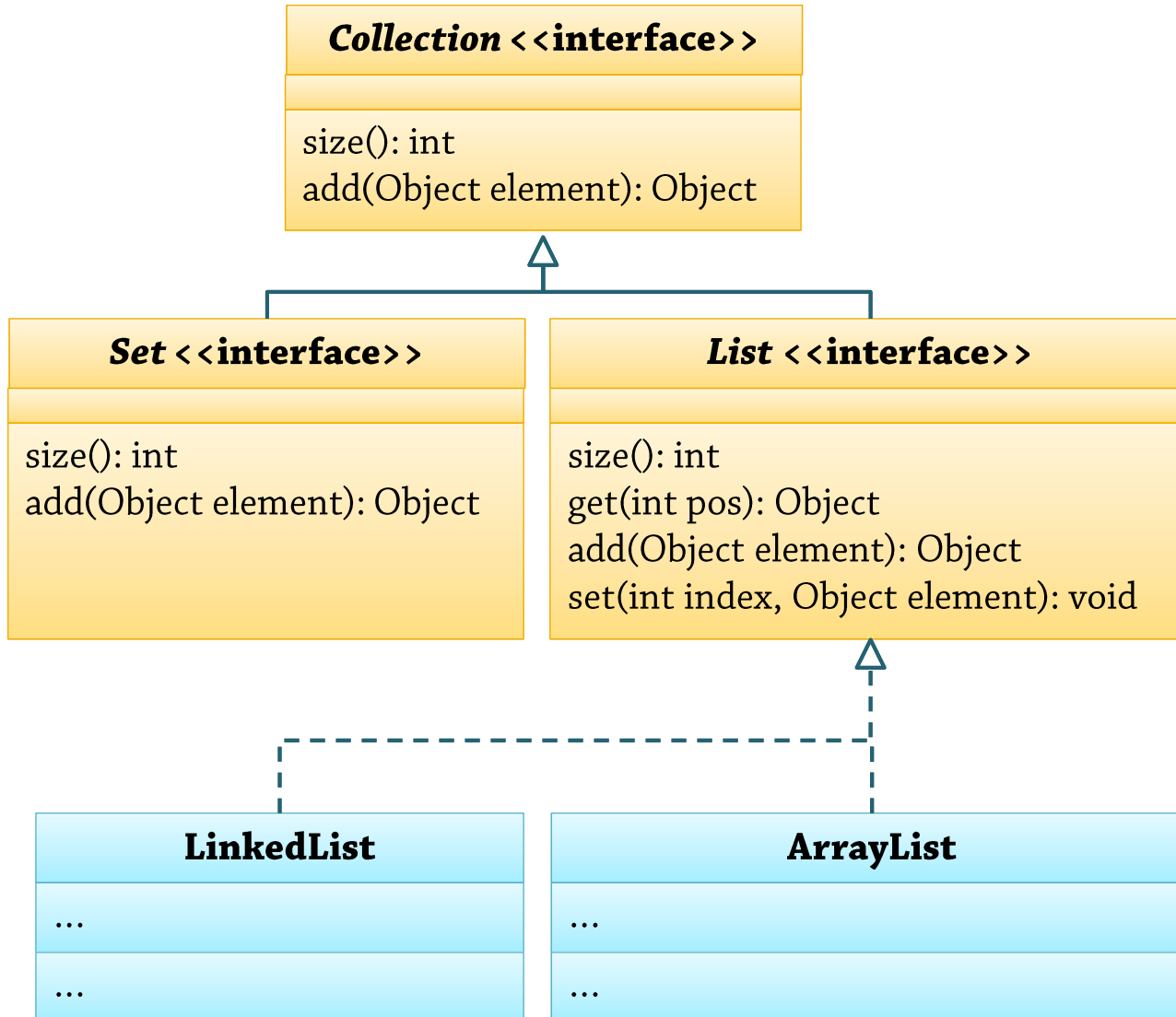
Ärvning av *implementation*, inklusive *abstrakta klasser*

Hur ska vi ärva?

När ska vi ärva?

Implementationsarv 1: Klassnivåer

- Nu kan vi ha godtyckligt antal nivåer...



...men bara en
klassnivå.
Vill vi ha mer? Varför?

- En anledning: **Addera** egenskaper + funktionalitet till en klass

**Vill ibland lagra
”geometriska” cirklar**

**x
y
radie**

Vissa ska vara ”grafiska”

**Outlinefärg
Fyllfärg
Ritmetod**

Ha kvar ”rena” cirklar:

- **Geometriklasser ska bara ha geometri, renare design**
 - **Lägre minnesåtgång**
- **Kommer från klassbibliotek, kan inte ändras**

Utökad version!

Implementationsarv 3: Nya medlemmar

```
public class GraphicCircle extends Circle {  
    private Color outline, fill;
```

Ärver x, y, r

Adderar outline, fill (Color-objekt)

```
    public GraphicCircle(double x, double y, double r, Color outline, Color fill) {  
        // ...  
        this.outline = outline;  
        this.fill = fill;  
    }  
    public void draw(DrawWindow dw) {  
        dw.drawCircle(x, y, r, outline, fill);  
    }  
}
```

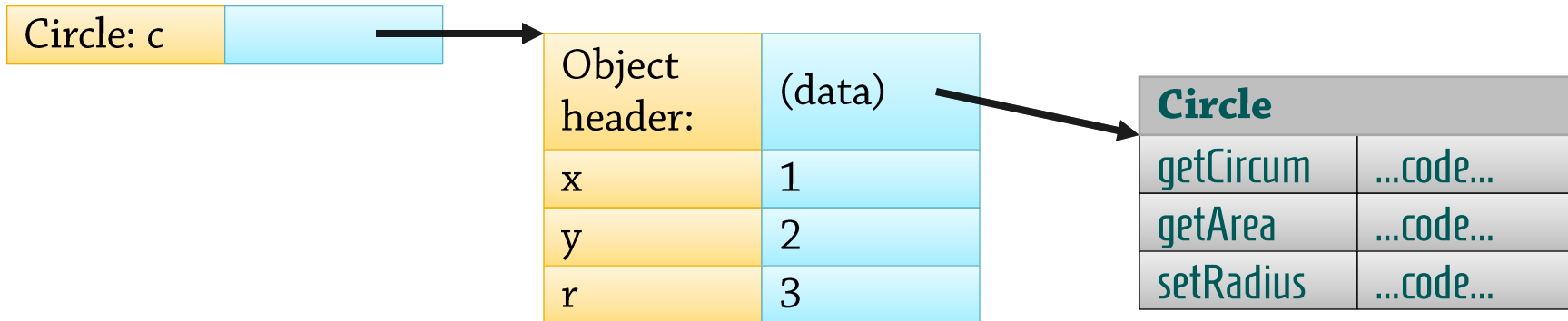
Ärver getRadius()

Adderar draw()

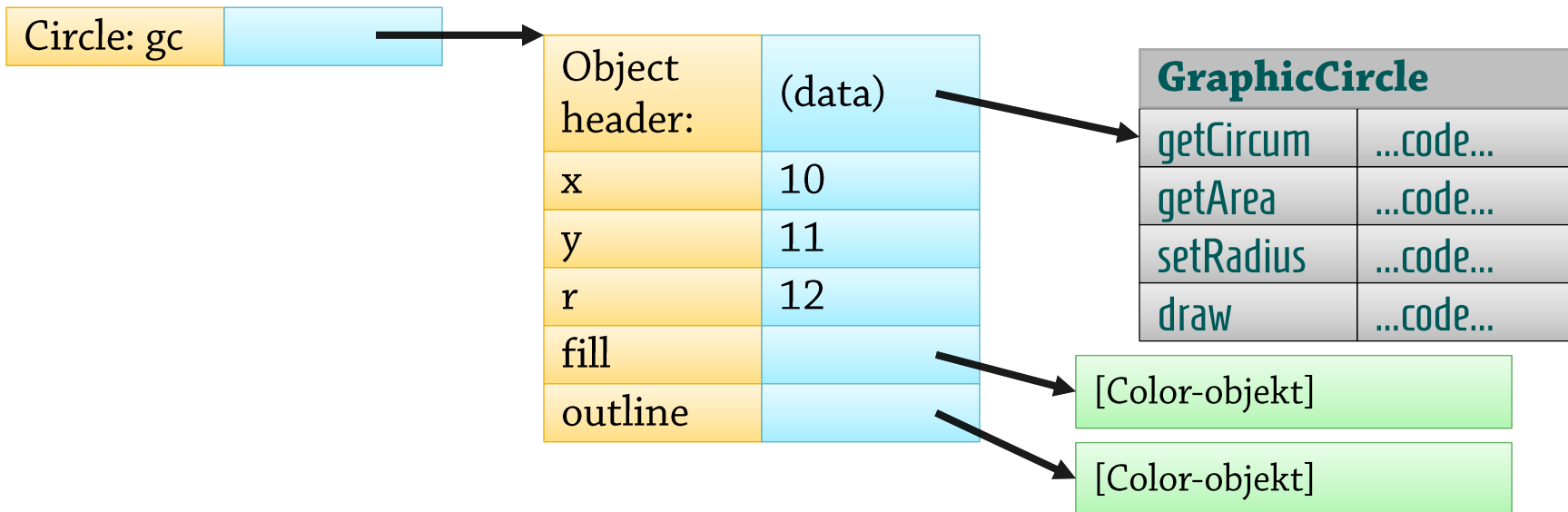
- GraphicCircle utökar eller ärver från Circle
 - GraphicCircle är en subklass till Circle
 - Circle är en superklass till GraphicCircle

Implementationsarv 4: Visualisering

▪ **Circle** c = **new** Circle(1, 2, 3);



▪ **Color** gray = **new** Color(200,200,200);
Color red = **new** Color(255,0,0);
Circle gc = **new** GraphicCircle(10, 11, 12, gray, red);



Implementationsav 5: Overriding

- Anledning 2: Ändra eller utöka nedärvd funktionalitet

```
public class Circle {  
    private double x, y, r;  
    public boolean isValid() {  
        if (r < 0) return false;  
        else return true;  
    }  
}
```

```
public class GraphicCircle extends Circle {  
    private Color outline, fill;
```

```
    public boolean isValid() {  
        if ( ) return false;  
        if (outline == null || fill == null) return false;  
        return true;  
    }  
}
```

Kan override:a
("ersätta") metoder:
Ge dem en ny
implementation

Fråga superklassen
vad den tycker...

Kräver identiska
parametertyper –
annars blir det
overloading

Implementationsav 6: super()

■ Modellering med **super()**

```
public class Circle {  
    private double x, y, r;  
    public boolean isValid() {  
        if (r < 0) return false;  
        else return true;  
    }  
}
```

```
public class GraphicCircle extends Circle {  
    private Color outline, fill;
```

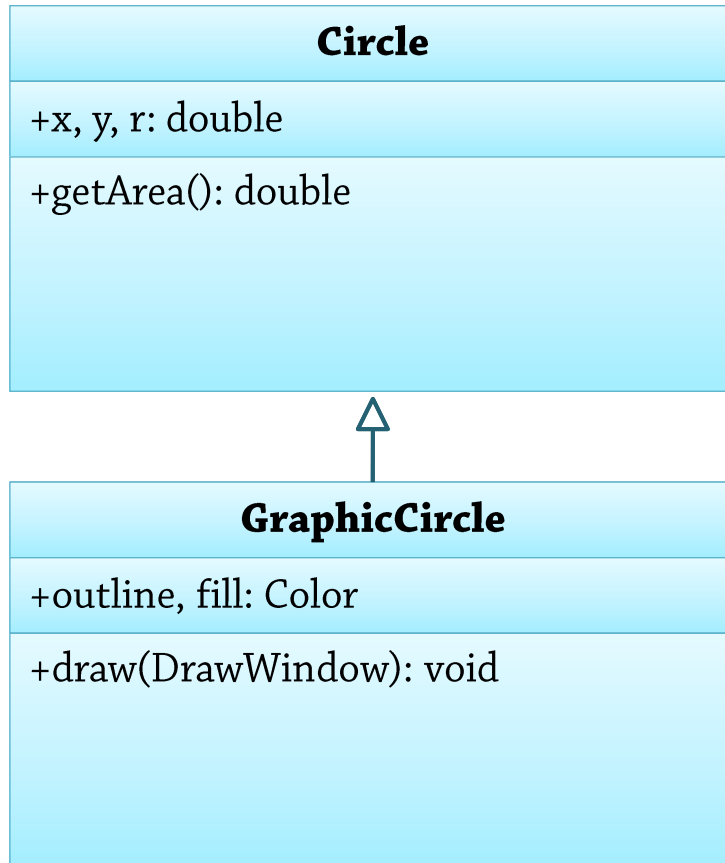
```
    public boolean isValid() {  
        if ( ) return false;  
        if (outline == null || fill == null) return false;  
        return true;  
    }  
}
```

3. ...skulle vi inte "få med" framtida ändringar i Circle:s isValid()!

2. Om vi istället skrev här:
if (r < 0) return false; ...

1. Modellering: Inte bara bekvämlighet
Superklassen vet när dess fält är giltiga – vi måste fråga den!
Sedan kan vi gå vidare med vårt ansvarsområde

- Klassrelation: **Generalisering** (ärvning)
 - Som för gränssnitt...



Annoteringar vid overriding

- Java: Kan ange "extra information om koden" via annoteringar
 - Exempel: **@Override** kan stoppas in automatiskt av IDEA

```
public class ArrayList {  
    public Object get(int index) { ... }  
}
```

```
public class DebugArrayList extends ArrayList {  
    @Override  
    public Object get(int index) {  
        System.out.println(  
            "Getting object at index " + index);  
        return super.get(index);  
    }  
}
```

Den här implementationen
ersätter en annan!

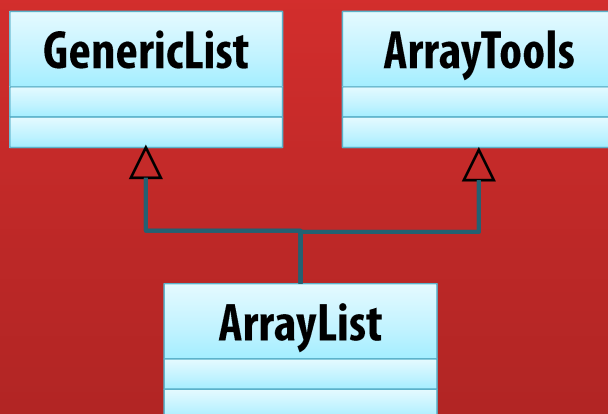
Stava fel, använd fel typer,
ta bort get() från superklassen



kompileringsfel!

Multipel ärvning?

- **Multipel ärvning** i vissa språk: Utöka **flera klasser**
 - Användbart och förvirrande



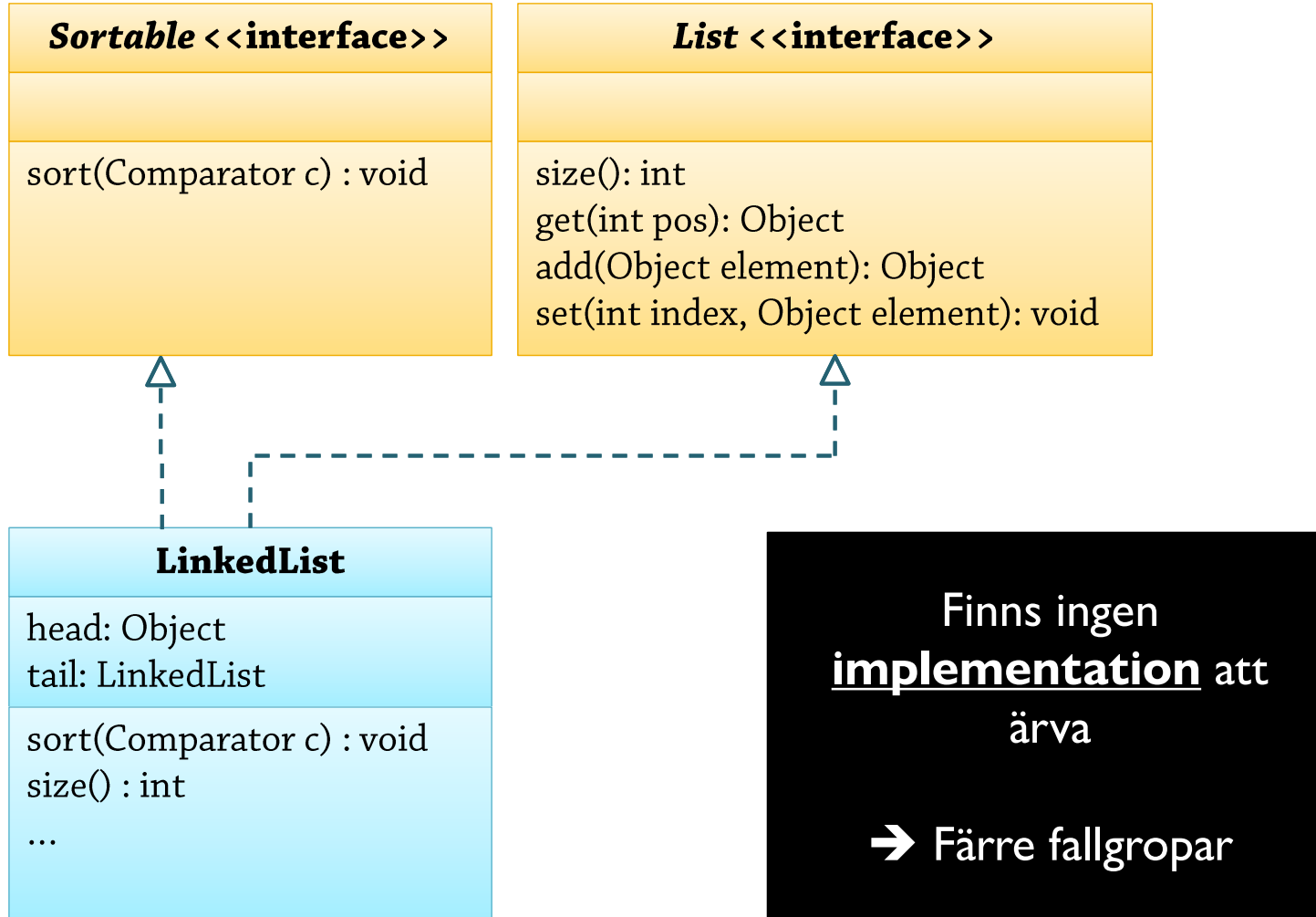
Om båda superklasser
har implementerat `isEmpty()`...

Vilken implementation
ska `ArrayList` ärvva?

Förbjudet i Java

Multipel implementation av gränssnitt

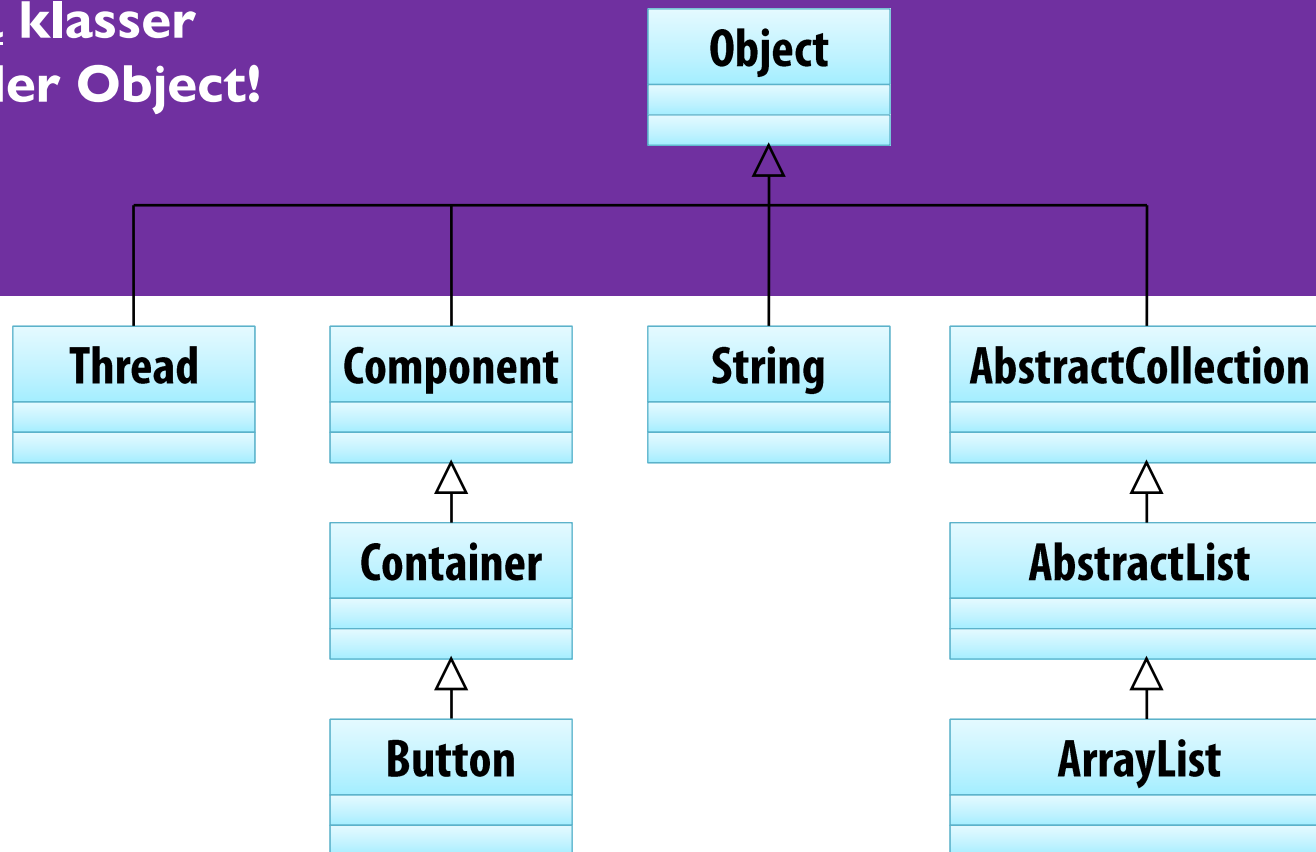
- Men en klass kan implementera **flera** gränssnitt



Klasshierarkins rötter

- I vissa språk har klasshierarkin **flera rötter**
 - Vissa klasser är **relaterade** via ärvning, andra **fristående**

Java: Alla klasser
finns under Object!



Utan "extends"...

```
class GenericList { }
```

Är ekvivalent med...

```
class GenericList extends Object { }
```

Med "extends"...

```
class DebugList extends MyList { }
```

```
class MyList extends ArrayList { }
```

```
class ArrayList extends GenericList { }
```

```
class GenericList { }
```

Så småningom kommer vi till Object!

- **Fördel:** Object-variabel kan peka på **vilket objekt som helst**
 - Användbart i **generella datatyper**
 - Set, List, ...
 - Vet inte typen av element i förväg

Implementationsarv och konstruktörer

- **new GraphicCircle**(1.0, 2.0, 3.0, c1, c2);
 - Först: minnesallokering, defaultvärden, för **alla** fält
 - **Ingen** får komma åt "gammalt skräp"

Object header:	(data)
x	0
y	0
r	0
outline	
fill	

Kryss = null
(pekar ingenstans)

Sedan släpps konstruktörer in:

Circle-delen "konstruerar sig"

GraphicCircle-delen "konstruerar sig"

I vilken ordning?

- **"Superklassen först"**
 - **Ingen** får komma åt **Circle**-fälten (x,y,r) innan **Circle** initialiserat dem
 - Inte ens dess egen subclass!

Konstruktörer och arv 2

- Men vi anropade ju **new** GraphicCircle(1.0, 2.0, 3.0, c1, c2);

```
public GraphicCircle(double x, double y, double r, Color outline, Color fill) {  
    super(x, y, r);  
    this.outline = outline;  
    this.fill = fill;  
}
```

Det första GraphicCircle gör:
Anropa superklassens konstruktor
→ Circle får initialisera sig

Sedan får GraphicCircle sin chans
att titta på objektet

Sedan får vi objektet från konstruktorn,
och kan titta på det

```
public GraphicCircle(double x, double y, double r, Color outline, Color fill) {  
    this.outline = outline;  
    this.fill = fill;  
}
```

Strunta i super(...) → *kompilatorn*
stoppar in super() utan argument

Men Circle() finns inte: Kompileringsfel!

Konstruktörer: Skicka vidare!

- Glöm inte att skicka vidare parametrar!
 - Ibland ser vi följande:

```
public class Circle {  
    public double x, y, r;  
}
```

Tom Circle-konstruktör

```
public class GraphicCircle extends Circle {  
    private Color outline, fill;  
    public GraphicCircle(double x, double y, double r, Color outline, Color fill) {  
        this.x = x;  
        this.y = y;  
        this.r = r;  
        this.outline = outline;  
        this.fill = fill;  
    }  
}
```

Anropar tom Circle-konstruktör
Petar själv in värden i Circle-delen

Dålig modellering!

Circle får inte bestämma över sig själv
Kan inte validera sina parametrar, ...
Kan inte ändra sin implementation
Upprepad kod i alla Circle-subklasser

Konstruktörer: Försvara klassen!

- Se till att Circle kan försvara sig!
 - Skapa bara konstruktörer som kräver rätt argument

```
public class Circle {  
    public double x, y, r;  
    public Circle(double x, double y, double r) {  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
}
```

Underklasser kan inte
låta bli att skicka x, y, r