Typhierarkier

Gränssnitt, ärvning, subtypspolymorfism, ...

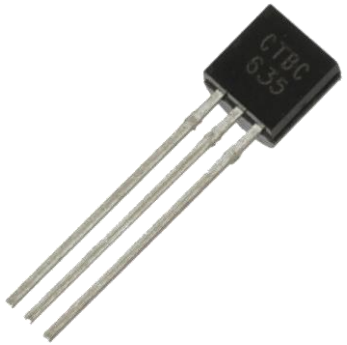
Hur används *hierarkier* för att modellera *nära relaterade typer*?

Motivering 1: Verkligheten

Klasser med olikheter

jonkv@ida

- Vi har pratat om **klasser** av objekt



Tydliga olikheter

➔

separata klasser

Liknande objekt

jonkv@ida

- Men om objekten är "**ganska lika**"?
  - **En** klass: Generella **motorfordon**?
  - **Tre** klasser: Bilar, bussar och lastbilar?
  - **Fem** klasser: Bil, lätt lastbil, tung lastbil, minibuss, buss?

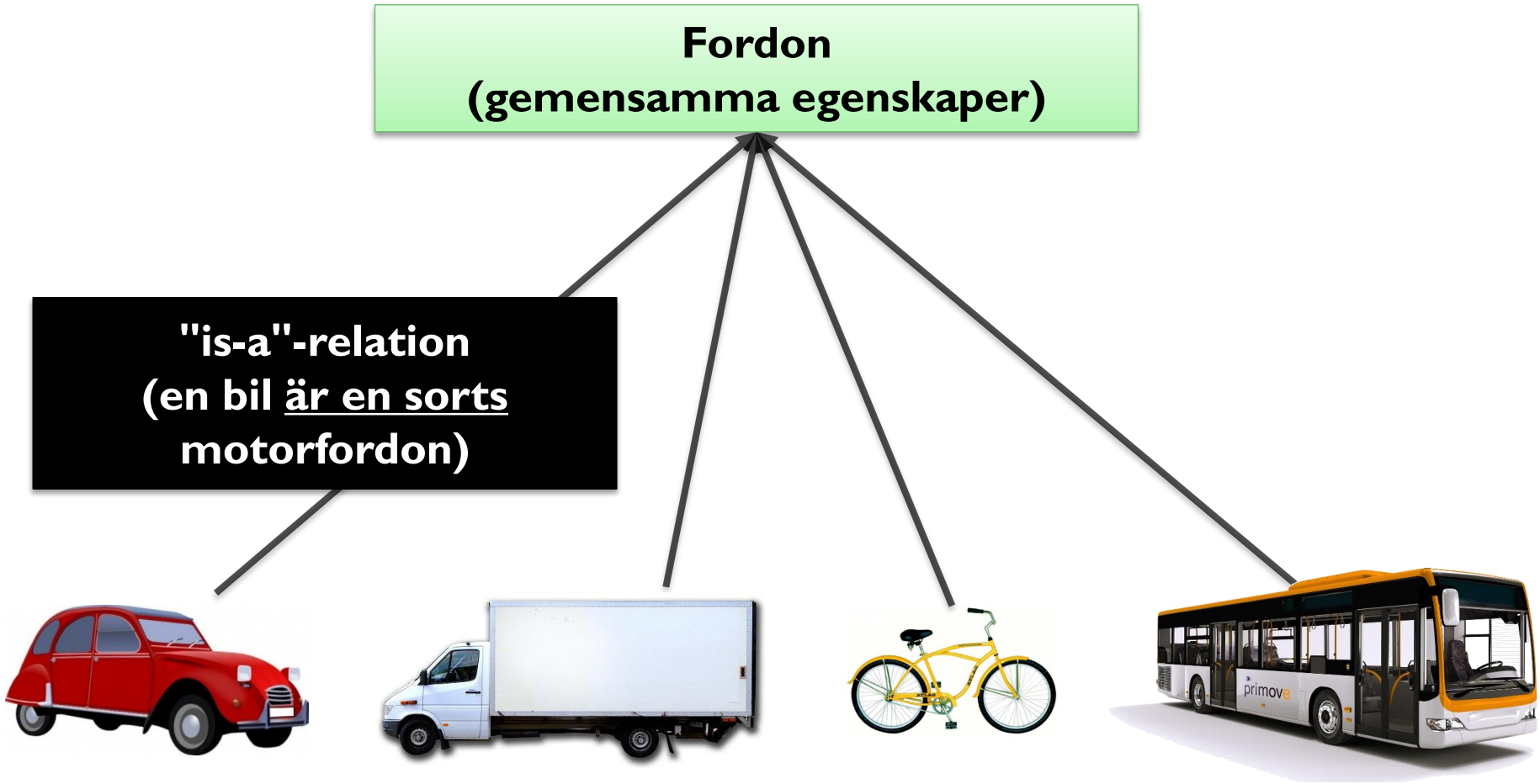
Bra: Vi kan **generalisera**…
"Detta gäller alla motorfordon"

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

Hierarkier av klasser

jonkv@ida

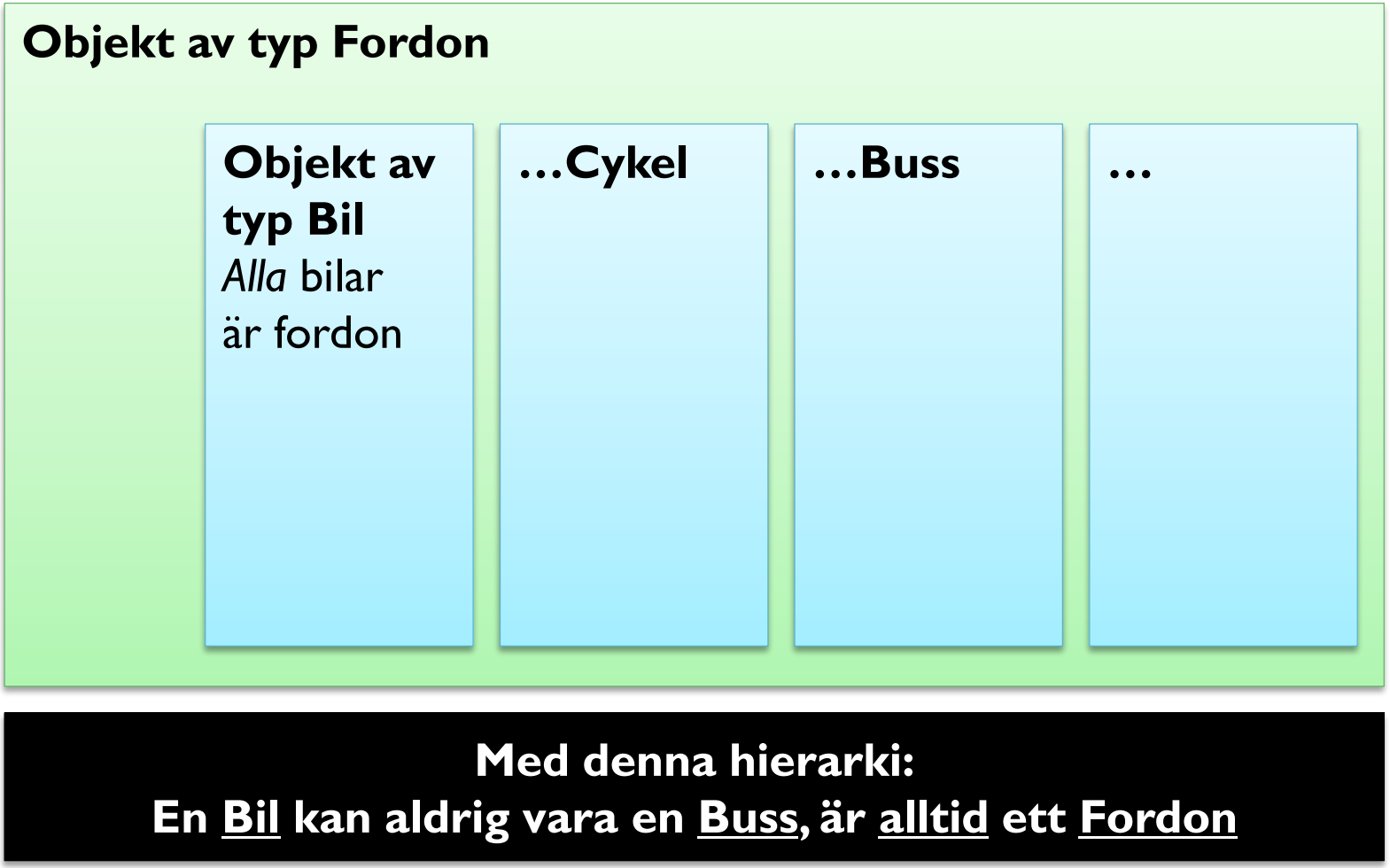
- Vi kan ha **hierarkier** med **både** generella och specifika klasser!



Hierarkier och delmängder

jonkv@ida

- En bil **är ett** fordon…



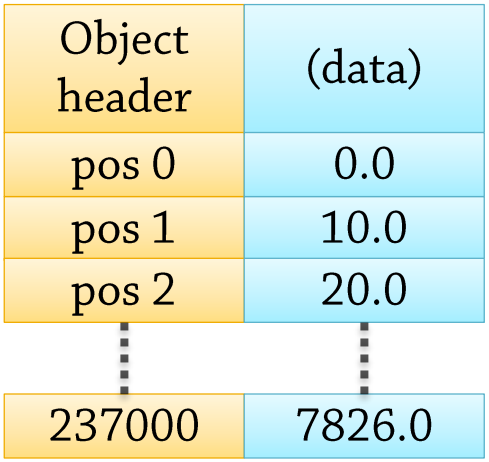
Motivation 2: Datastrukturer

Liknande datatyper (1)

jonkv@ida

- Flera sätt att lagra **listor**!

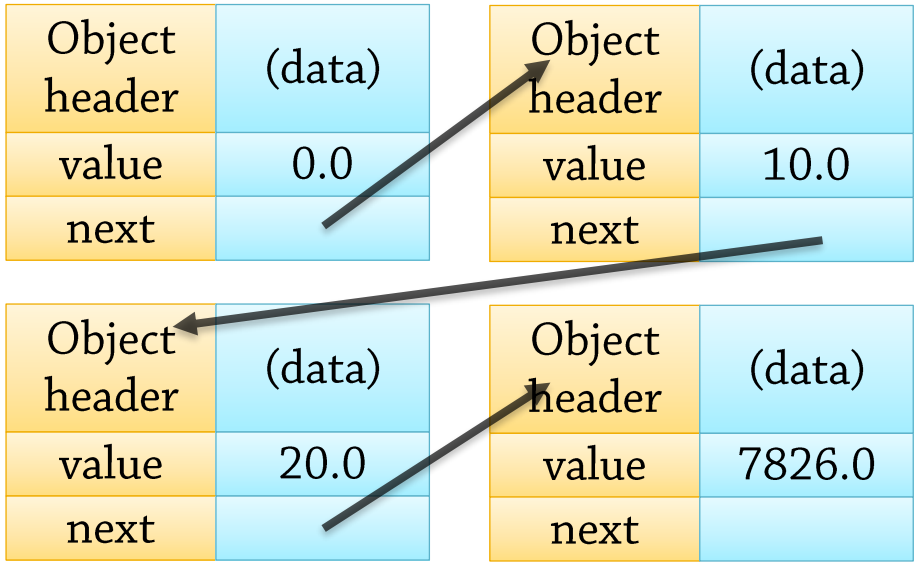
"Linjärt" i minnet…



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)

jonkv@ida

- **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	ArrayList	SkipList
		…
size(): int **get**(int pos): Object **add**(Object element): Object **set**(int pos, Object element): void **insert**(…): void	**size**(): int **get**(int pos): Object **add**(Object element): Object **set**(int pos, Object element): void	**size**(): int **get**(int pos): Object **add**(Object element): Object **set**(int pos, Object element): void

Liknande datatyper (3)

jonkv@ida

- 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

jonkv@ida

Python

```
class ArrayList:
    def extend(self, list):
        …

class LinkedList:
    def extend(self, list):
        …

def add_lists(list1, list2):
    list1.extend(list2)
    return list1
```

Vi har en ArrayList-klass

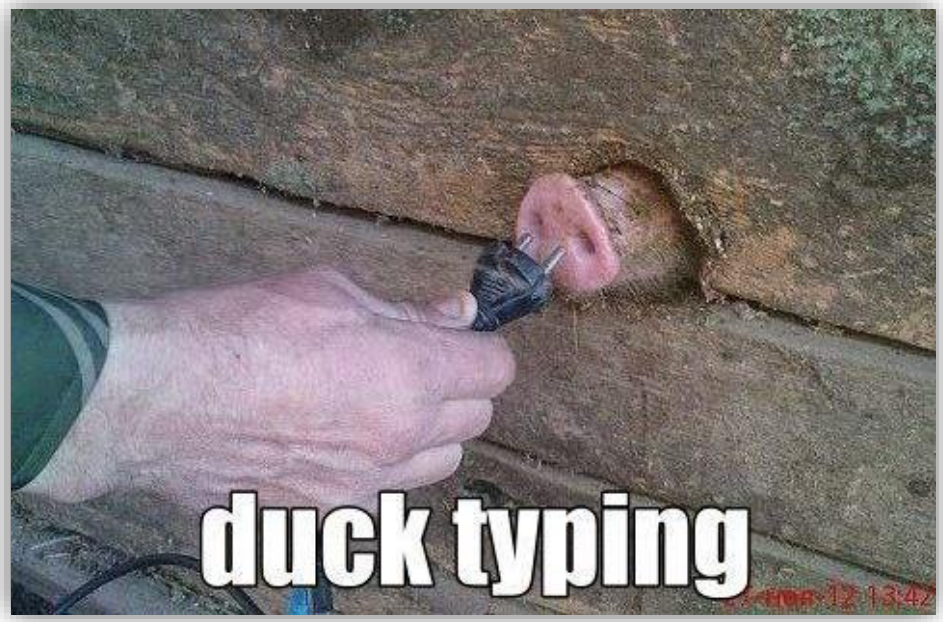
Parametertyp anges inte – kontrollera vid körning om objektet har extend()!

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



Utan Duck Typing

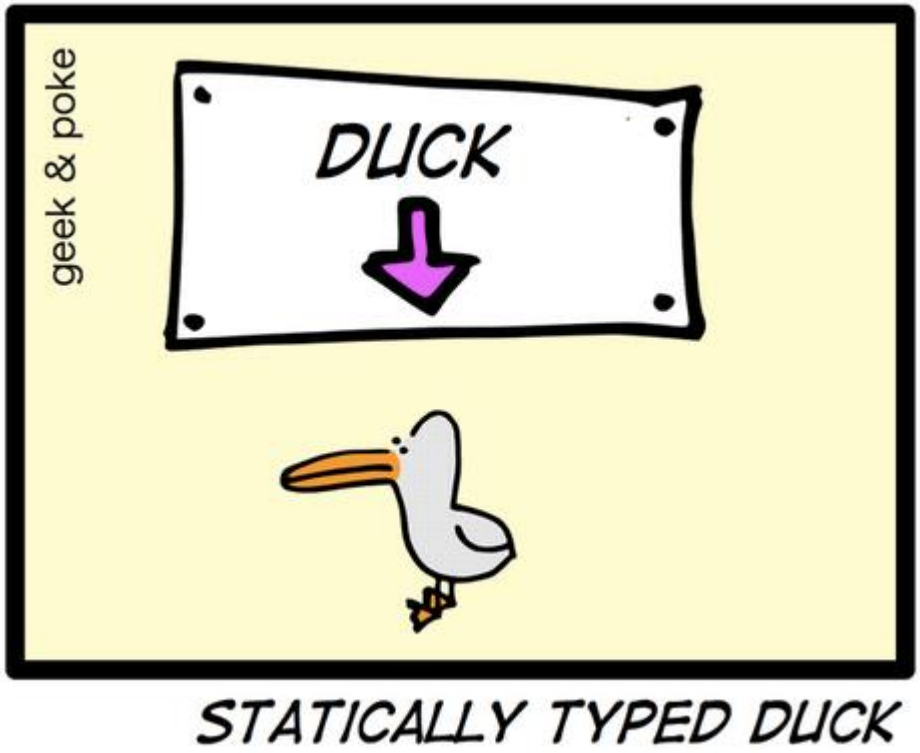
jonkv@ida

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”

Hierarkier

jonkv@ida

- Lösning: Vill ha hierarkier även här

Enkla typhierarkier i Java: Gränssnitt (interface)

Interface: Intro

- Ett **interface** (*gränssnitt*) 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);
}
```

Dokumentation
anger *krav* för alla
som implementerar List

Metodsignaturer:
Returtyp, namn, parametrar
Inga fält, ingen metodkod…

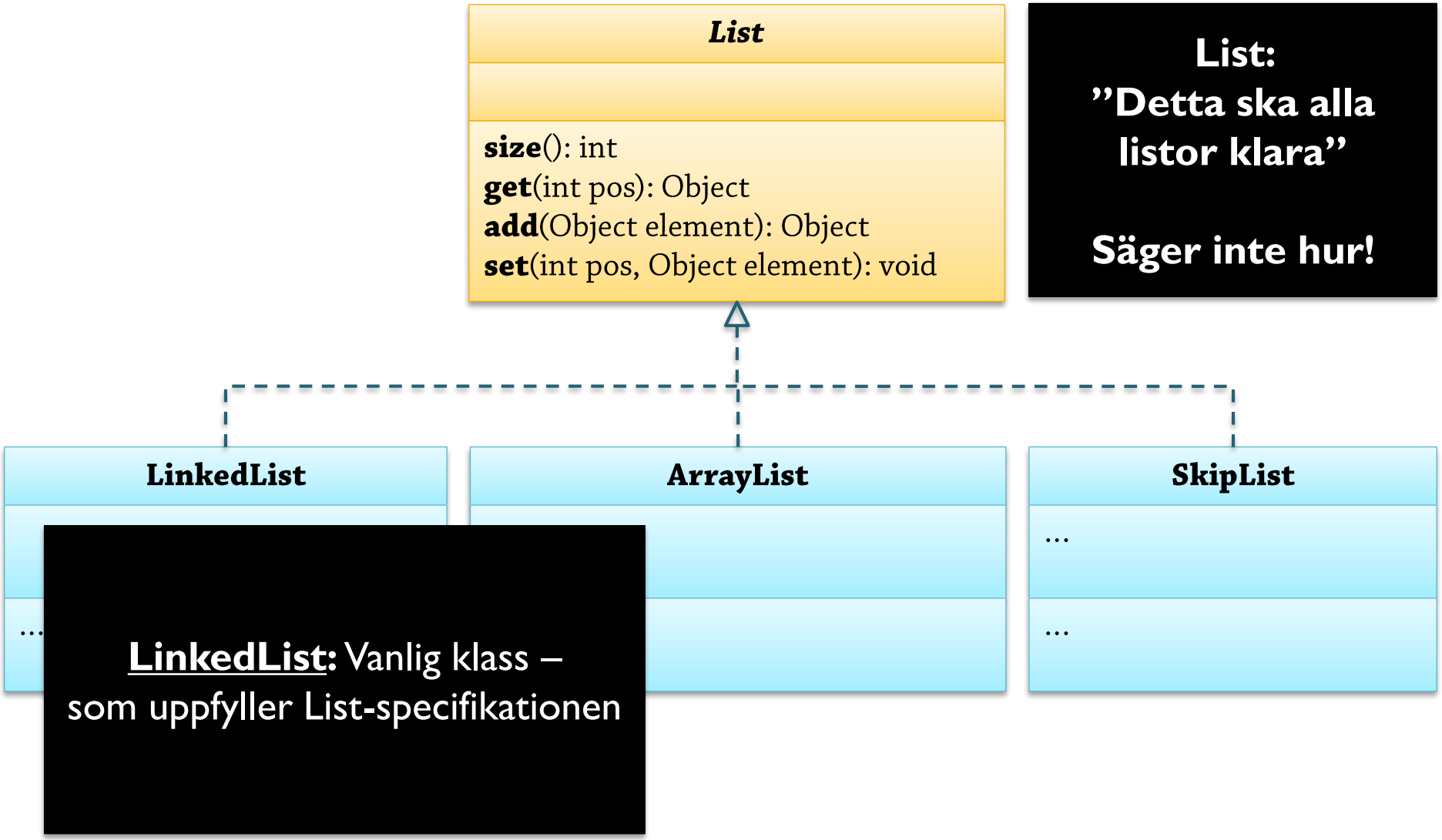
Metoder i ett gränssnitt
är alltid (underförstått):
public – tillgängliga för alla
abstract – *ej implementerade här*

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

Interface: Hierarkier 1

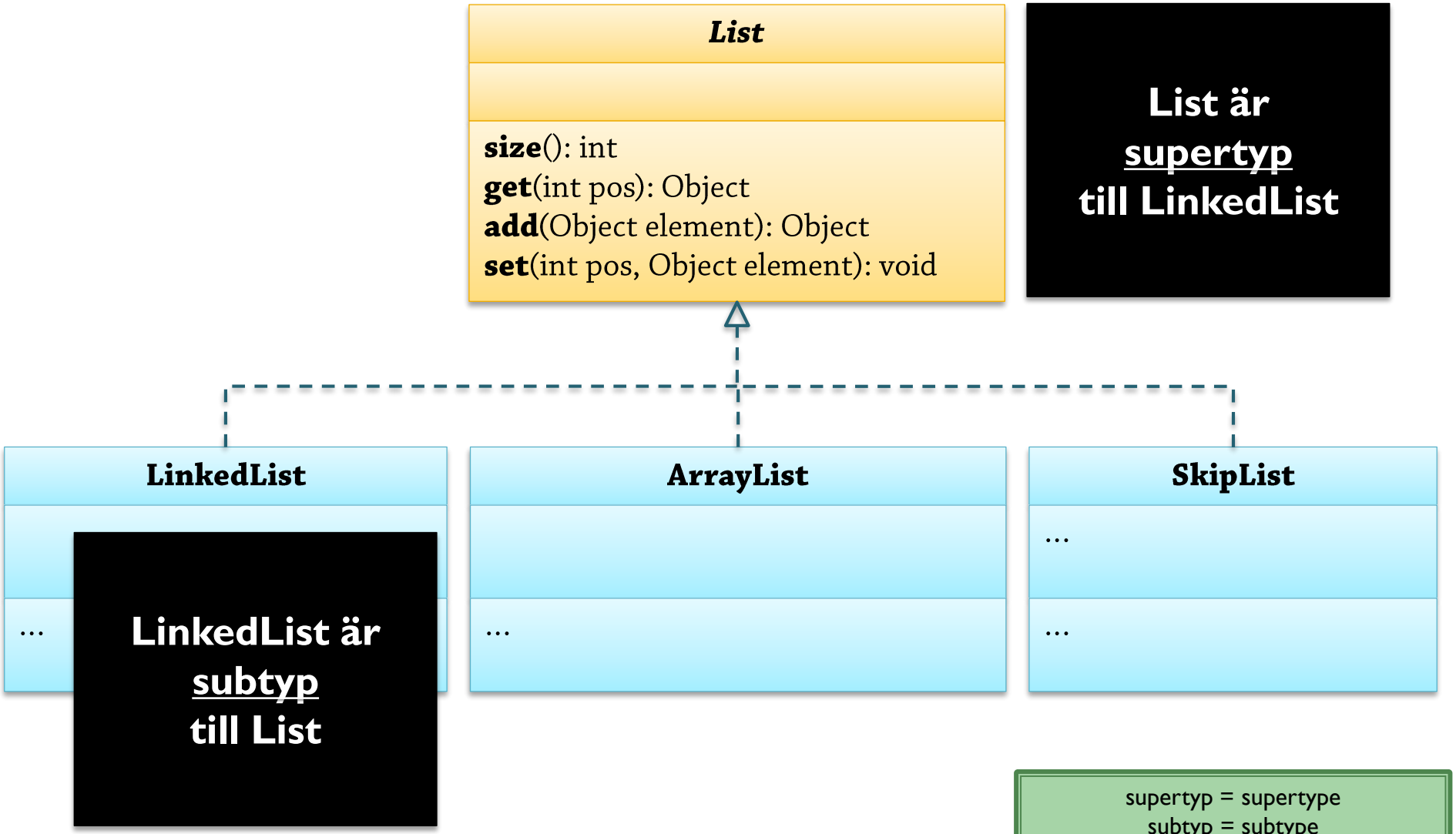
jonkv@ida

- Möjliggör en **hierarki** av **datatyper**!



Interface: Hierarkier 2

- Terminologi:



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

jonkv@ida

- En Java-klass kan **implementera** ett gränssnitt

```
public class ArrayList implements List {
   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

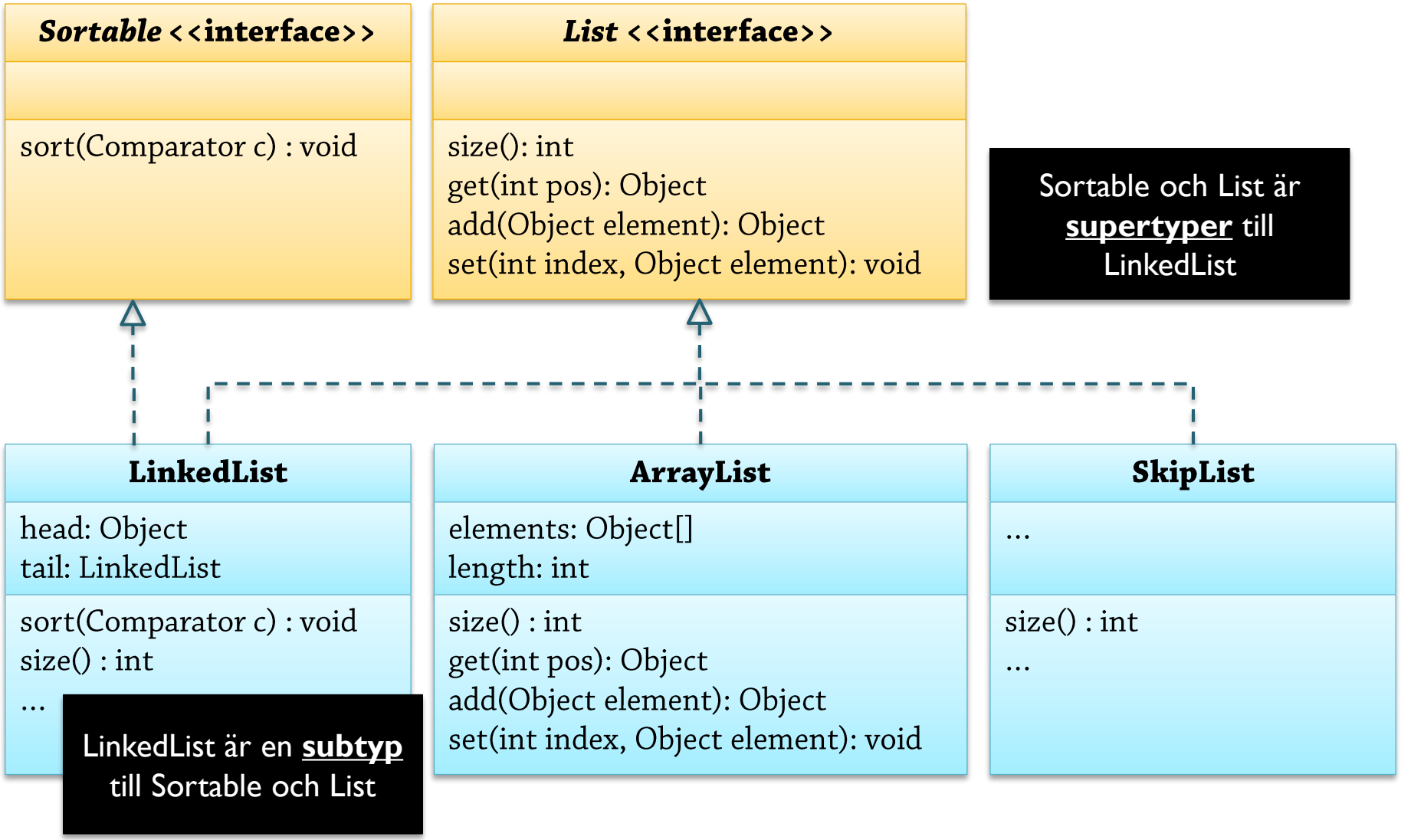
Kan ha fler metoder än gränssnittet
(håller ändå vad den lovar)

implementera = implement

Interface 3: Typhierarki

jonkv@ida

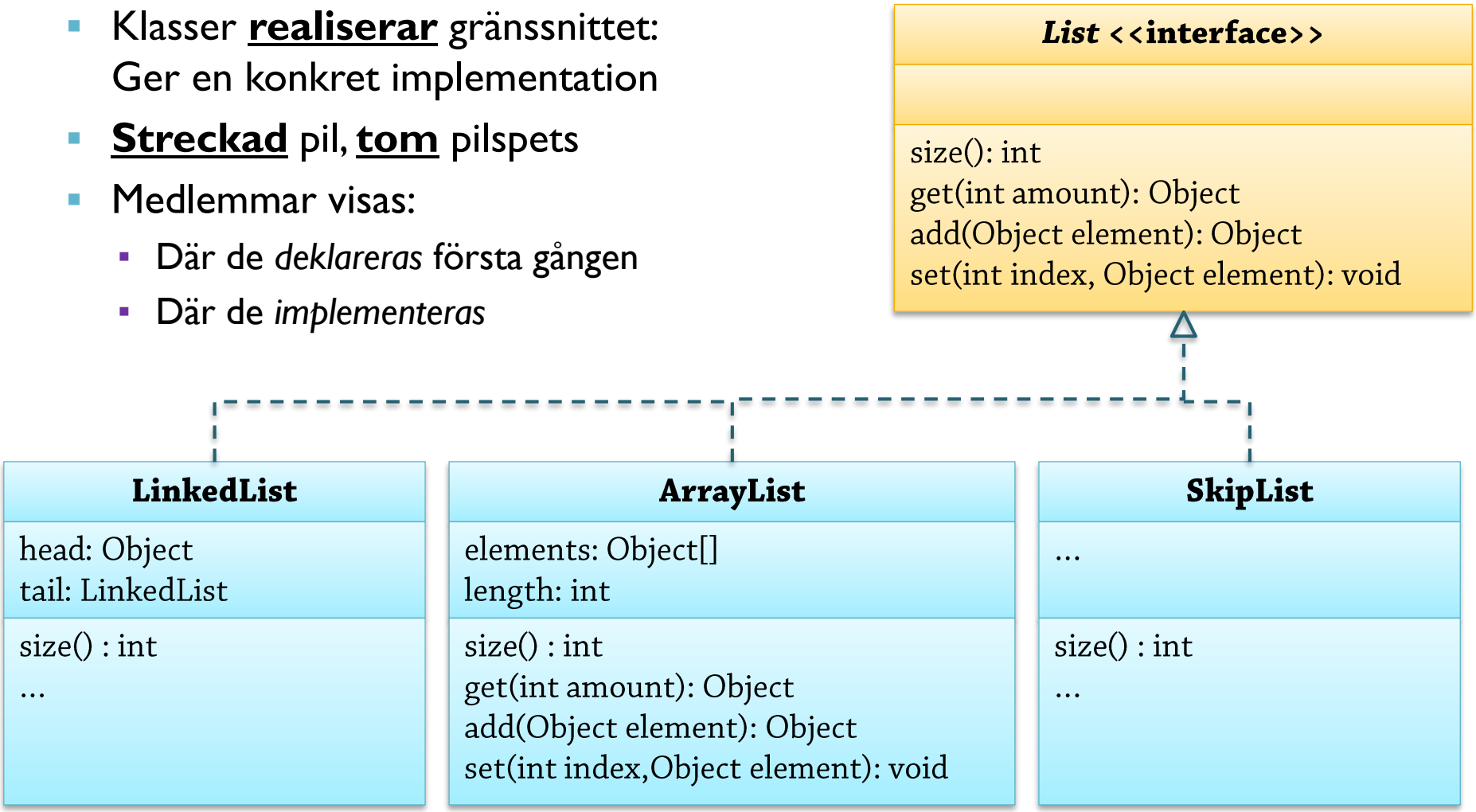
- En klass kan implementera **flera** gränssnitt



UML: Klassdiagram med gränssnitt

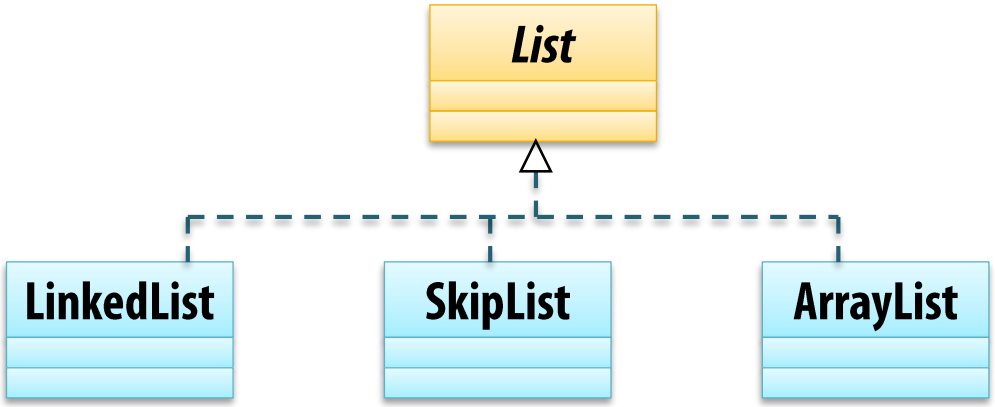
jonkv@ida

- **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 *deklareras* första gången
    - Där de *implementeras*



Om man har flera "klassvarianter" med gemensam bas:

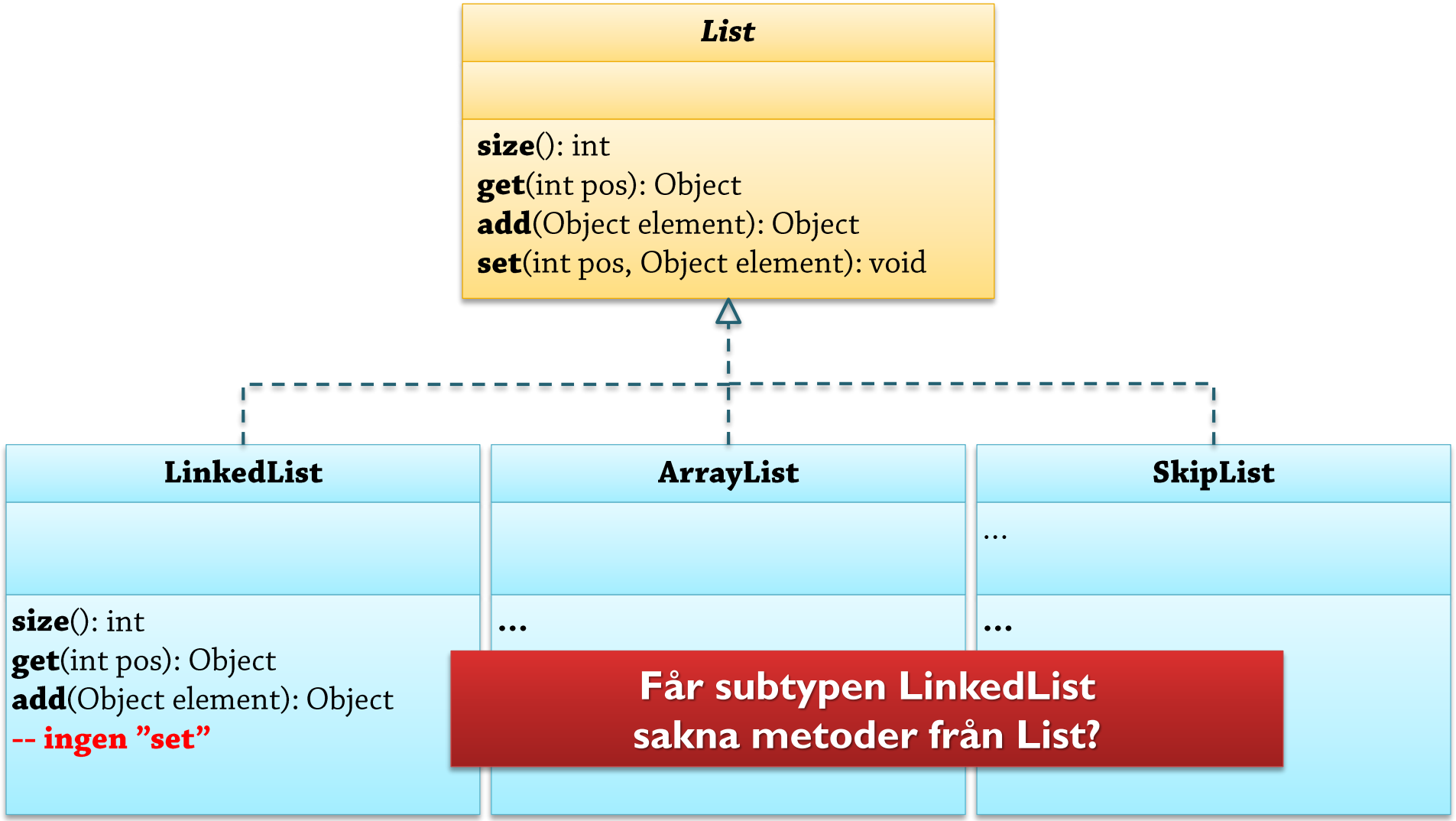
Oftast bör man ha ett gränssnitt,
så den gemensamma basen kan användas



Hur ska en typhierarki se ut?
Kontrakt och Liskov Substitution Principle

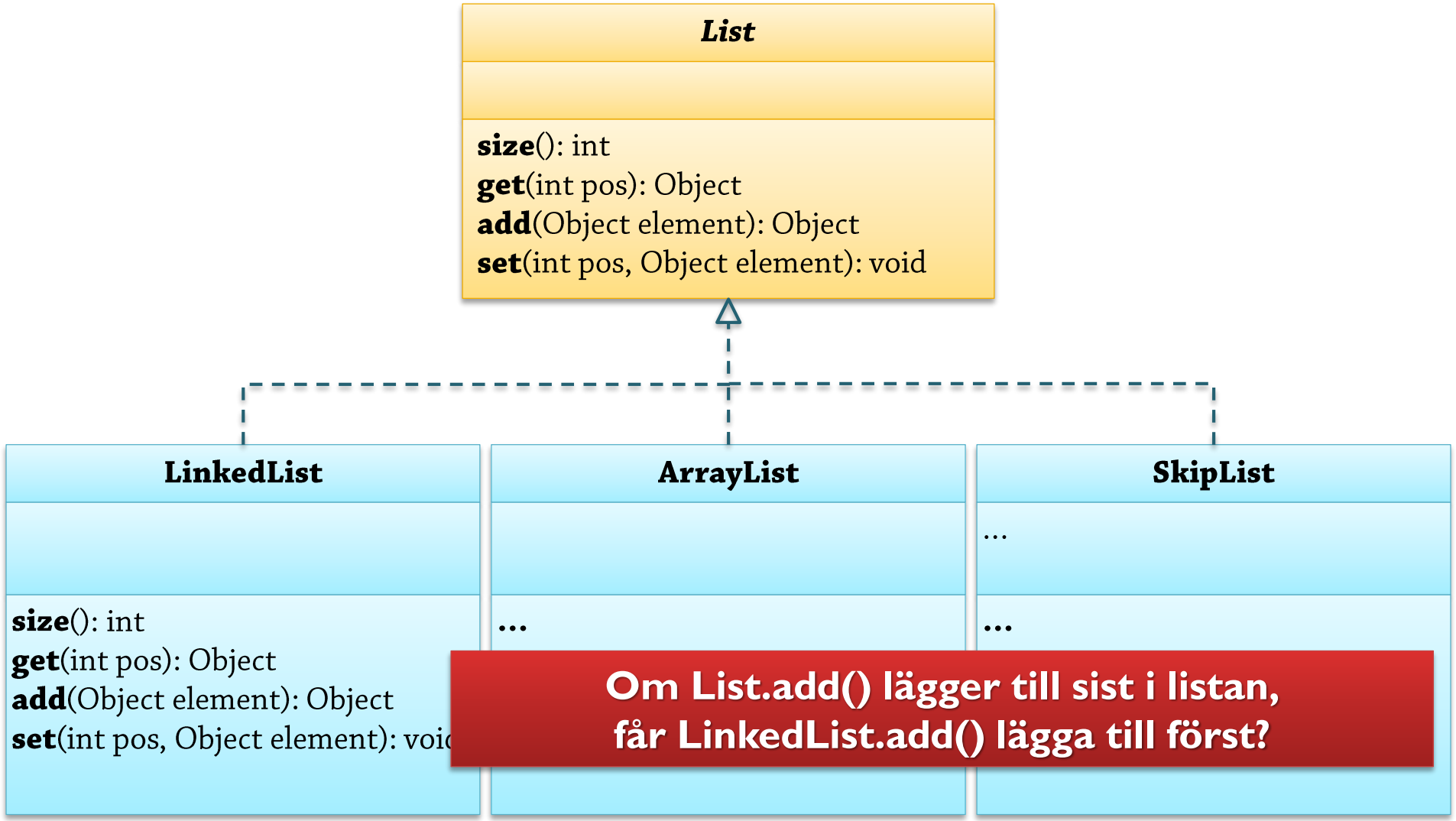
Krav på hierarkier 1

jonkv@ida



Krav på hierarkier 2

jonkv@ida



Krav på hierarkier 3

- Även **List** har ett **kontrakt**
  - En ArrayList **är** en List
    ➔ måste uppfylla samma kontrakt!
  - (Fåglar har fjädrar,
    ankor är fåglar
    ➔ ankor har fjädrar)

- Subtypens eget kontrakt får bara:
  - **Lova mer** – ge *starkare* garantier (ankor kan kvacka också)
  - **Kräva mindre** – ha *svagare* krav

- Det är detta som garanterar:
  - Om vi accepterar en **List**,
    är vi **nöjda** med
    vilken subtyp som helst!

```
void quicksort(List someList) {
   // Vi programmerar
   // utifrån List:s kontrakt

   // Detta uppfylls oavsett om vi får en
   // ArrayList, SkipList, LinkedList, ...
}
```

Krav på hierarkier 4: LSP

- En formell beskrivning: **Liskov Substitution Principle**
  - *Let q(x) be a property provable about objects x of type T.*
    Then q(y) should be provable for objects y of type S,
    where S is a subtype of T.
  - **Om** vi kan bevisa något om alla List-objekt,
    måste detta även gälla för alla LinkedList-objekt.

Barbara Liskov

Kontrakt, implementation och kovarianta returtyper

Kovarianta returtyper!

jonkv@ida

- LSP säger:
  - Subtypens eget kontrakt får bara:
    - **Lova mer** – ge *starkare* garantier
    - **Kräva mindre** – ha *svagare* krav

Java låter subklasser stärka löftet om returtyp

```
public interface List
{
   List makeCopy();
   List convert();
}
```

```
public class ArrayList implements List
{
   ArrayList makeCopy() { … }
   SkipList convert() { … }
}
```

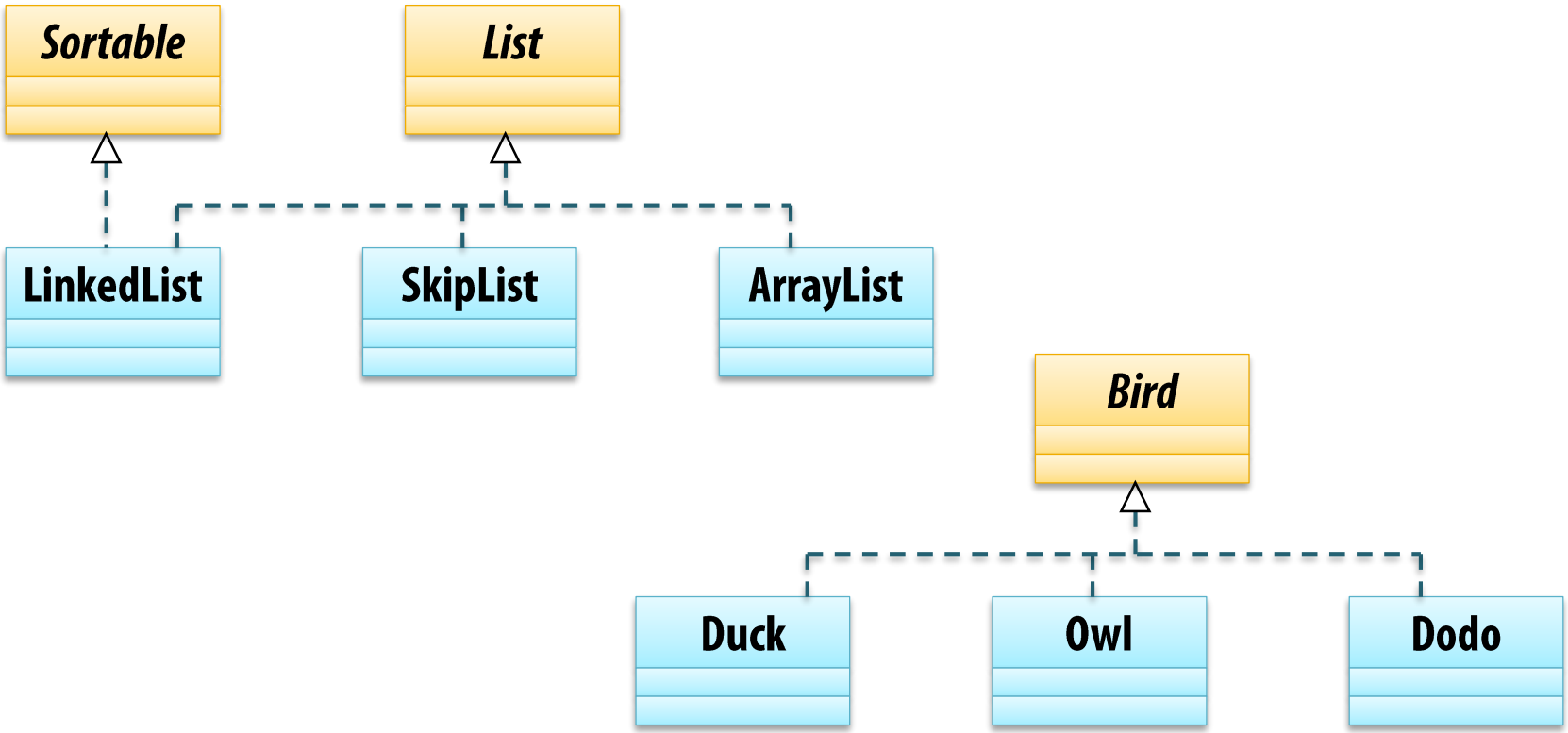
Jag lovar att det är en lista – det är till och med en ArrayList!

Varierar "åt samma håll"
(en *specialisering* av List *specialiserar* även returtyperna)
➔ *"kovariant"*

Statiska och dynamiska typer, statisk och dynamisk typkontroll

Objekt med flera typer

- Ett objekt har nu **flera typer**
  - Varje **LinkedList** är **också** av typen **List** och **Sortable**



Vilka effekter får det?

Peka på en subtyp – ett par frågor

jonkv@ida

- En **objektvariabel** kan peka på objekt av **godtycklig subtyp**
  - `List names = new ArrayList();`
    `Bird myBird = new Duck();`

names kan bara peka på List-objekt…
Och en ArrayList är ju en List!

Så vilken typ har variabeln egentligen?

- En **objektvariabel** kan pekas om till ett objekt av **annan typ**
  - `List numbers = new SkipList();`
    `…`
    `numbers = new LinkedList();`

Både SkipList och LinkedList
är ju specialfall av List…

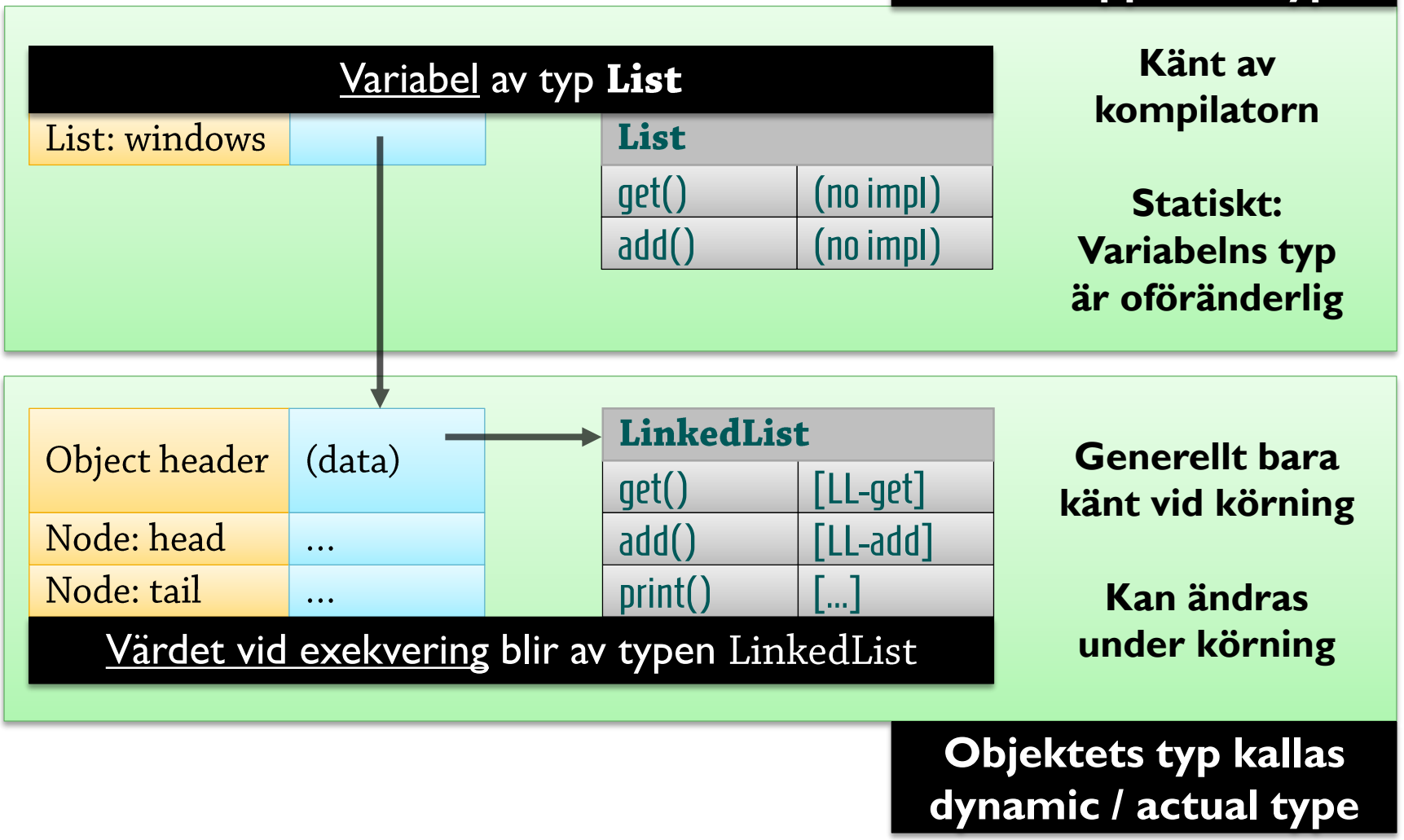
Så kan variabler byta typ?

Typer: Apparent / Actual

jonkv@ida

- Med typhierarkier:
  - List windows = application.getWindows();

Variabelns typ kallas static / apparent type



Objektets typ kallas dynamic / actual type

Dynamisk typ: Inte känd före körning

jonkv@ida

```
public class ListHandler {
   public void useList(List elements) {
      // Vilken sorts lista är elements?
   }
}
```

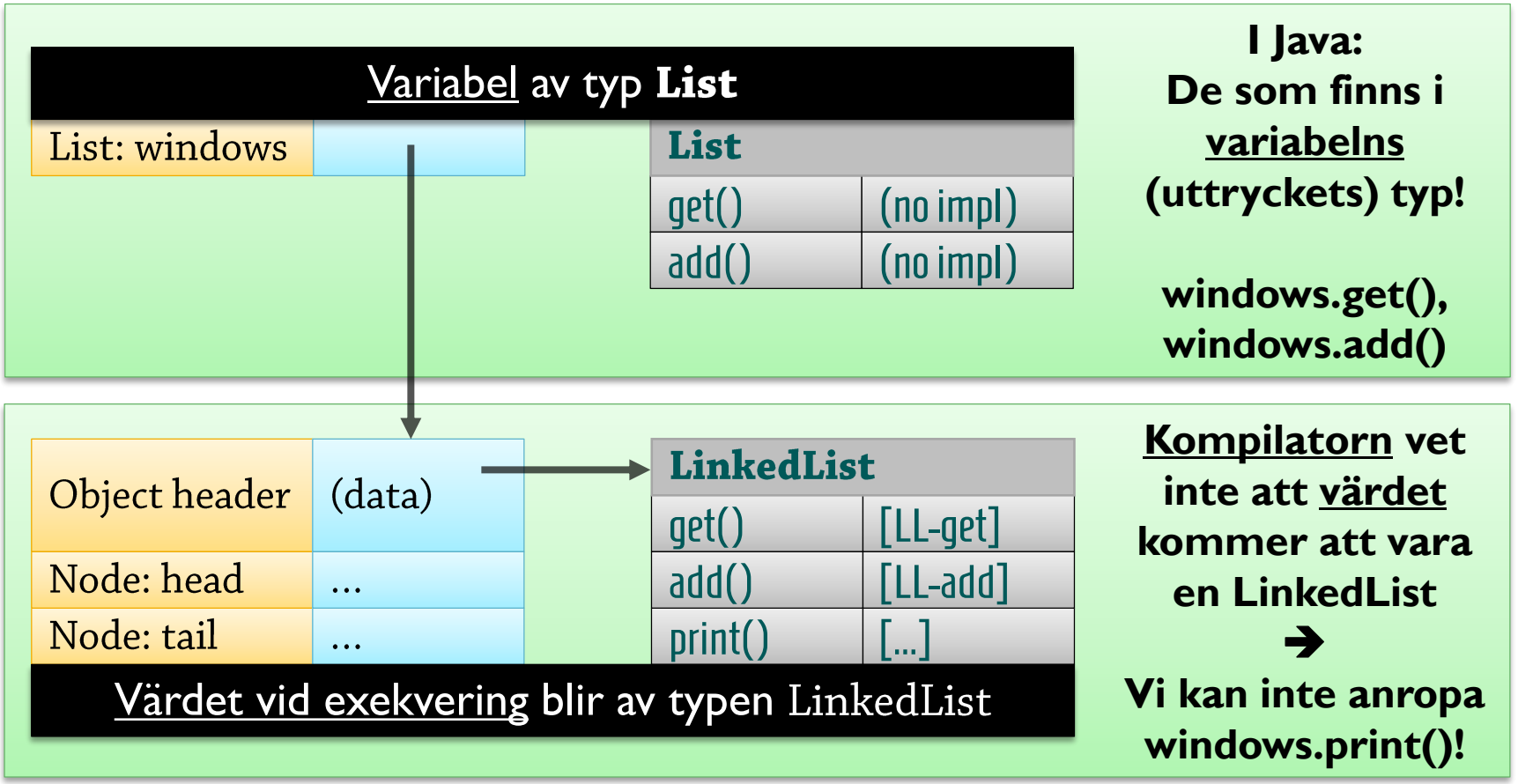
```
public class Randomizer {
   public void useList() {
      Random rnd = new Random();
      List list;
      if (rnd.nextBoolean()) {
         list = new ArrayList();
      } else {
         list = new LinkedList();
      }
      list.add(...);
   }
}
```

Kan **inte** avgöra typen i förväg...

Synliga (tillgängliga) medlemmar

jonkv@ida

- Givet en variabel, **vilka metoder och fält** är **synliga**?
  - List windows = application.getWindows();



Statisk typkontroll

jonkv@ida

- Så **typkontrollen** är **fortfarande statisk**

```
public class Application {
   public List     getWindows() { … }
   public SkipList getFiles() { … }
}
```

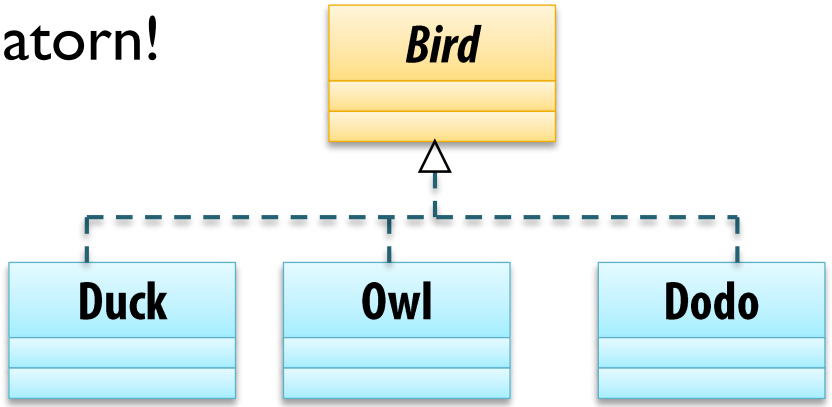
```
Application app     = …;
List windows        = app.getWindows();
List files          = app.getFiles();
```

Kompilatorn testar:
getFiles() returnerar SkipList,
files är av typ List
➔ Allt OK!

Vem vet mest?

jonkv@ida

- Men tänk om **vi vet mer** än kompilatorn!
  - Vi har listor av fåglar
    - ```
      public class BirdList {
          public void add(Bird foo) { … }
          public Bird get(int index) { … }
      }
      ```
  - Vi stoppar in något först i listan
    - ```
      BirdList lista = new BirdList();
      lista.add(new Owl());
      ```
  - Vi plockar ut det
    - ```
      Bird a = lista.get(0);        // Kompilatorn accepterar
      ```
    - ```
      Owl b = lista.get(0);         // Fel! Metoden get() lovar bara att ge oss en Bird,
                                    // och inte alla fåglar är ugglor
      ```



Men vi vet ju vad vi stoppade in där!

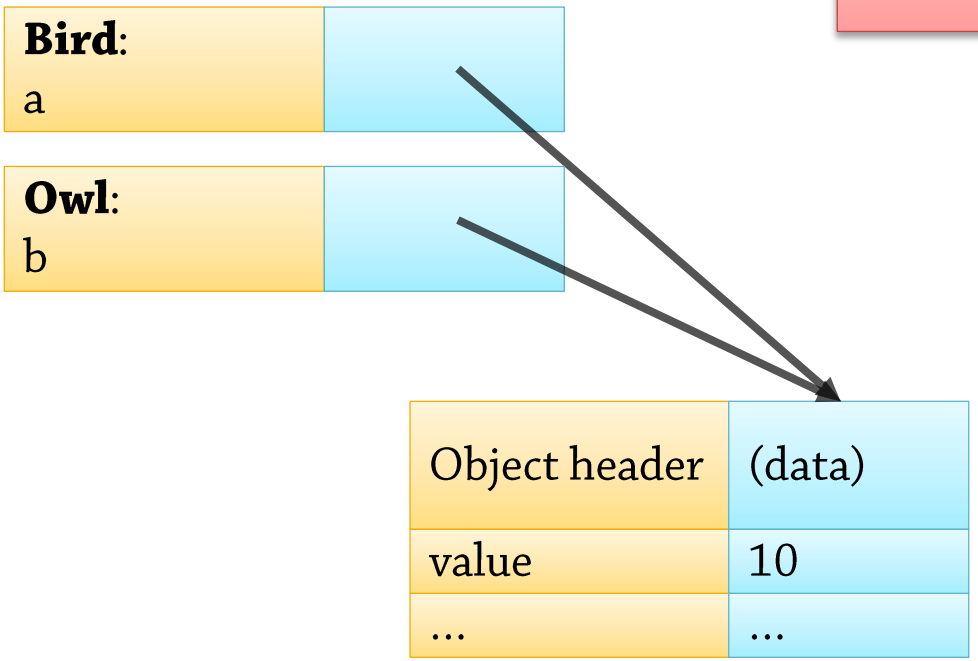
Att veta mer (2)

jonkv@ida

- Vi kan **tala om för kompilatorn** vad vi vet!
  - Använd en **cast**
    - `Owl b = (Owl) lista.get(0);`
  - Resultatet är en **ny pekare** till **samma objekt**
    - Inte ett nytt "konverterat" objekt!

Denna cast är ett *löfte*:
"Fågeln **är** faktiskt en uggla!"

Om man **ljuger**: **ClassCastException**
(**här** finns **dynamisk** typkontroll,
dvs. vid körningen!)



Dynamisk bindning, subtypspolymorfism

Namnbindning

jonkv@ida

- **Namnbindning**:
  - Givet en **identifierare** (namn), **vilken** variabel / fält / metod / … menar vi?
  - För **variabler** tar *kompilatorn* reda på detta:
    -
```
public class Binding1 {
    public void foo() {
        int index = 100;
        int other = 200;
        System.out.println(index);  // "index" binds till metodens första variabel
    }
}
```
    -
```
public class Binding2 {
    public int index = 100;
    public void foo() {
        int other = 200;
        System.out.println(index);  // "index" binds till objektets första fält
    }
}
```

Detta är statisk bindning = tidig bindning:
Sker före programmet körs

Bindning av metoder

jonkv@ida

- Att binda **metodnamn** till **implementation**:

Utan typhierarkier (annat språk)

- Variabelns typ = objektets typ
  ➔ **Statisk bindning** är möjlig

```
ArrayList myList = ...;
myList.add(...);
```

Vilken variant av add()?
Måste vara ArrayList:s!
Känt vid kompileringen:
Namnet add() kan
statiskt bindas
till en funktion av kompilatorn

Med typhierarkier (Java)

- Variabelns typ != objektets typ
  ➔ **Dynamisk bindning** krävs

```
List myList = ...;
myList.add(...);
```

Vilken variant av add()?
ArrayList:s? LinkedList:s?
Okänt vid kompileringen:
Namnet add() måste
dynamiskt bindas
till en funktion vid körning

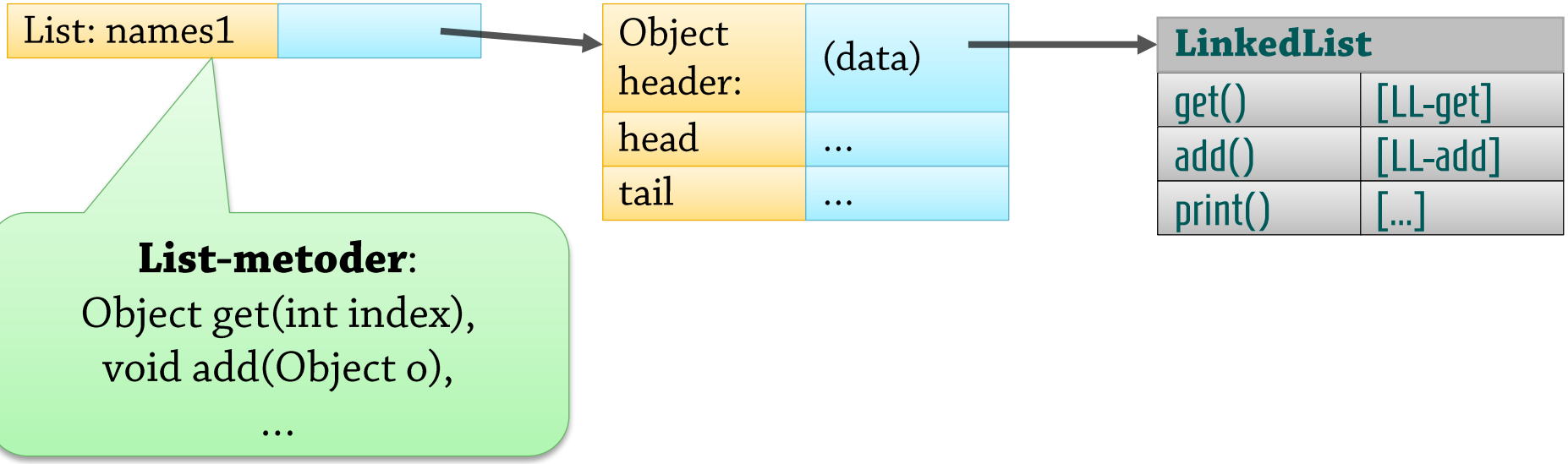
Metodtabeller

jonkv@ida

- Varje objekt känner till sin **klass**
  - Varje klass känner till sin **metodkod**

Klassinfo med metodtabell

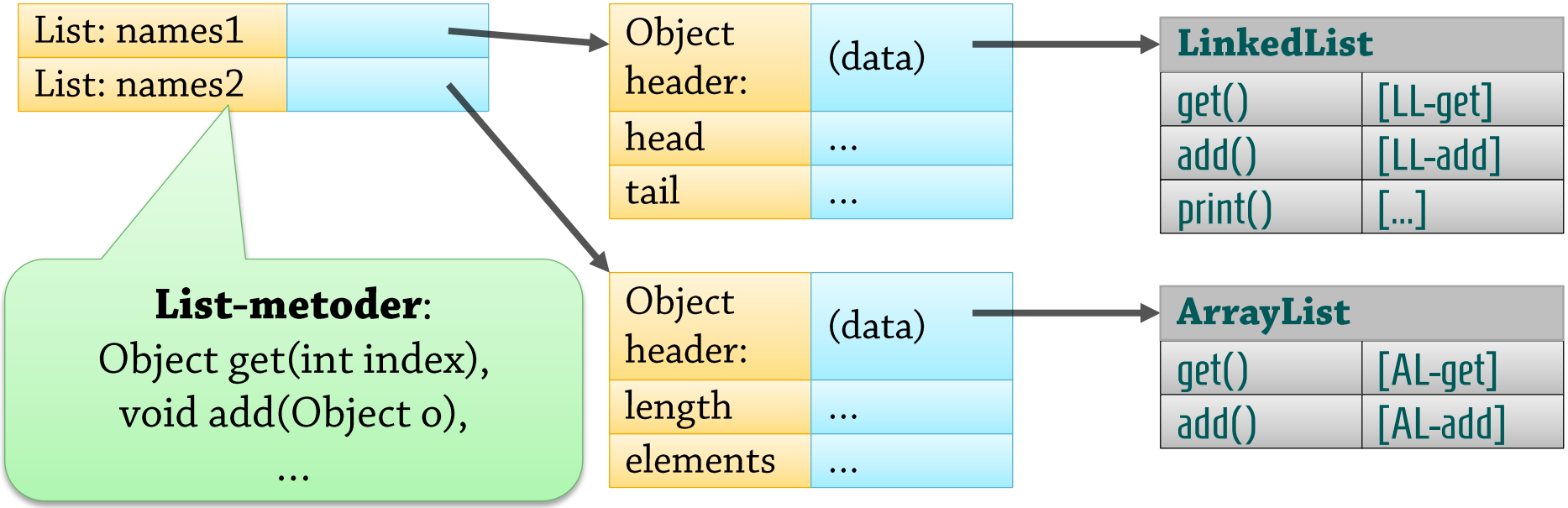
Virtual method table: vtable



Dynamisk bindning

jonkv@ida

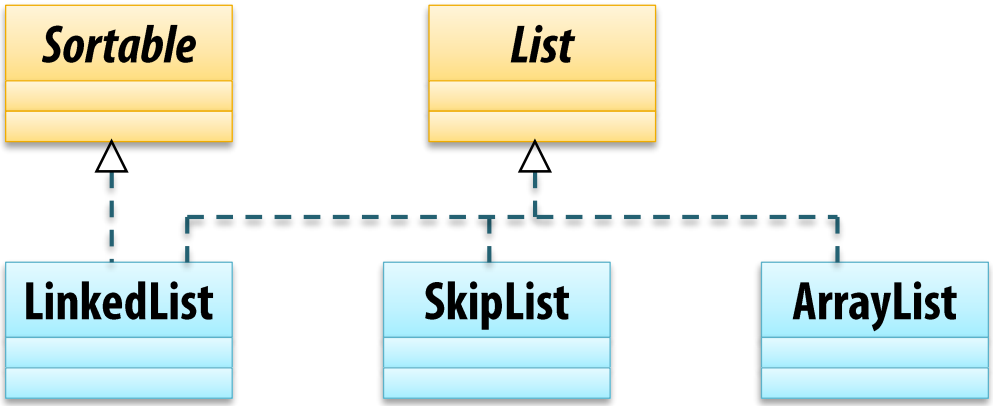
- **Dynamisk bindning** = **sen bindning** = **dynamisk dispatch**
  - *Objektets* typ avgör
    - List names1 = foo.getNames(); // Returnerar en **LinkedList**
      names1.add(n1); // LinkedList-implementationen av add
    - List names2 = bar.getNames(); // Returnerar en **ArrayList**
      names2.add(n1); // ArrayList-implementationen av add
  - *Objektet* talar (indirekt) om för oss **var implementationen finns**:



Typhierarkin + sen bindning ger oss…

Subtypspolymorfism

- πολυμορφισμός = som har flera former
- Här:
  - Ett **metodnamn**, deklarerat i en typ
  - Flera **implementationer**, skrivna i **subtyperna**
  - Kallas ofta enbart "polymorfism" – men polymorfism är egentligen bredare begrepp



Overloading

jonkv@ida

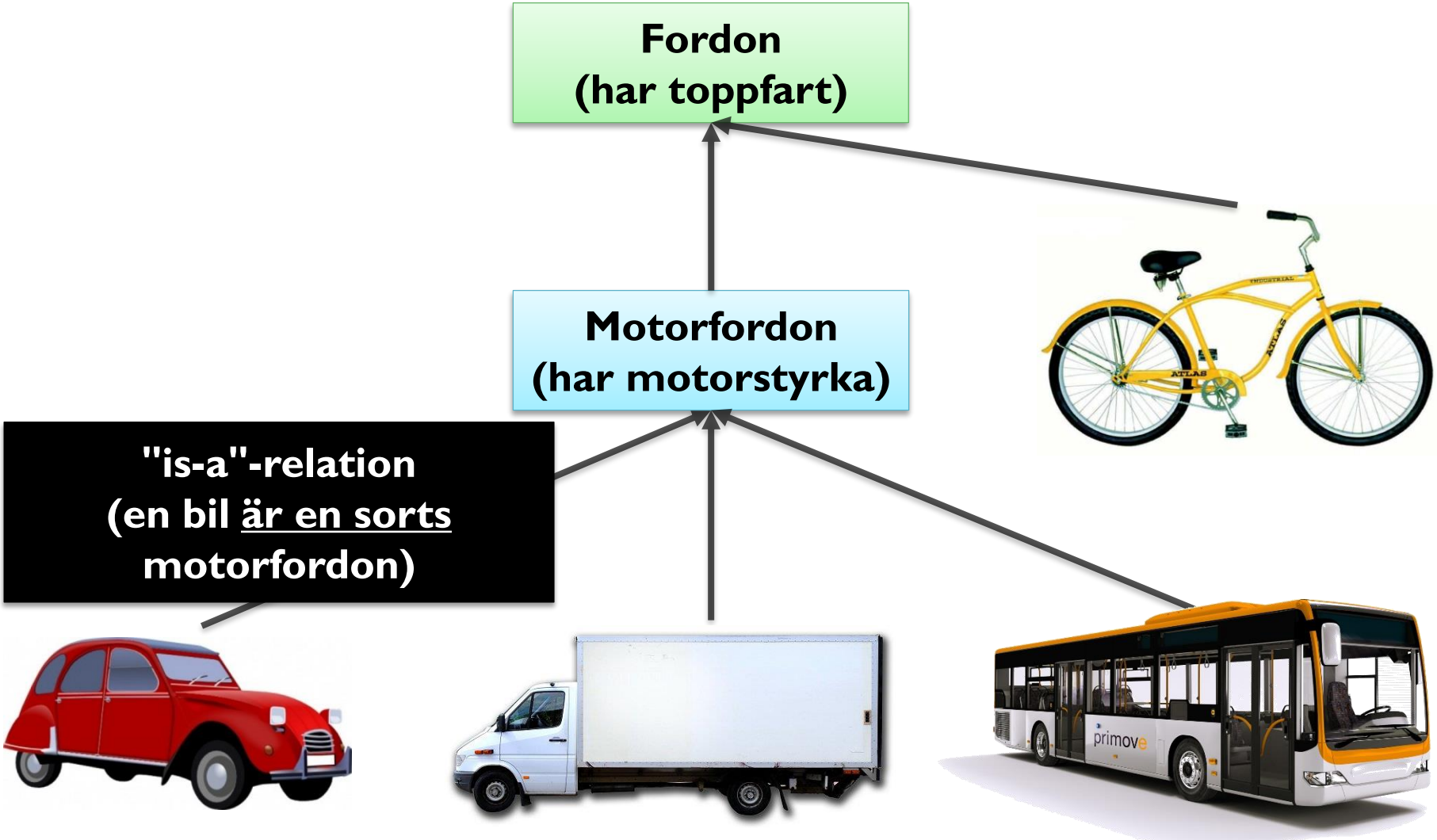
- Kontrasteras med **överlagring** = **ad-hoc-polymorfism**
  - Samma namn, flera implementationer (för olika argumenttyper)
    - ```
      public class Printer {
          public void print(int val) { … }
          public void print(double val) { … }
          public void print(double val, int precision) { … }
      }
      ```
    - Inte ett OO-begrepp – mer allmänt!
  - Alla varianter är kända vid kompileringen
    - Ger **statisk bindning**
      ```

Ärvning mellan gränssnitt

Hierarkier av klasser

jonkv@ida

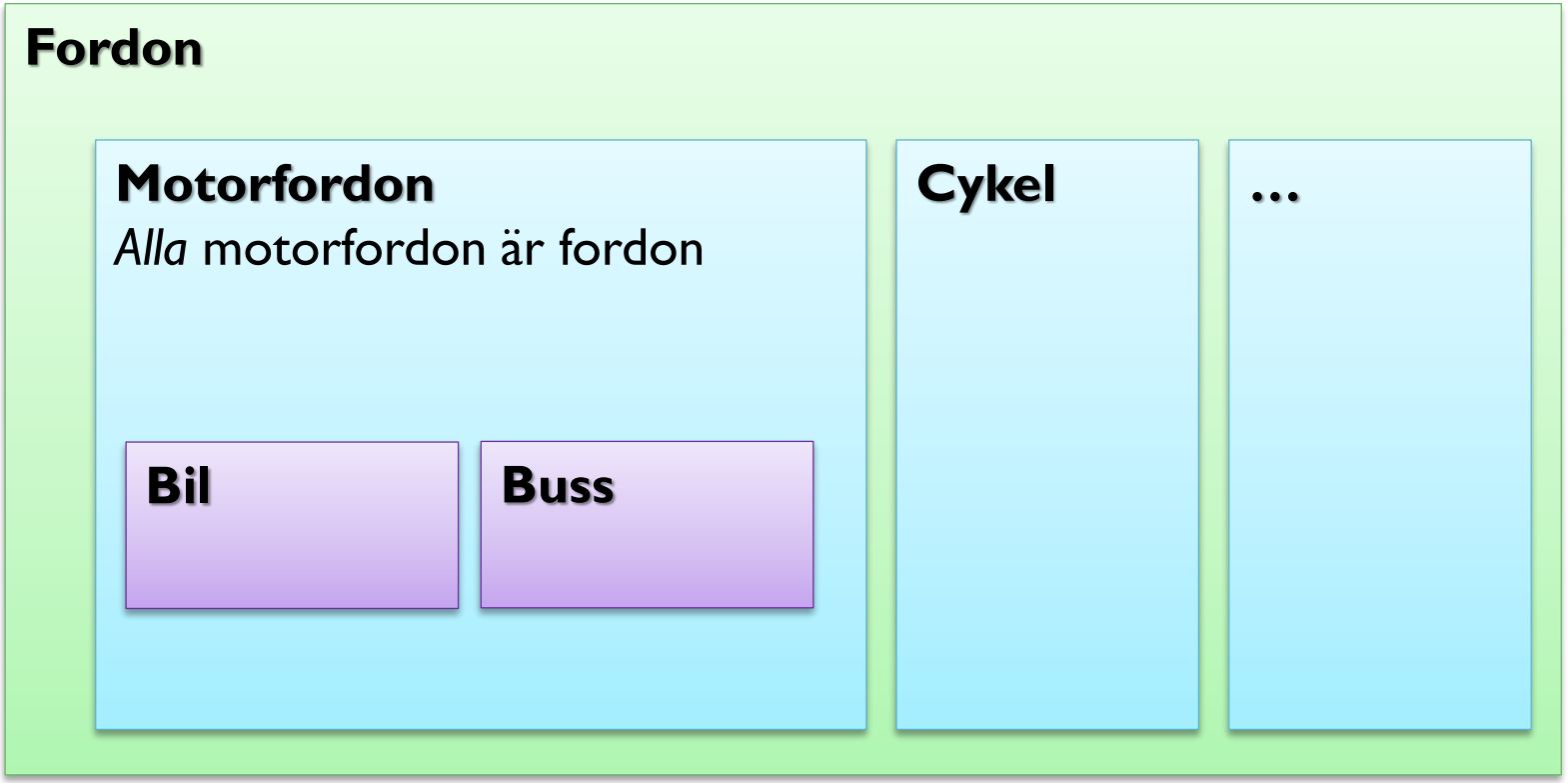
- Hierarkier kan ha **flera nivåer**



Hierarkier och delmängder

jonkv@ida

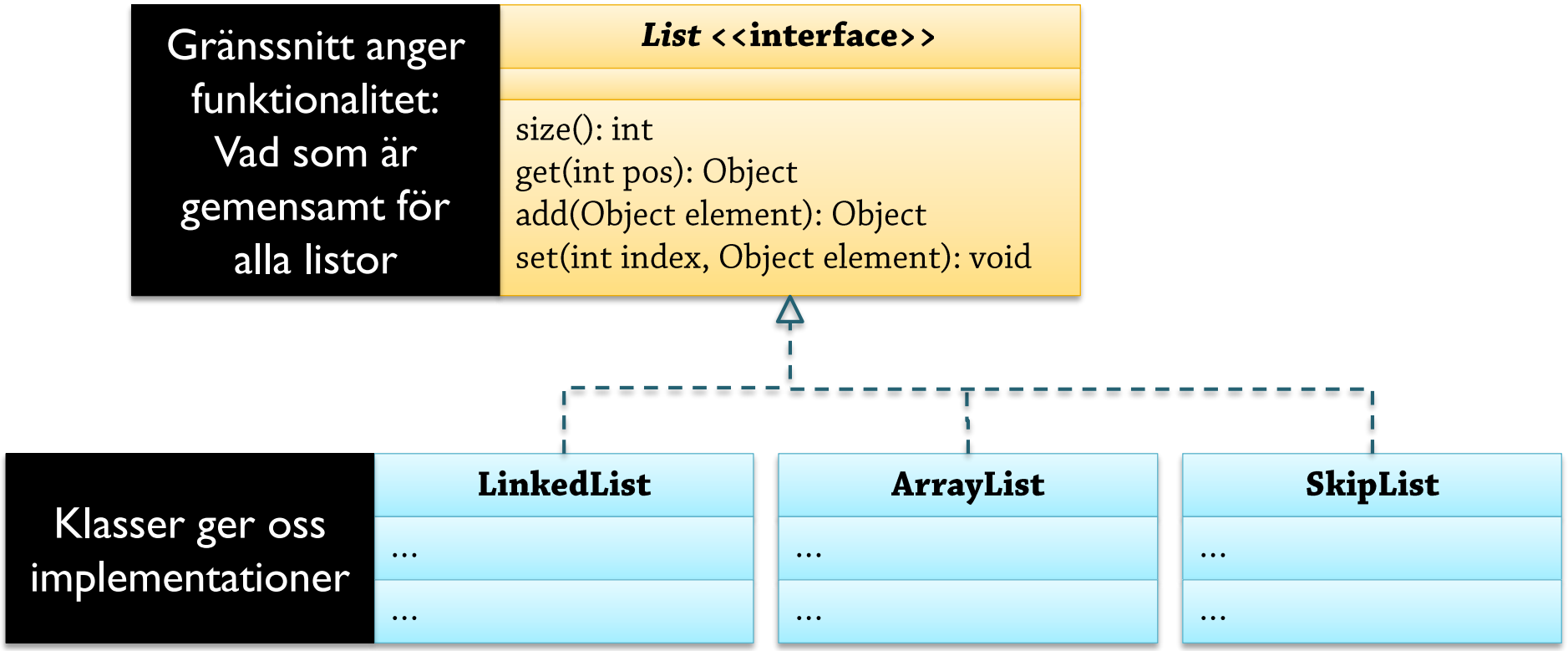
- Fortfarande ”alla x är y”:



Interface-ärvning 1: Hierarkier

jonkv@ida

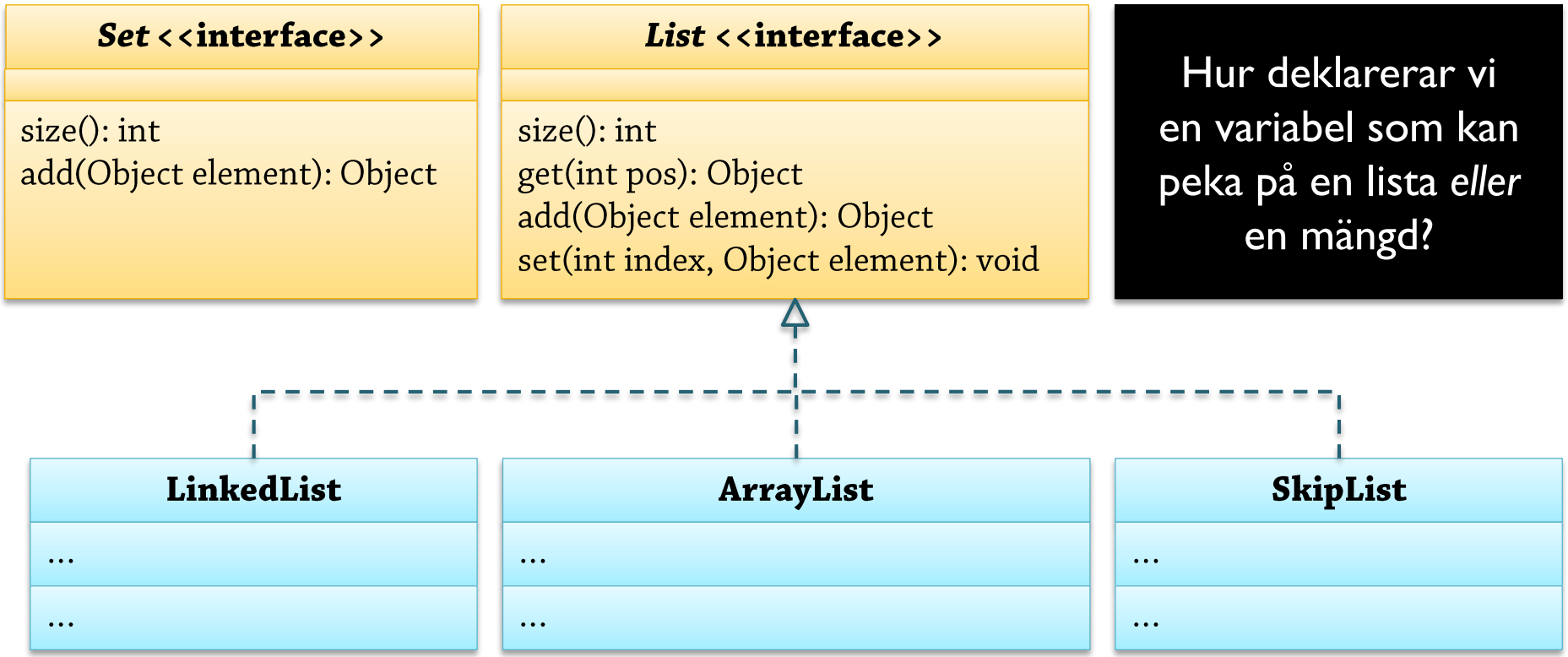
- Vi såg typhierarkier i **två nivåer**



Interface-ärvning 2: Gemensamt

jonkv@ida

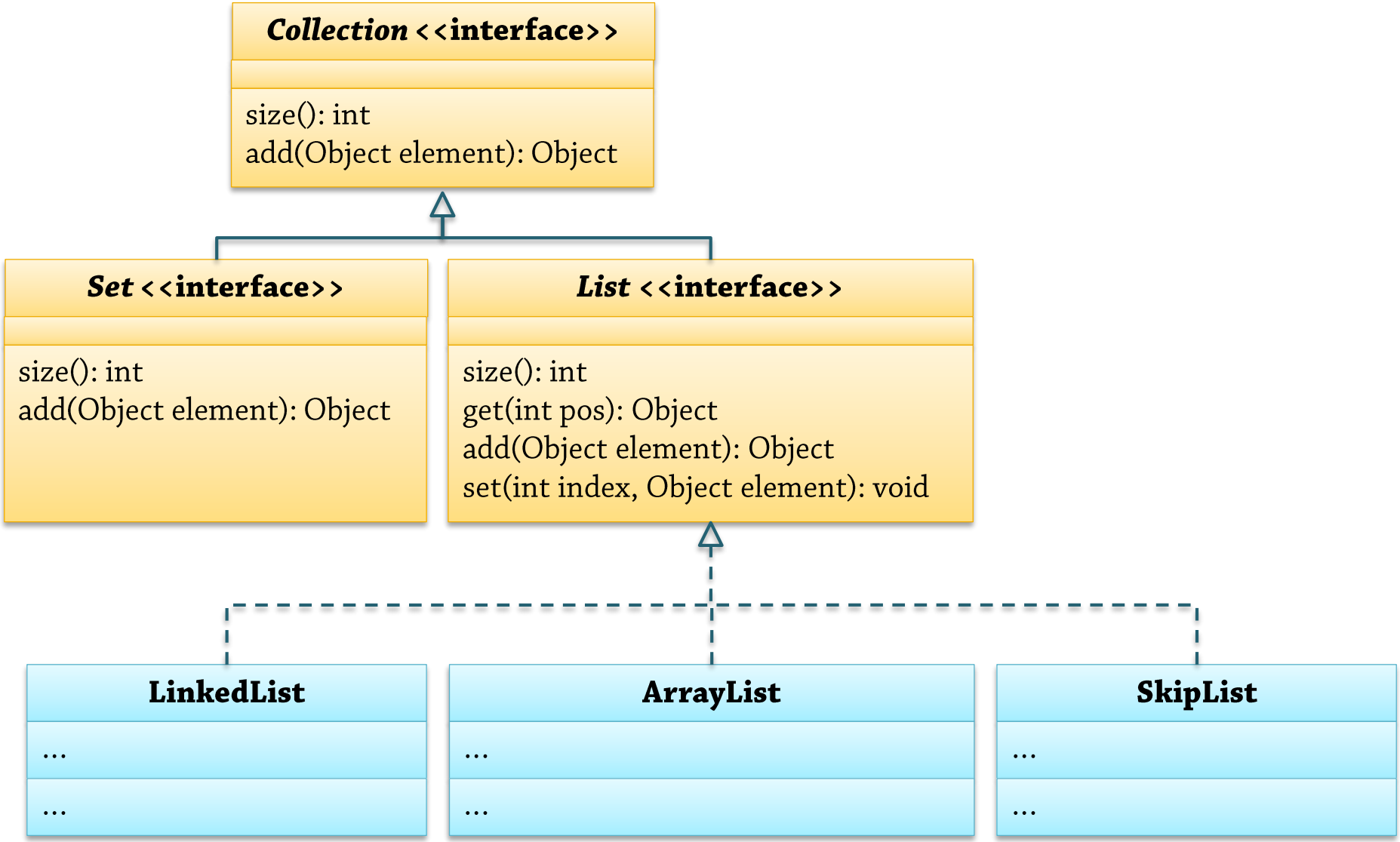
- Men **List** har också något gemensamt med **Set**…



Interface-ärvning 3: En nivå till

jonkv@ida

- Genom ytterligare en nivå!



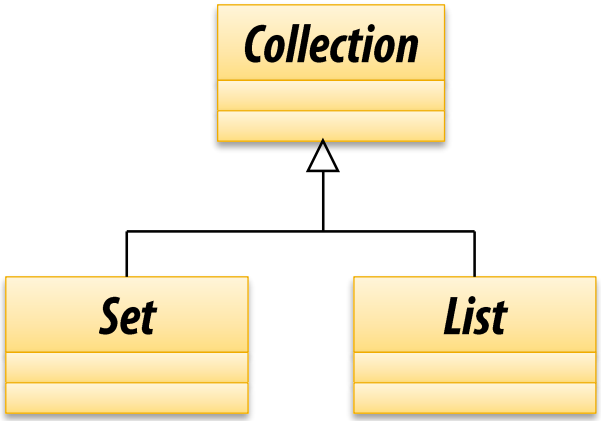
Interface-ärvning 4

- Detta är **interface-ärvning**
  - **public interface** Collection {
    void add(Object obj);
    boolean contains(Object obj);
    }
  - **public interface** List **extends** Collection {
    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

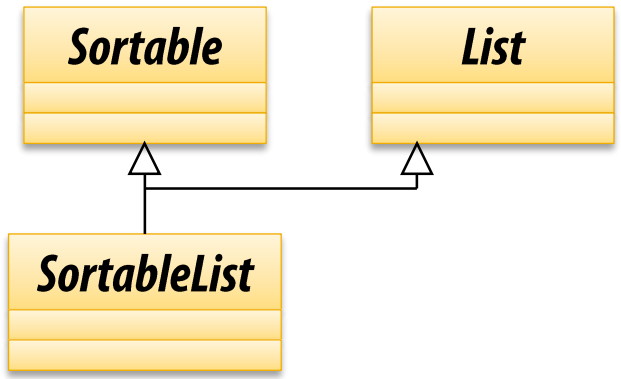


Ärver	kontrakt/löften: Collection lovar att ha en add()-metod, …
Adderar	egna löften: List lovar också …

Interface-ärvning 5: Multipel ärvning

jonkv@ida

- **Multipel** ärvning tillåts för gränssnitt
  - ```
    public interface SortableList extends Sortable, List {
        // Kombinerar supertypernas löften / kontrakt
        // Inget nytt adderat
    }
    ```
  - ```
    public interface NetworkSocket extends DataInput, DataOutput {
        // Kombination av två supertyper
        // Plus: ytterligare en metod krävs
        public NetworkAddress getRemoteHost();
    }
    ```



multipel ärvning = multiple inheritance

UML-klassdiagram: Generalisering

jonkv@ida

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