Typhierarkier del 2

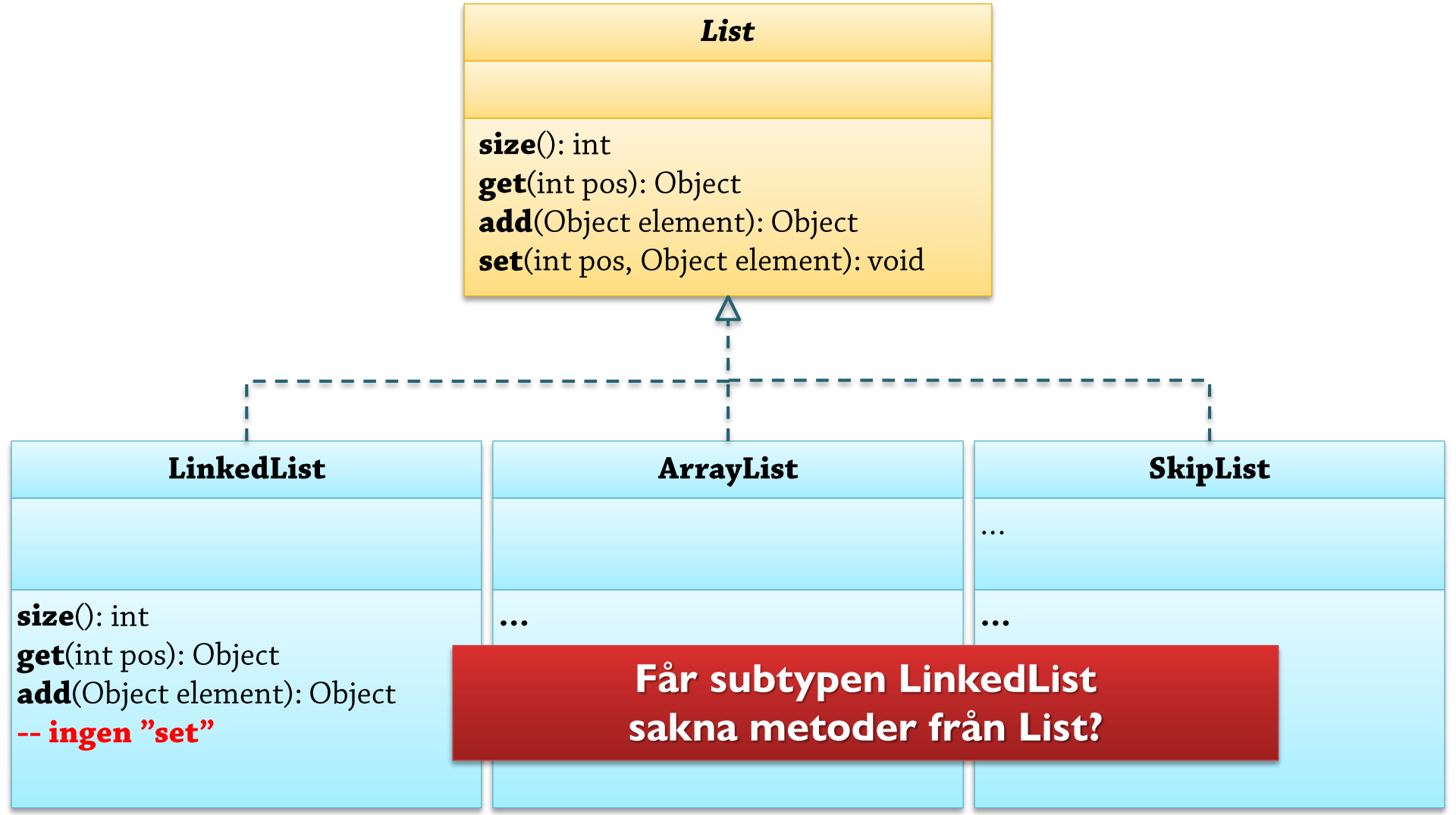
Vad krävs? Hur fungerar det?

Typhierarkier och kontrakt: Liskov Substitution Principle

Hur får en subtyp fungera egentligen?

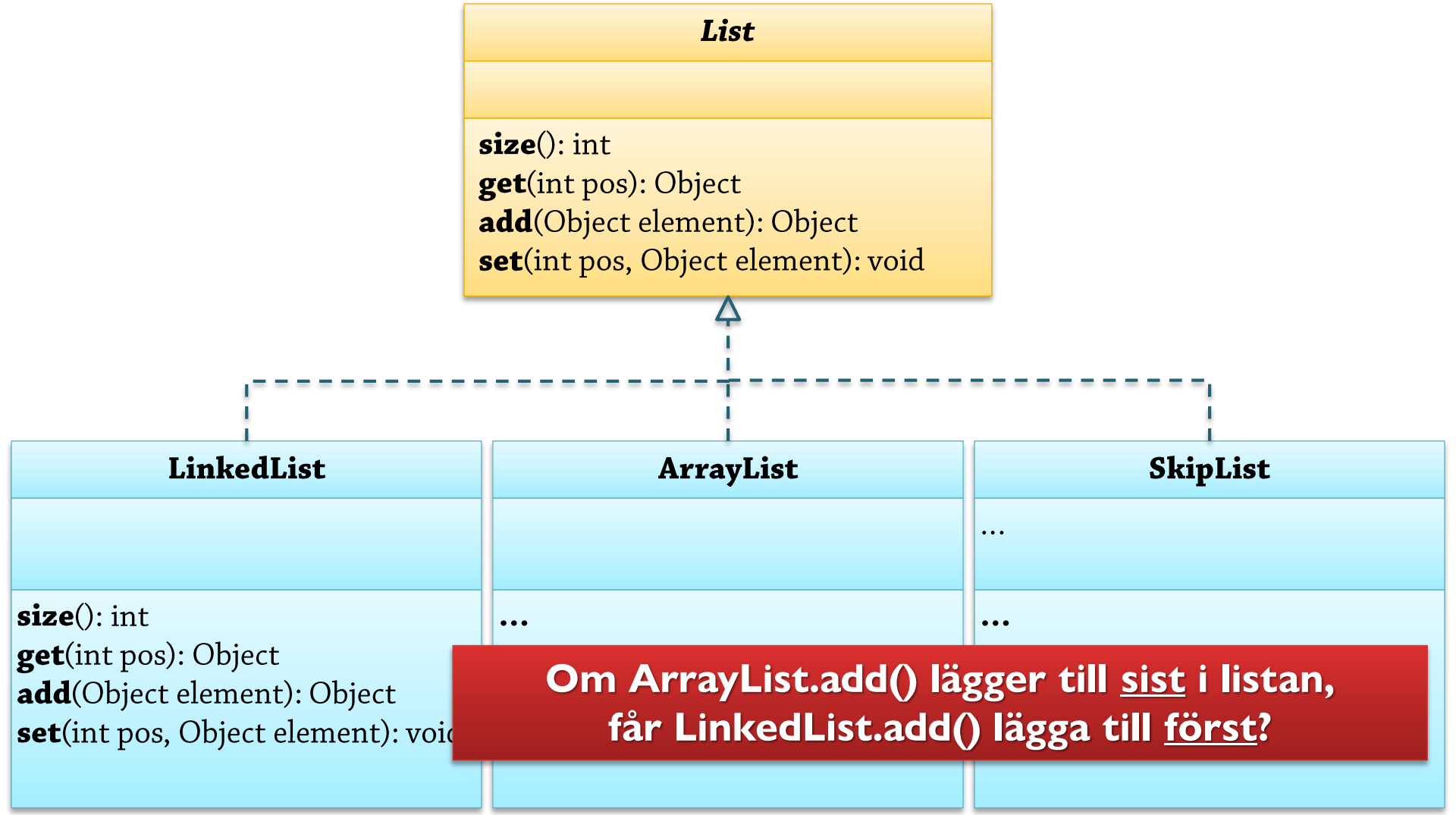
Krav på hierarkier 1

jonkv@ida



Krav på hierarkier 2

jonkv@ida



Krav på hierarkier 3

jonkv@ida

- Även **List** har ett **kontrakt**
  - En ArrayList **är** en List
    ➔ måste uppfylla hela detta kontrakt!
  - (**Alla** fåglar ska ha fjädrar,
    ankor **är** en sorts fåglar
    ➔ alla **ankor** ska ha fjädrar)

- Subtypens ***eget*** kontrakt får:
  - **Lova mer** – ge *starkare* garantier (alla ankor kan dessutom kvacka)
  - **Kräva mindre** – ha *svagare* krav, klara fler sorters input, …

```
public void quicksort(List someList) {
   // Vi programmerar utifrån List:s kontrakt – använder oss bara av dessa garantier
   if (someList.isEmpty()) { … }  // List säger: isEmpty returnerar sant om listan är tom

   // Detta kontrakt uppfylls av alla subtyper: ArrayList, SkipList, LinkedList, …
   // Alltså fungerar metoden för alla subtyper: Skicka in en ArrayList
}
```

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

Överkurs: Kovarianta returtyper!

jonkv@ida

- **Lova mer** – ge *starkare* garantier?

Java låter subklasser stärka löftet om returtyp

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

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

Lovar fortfarande att det är en lista – 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

Hur fungerar typhierarkier tillsammans med manifest typning?

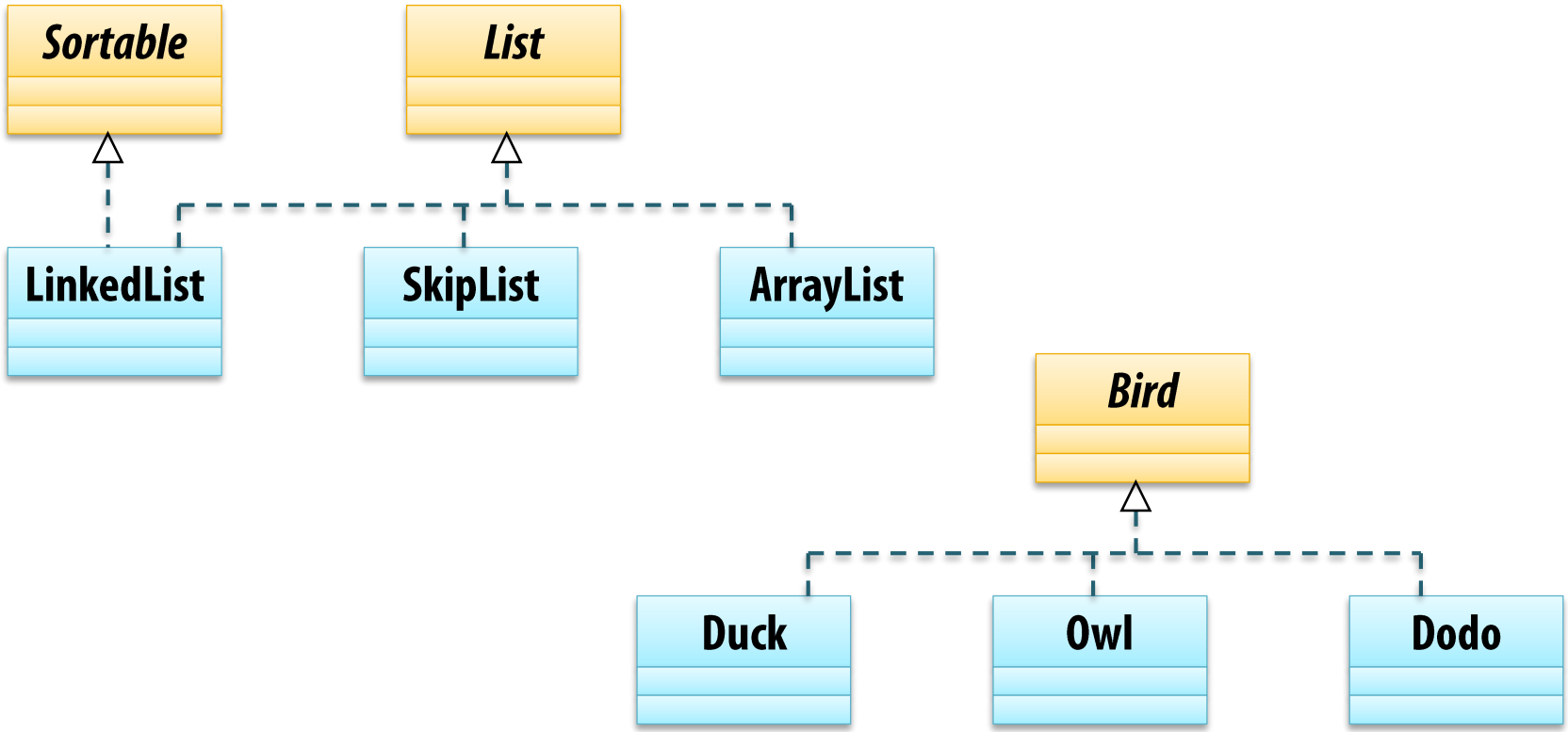
Vilken typ *har* ett objekt?

Eller vilka *typer*…?

Objekt med flera typer

jonkv@ida

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



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…

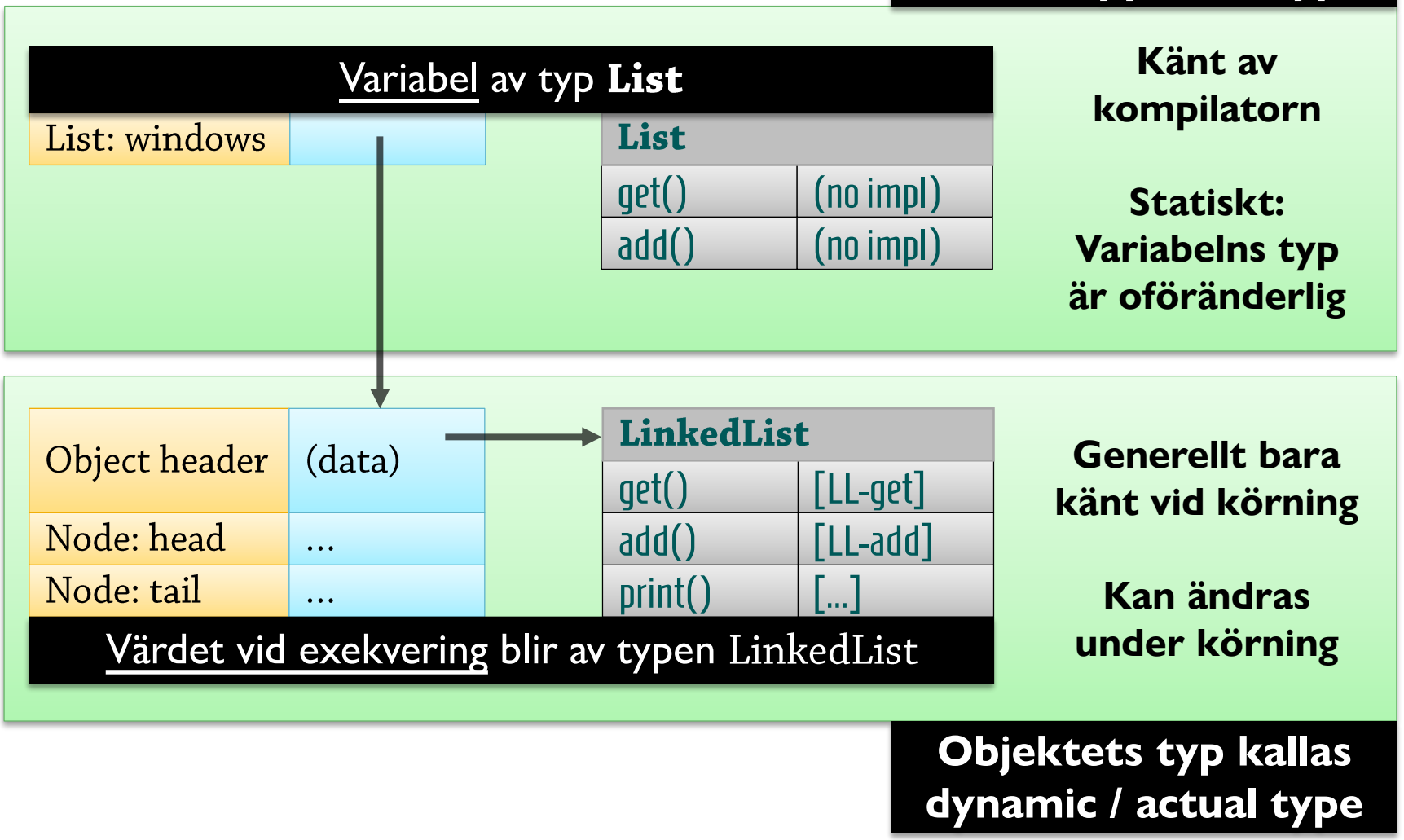
Så kan variabler byta typ?

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 kommer elements att vara?
   }
}
```

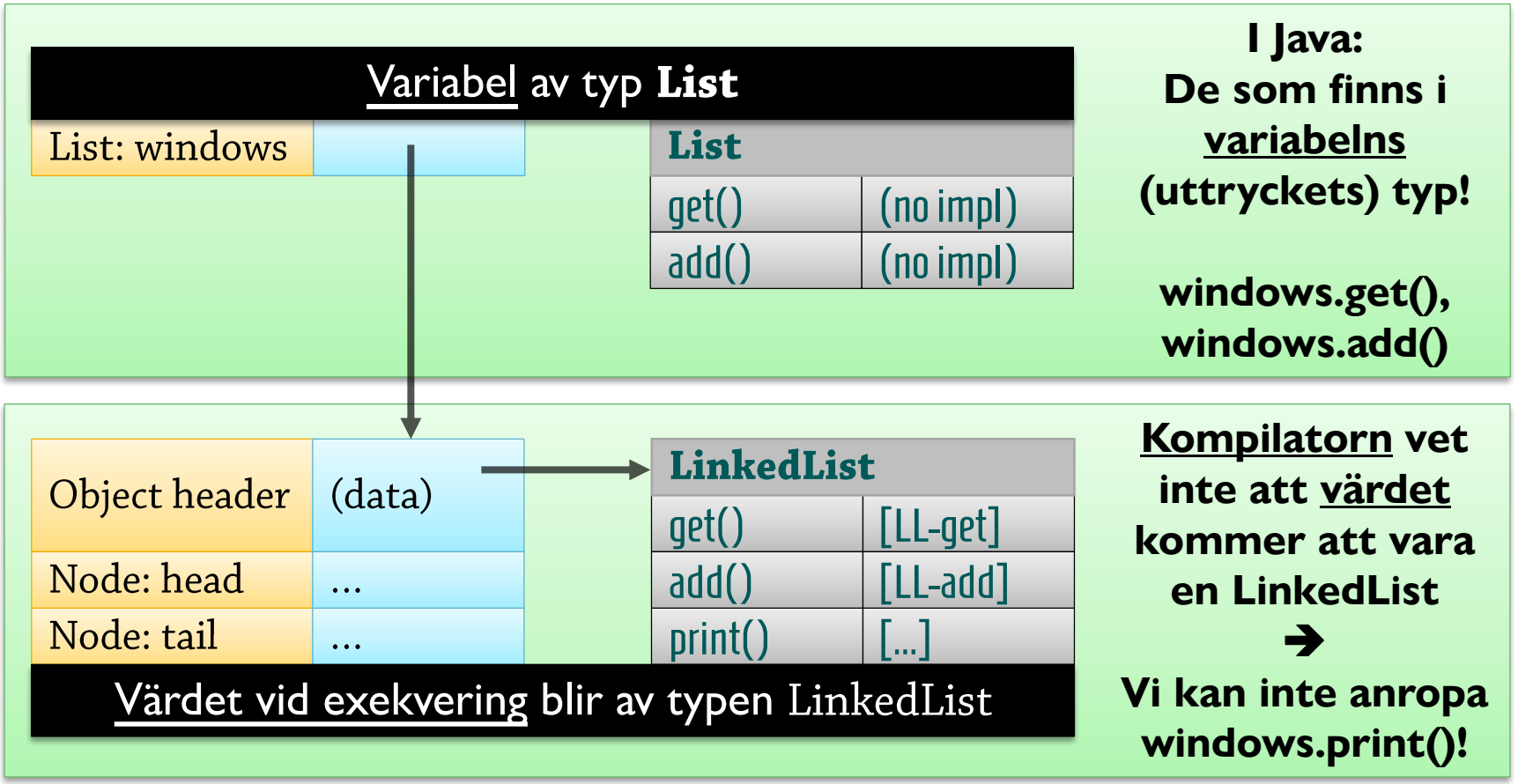
```
public class Randomizer {
   private final Random rnd = new Random();
   public void useList() {
      List list;
      if (rnd.nextBoolean()) {
         list = new ArrayList();
      } else {
         list = new LinkedList();
      }
      list.add(...);
   }
}
```

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**

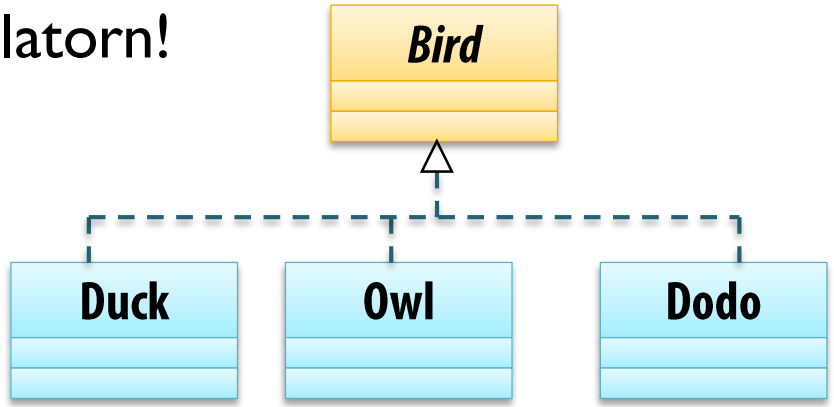
```
public class Application {
   public List      getWindows() { … }
   public SkipList  getFiles() { … }
}
```

```
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?

- 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
      ```



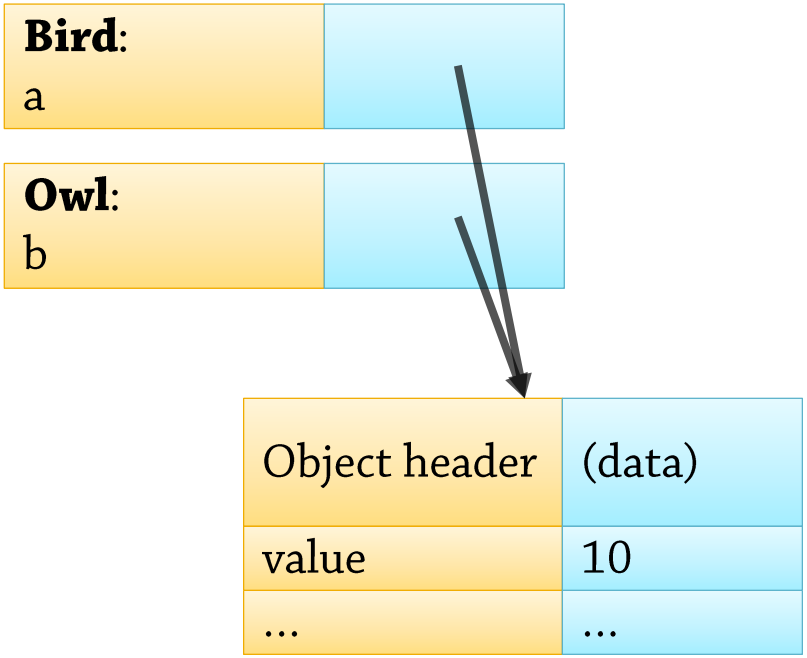
Men vi vet ju vad vi stoppade in där!

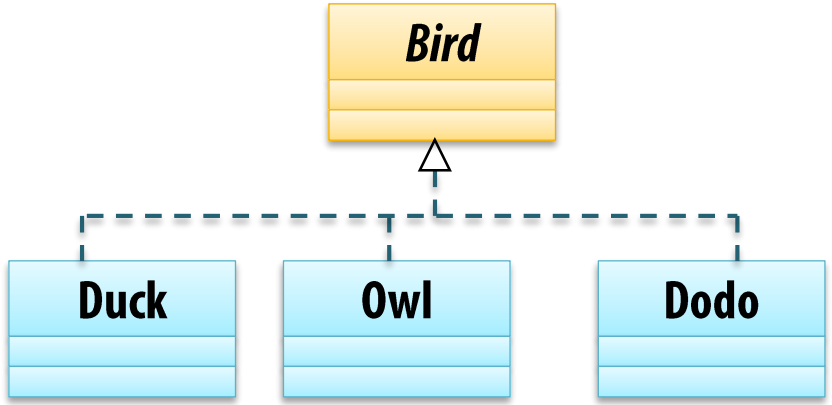
Att veta mer (2)

- 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

Hur fungerar egentligen overriding?
Hur anropas rätt implementation? Vad *är* rätt implementation?

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**:

Om vi inte hade typhierarkier	Med typhierarkier (Java)
▪ Variabelns typ = objektets typ ➔ **Statisk bindning** är möjlig	▪ Variabelns typ != objektets typ ➔ **Dynamisk bindning** krävs
ArrayList myList = …; myList.add(…);	List myList = …; myList.add(…);
Vilken variant av add()? **Måste vara ArrayList:s!** Känt vid kompileringen: Namnet add() kan ***statiskt bindas*** till en funktion av kompilatorn	**Vilken variant av add()?** **ArrayList:s? LinkedList:s?** Okänt vid kompileringen: Namnet add() måste ***dynamiskt bindas*** till en funktion vid körning

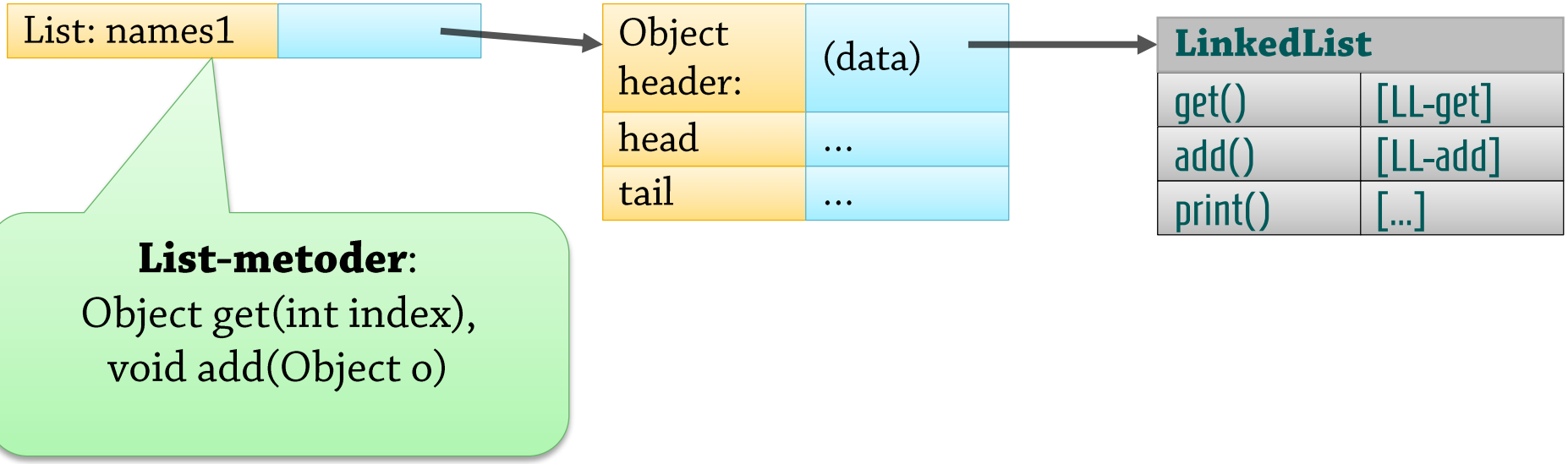
Metodtabeller

jonkv@ida

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

Klassinfo med metodtabell

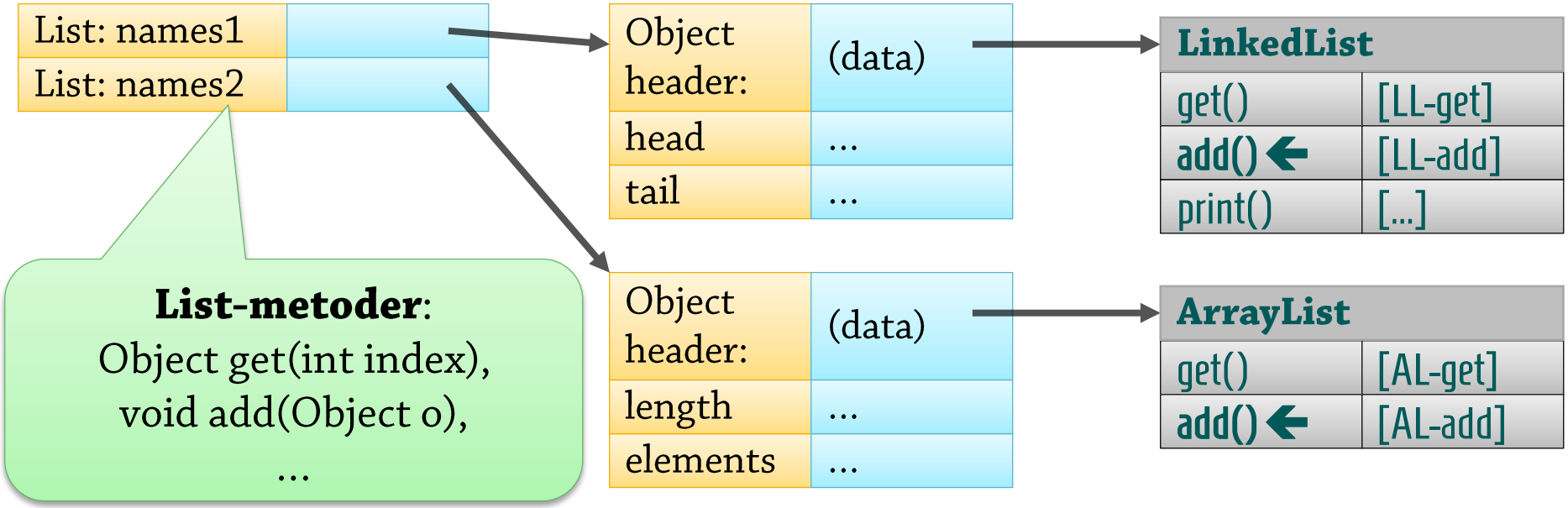
Virtual method table: vtable



Dynamisk bindning

jonkv@ida

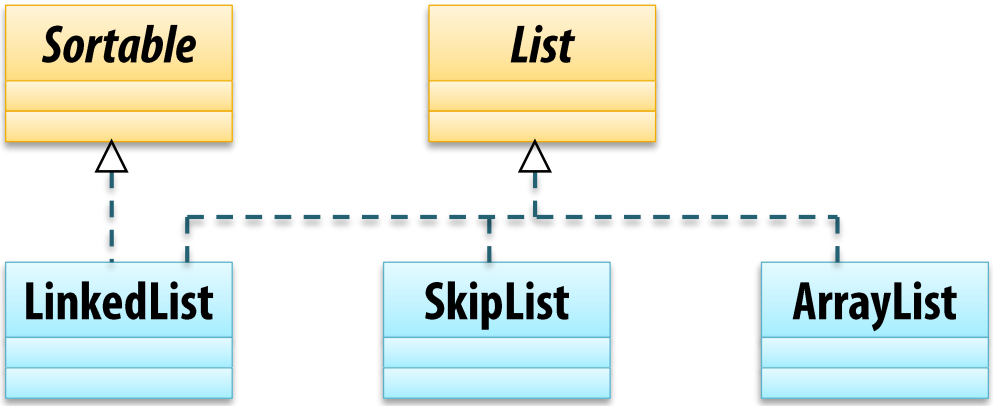
- **Dynamisk bindning** = **sen bindning** = **dynamisk dispatch**
  - *Objektets* typ avgör
    - List names1 = foo.getNames(); // Returnerar en **LinkedList**
      names1.add(n1); // LinkedList-implementationen av add
    - List names2 = bar.getNames(); // Returnerar en **ArrayList**
      names2.add(n1); // 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

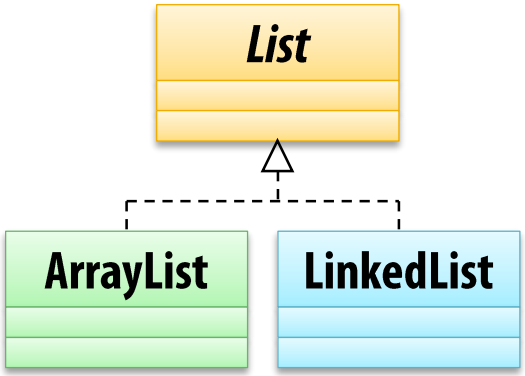
jonkv@ida

- 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**
      ```

Abstrakta klasser i Java

Motivation

- Klasshierarkin ska grundas i *begrepp, klassificering*
  - Naturligt att prata om *listor* i allmänhet ➔ ge flera *implementationer*



”X är en speciell sorts Y, som…”

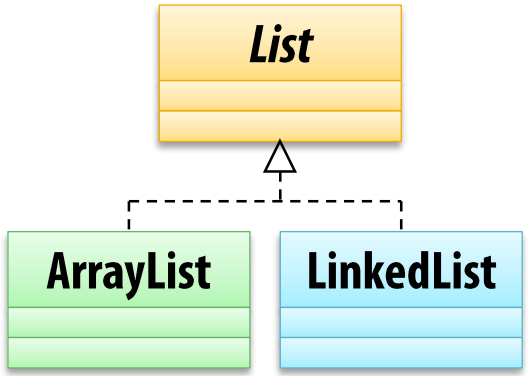
  - Naturligt att prata om *grafiska komponenter* i allmänhet
    - ➔ Vissa komponenter är *knappar*
    - ➔ Vissa komponenter är *textfält*
    - ➔ …

Men ibland vill vi <u>också</u> undvika redundant kod!

(Vanligt ”problem” i projekt…)

Motivation 2: Duplicerad kod


```
public class ArrayList implements List {
   Object[] elements = new Object[42];
   int length = 0;
   int size() { return length; }
   Object get(int index) { return elements[index]; }
   void add(Object element) { … }
   void set(int index, Object element) { …
   int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```




```
public class LinkedList implements List {
   Node head = null, tail = null;
   int length = 0;
   int size() { return length; }
   Object get(int index) { … }
   void add(Object element) { … }
   void set(int index, Object element) { … }
   int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Massor av duplicerad kod…
Hur kan vi använda *samma* kod för båda klasser?

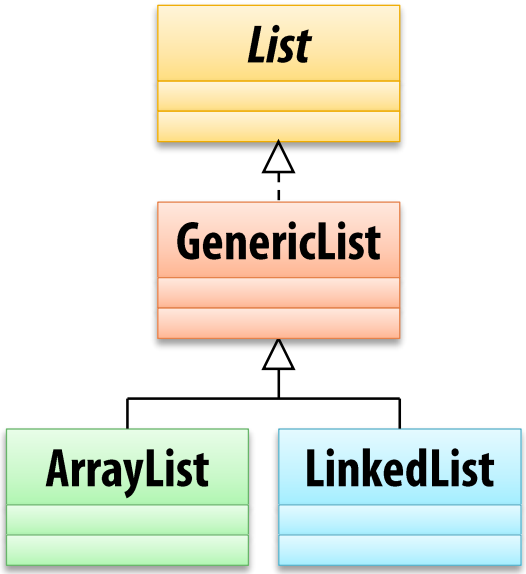
Inte flytta till **List**: Gränssnitt ska ange *krav*, inte *implementation*

Försök 1: Återanvända med konkret klass?

jonkv@ida

```
public class GenericList implements List {
   int length = 0;
   int size() { return length; }
   int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Bra försök,
men GenericList skulle behöva
implementera add(), get(), set()!




```
public class ArrayList extends GenericList {
   Object[] elements = new Object[42];
   Object get(final int index) { return elements[index]; }
   void add(final Object element) { … }
   void set(int index, Object element) { … }
}
```

```
public class LinkedList extends GenericList {
   Node head = null, tail = null;
   Object get(final int index) { … }
   void add(final Object element) { … }
   void set(int index, Object element) { … }
}
```

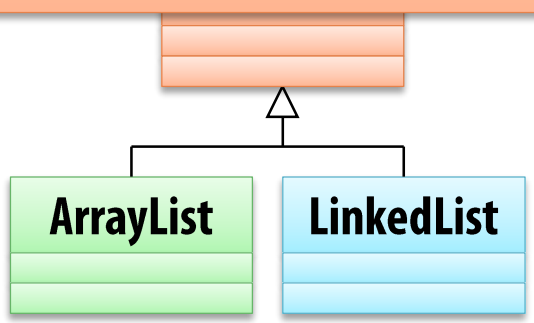
Försök 2: Dummy-implementation?

jonkv@ida

```
public class GenericList implements List {
   int length = 0;
   int size() { return length; }
   int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
   Object get(final int index) {
      return null;
   }
   void add(final Object element) {}
   …
}
```

Bra försök, men GenericList skulle inte uppfylla kraven på List: get ska *returnera rätt element*!

```
ends GenericList {
s = new Object[42];
int index) { return elements[index]; }
bject element) { … }
x, Object element) { … }
```




```
xtends GenericList {
   Node head = null, tail = null;
   Object get(final int index) { … }
   void add(final Object element) { … }
   void set(int index, Object element) { … }
}
```

Abstract 1: Intro

- Vi vill ha en ***klass***, där:
  - *Vissa* metoder ska implementeras
  - *Andra* metoder ska *krävas*, men inte implementeras
- Klassen blir **ofullständig**, ungefär som gränssnitt
  - Partiellt implementerad
- Kallas ***abstrakt***
  - *existing in thought or as an idea but not having a physical or concrete existence*

Mest för att undvika redundant kod!

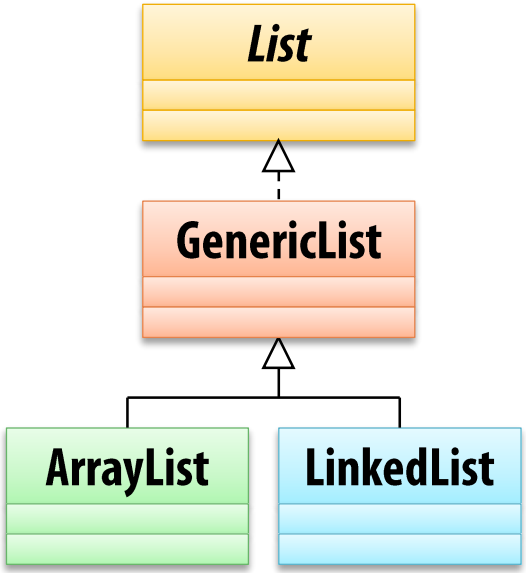
Abstract 2: För återanvändning!

jonkv@ida

```
public abstract class GenericList implements List {
   protected int length = 0;
   public int size() { return length; }
   public int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Ärver löften från List

Saknar ändå get(), add, set()




```
public class ArrayList extends GenericList {
   public Object[] elements = new Object[42];
   public Object get(final int index) { … }
   public void add(final Object element) { … }
   public void set(int index, Object element) { …
}
```

```
public class LinkedList extends GenericList {
   public Node head = null, tail = null;
   public Object get(final int index) { … }
   public void add(final Object element) { … }
   public void set(int index, Object element) { …
}
```

Ärver löften/kod från GenericList

Implementerar det som saknas

Abstract 3: Gå hela vägen!

jonkv@ida

```
abstract class GenericList implements List {
   int indexOf(Object o) {
      for (int i = 0; i < size(); i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Relaterat problem i projekt:
Ärvningen finns där,
men **medlemmar** repeteras ändå!

```
class ArrayList extends GenericList {
   int length = 0;
   int size() { return length; }
   Object[] elements = new Object[42];
   …
}
```

Flytta till superklassen!

```
class LinkedList extends GenericList {
   int length = 0;
   int size() { return length; }
   Node head = null, tail = null;
   …
}
```

Abstract 4: Varning för upprepning

jonkv@ida

```
abstract class GenericList implements List {
  int length = 0;
  int size() { return length; }
  int indexOf(Object o) {
    for (int i = 0; i < size(); i++)
      if (get(i).equals(o)) return i;
    return –1;
  }
}
```

Relaterat problem i projekt:
Ärvningen finns där,
men **fält** repeteras ändå!

```
class ArrayList extends GenericList {
  int length = 0;   // Fält har inte "override"
  Object[] elements = new Object[42];
  …
}
```

Ger **skuggning** av fält:
Varje ArrayList har TVÅ fält med namnet length

Vilket som används beror på **pekartypen**

Slösar minne
Svårt att förstå!

```
class LinkedList extends GenericList {
  int length = 0;
  Node head = null, tail = null;
  …
}
```

Abstract 5: Instansiering

jonkv@ida

```
public abstract class GenericList implements List
{
   protected int length = 0;
   public int size() { return length; }
   public int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Kan inte instansieras
med ”**new** GenericList()”

Kan bara skapa *konkreta* objekt

```
public class LinkedList extends GenericList {
   public Node head = null, tail = null;
   public Object get(final int index) { … }
   public void add(final Object element) { … }
   public void set(int index, Object element) { … }
}
```

Men vi kan skapa
”**new** LinkedList()”,
vilket *ger* en sorts
GenericList!

Abstract 6: Konstruktorer

jonkv@ida

- Även en abstrakt klass ska **initialisera** sina objekt

```
abstract class GenericList implements List {
   protected int length;
   protected GenericList() {
      this.length = 0;
   }
   public int size() { return length; }
   public int indexOf(Object obj) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(obj)) return i;
      return –1;
   }
}
```

Det finns fält
(som ärvs av ArrayList, …)

Initialiseras i en konstruktor:
Klassen bestämmer
över sina objekt

```
public class ArrayList extends GenericList {
   private Object[] elements;

   public ArrayList() {
      super();
      this.elements = new Object[42];
   }
}
```

new ArrayList() ➔
super() = GenericList()

Abstract 7: Protected

jonkv@ida

```
abstract class GenericList implements List {
   protected int length;
   protected GenericList() {
      this.length = 0;
   }
   public int size() { return length; }
   public int indexOf(Object obj) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(obj)) return i;
      return –1;
   }
}
```

Konstruktorer i **abstrakta** klasser bör vara *protected*:
Tillgängliga i subklasser
(enda som kan anropa dem!)

Abstract 8: Begrepp?

jonkv@ida

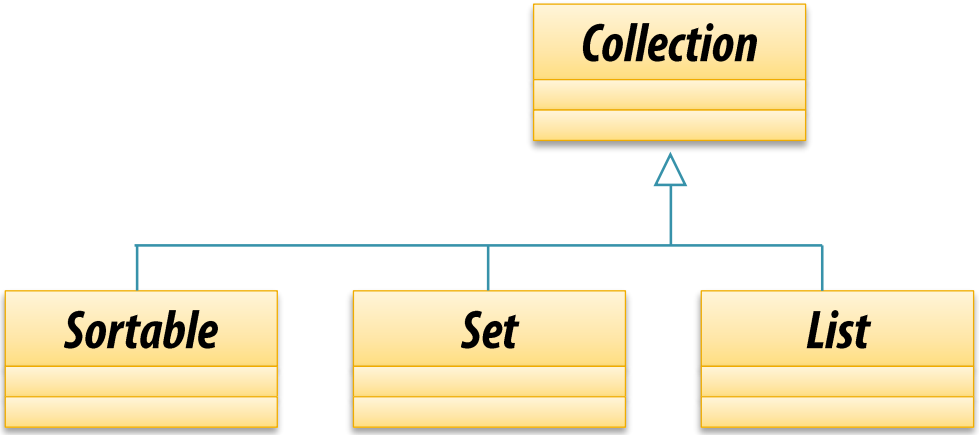
- Abstrakta klasser:
  Kan vara ett **undantag** från ”begreppsmodelleringen”
  - Inget behov att **prata om** GenericList:
    Då räcker det med List
  - Men vi vill *samla gemensam kod*

```
abstract class GenericList implements List {
   protected int length;
   protected GenericList() {
      this.length = 0;
   }
   public int size() { return length; }
   public int indexOf(Object obj) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(obj)) return i;
      return –1;
   }
}
```

Sammanfattning av typhierarkier

Sammanfattning: Klasshierarkier

jonkv@ida

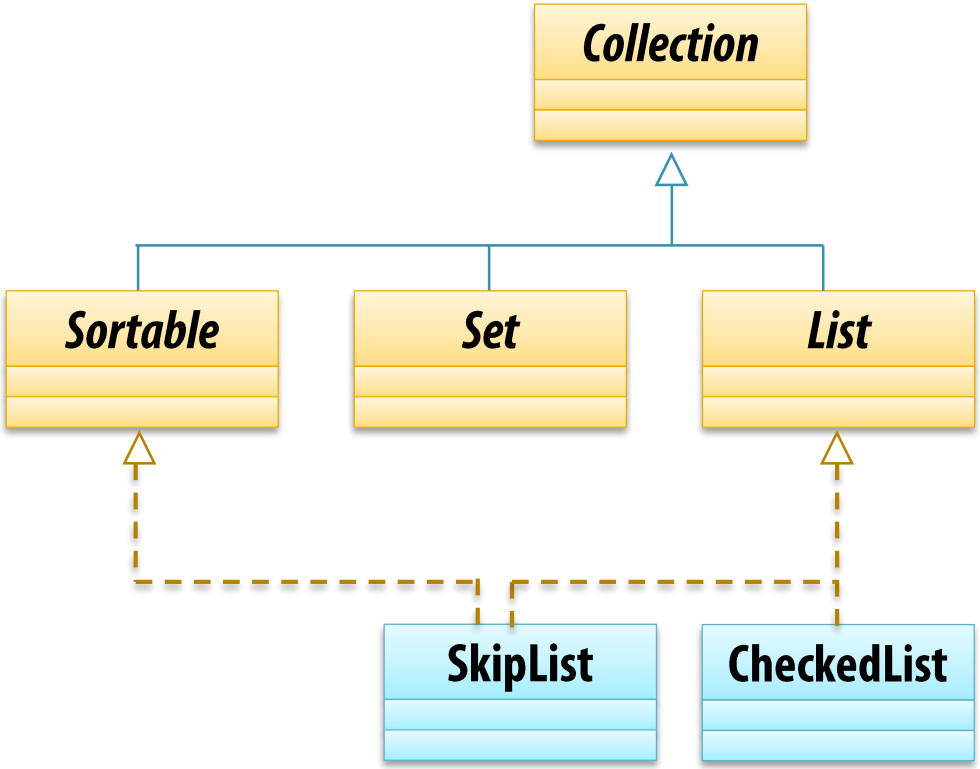


2. Ett gränssnitt kan
ärva från och ***utöka***
(löftena som gavs av)
ett annat gränssnitt

1. Ofta (men inte alltid)
har vi en mängd **gränssnitt**
som definierar ***funktionalitet***
och ger ***löften (kontrakt)***,
bland annat genom att ange
abstrakta metoder

Sammanfattning: Klasshierarkier (2)

jonkv@ida



3. En metod kan ta "godtycklig Collection" som parameter

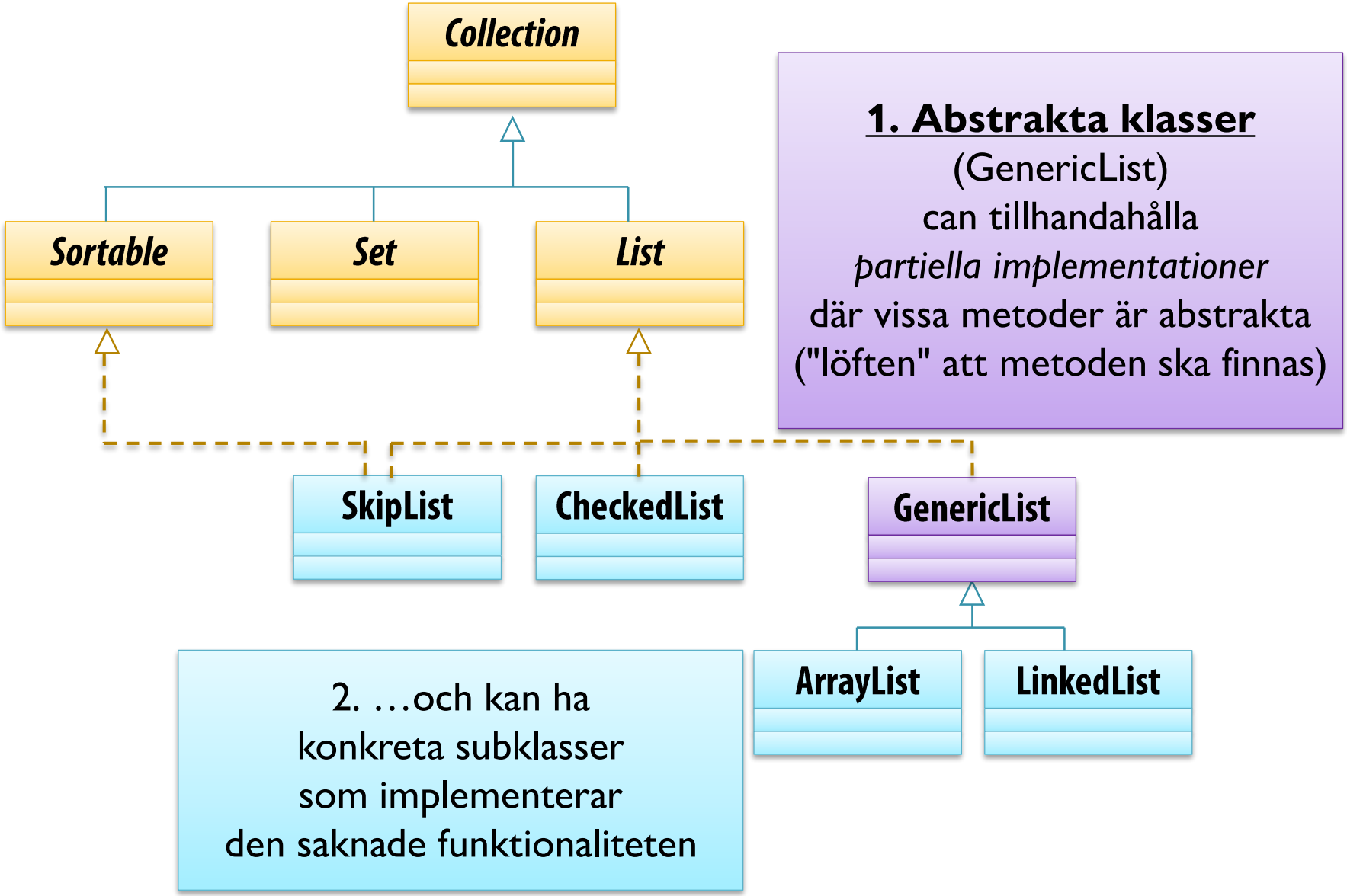
2. En metod kan ta ”godtycklig List” som parameter. Kan vara en SkipList eller CheckedList – spelar ingen roll!

1. Klasser kan *implementera* gränssnitt
➔ måste uppfylla deras kontrakt

4. En klass kan implementera flera gränssnitt.
Ingen tvetydighet, eftersom ingen kod ärvs.

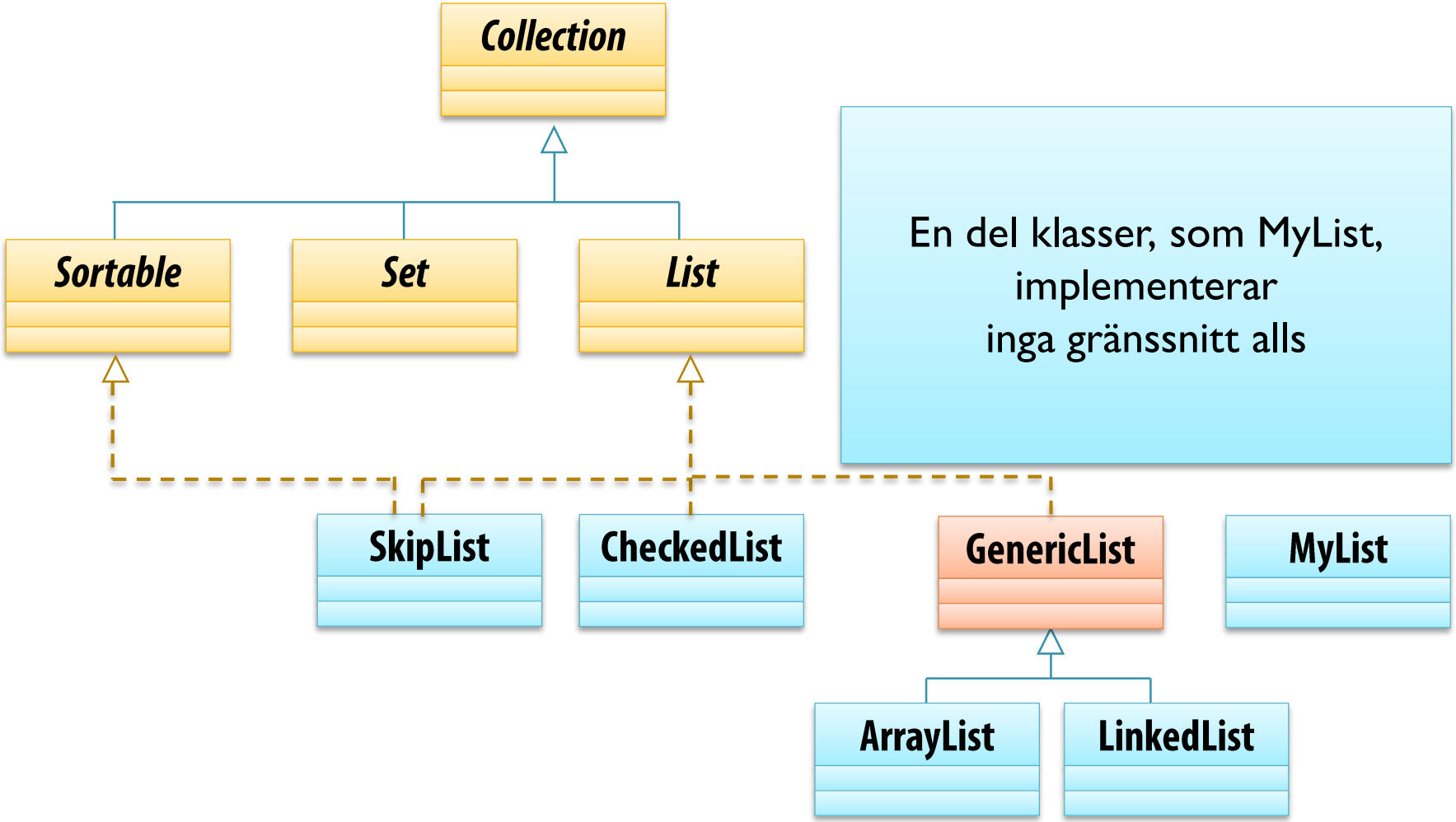
Sammanfattning: Klasshierarkier (3)

jonkv@ida



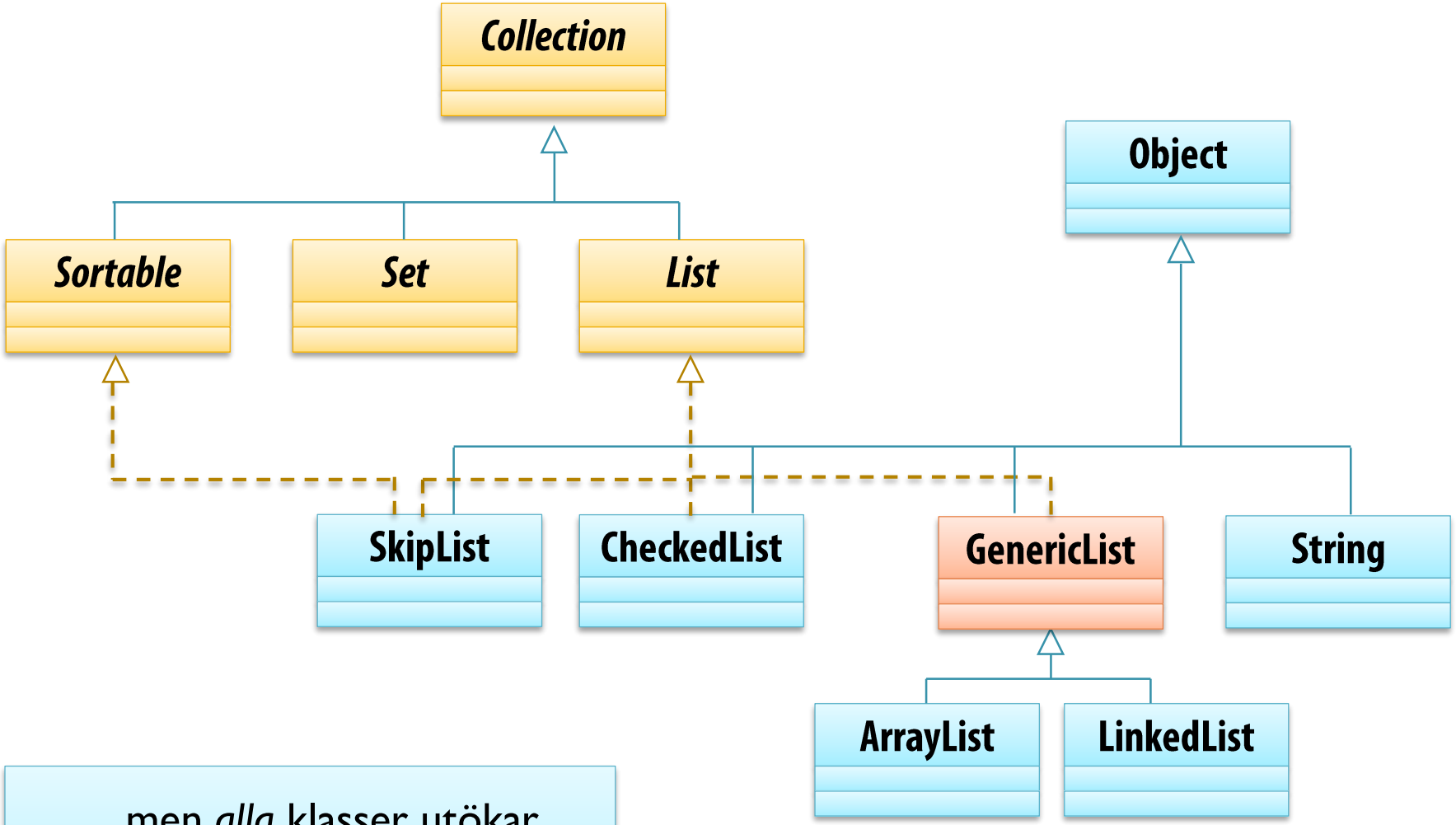
Sammanfattning: Klasshierarkier (4)

jonkv@ida



Sammanfattning: Klasshierarkier (5)

jonkv@ida



…men *alla* klasser utökar
(direkt eller indirekt)
java.lang.Object!

Egen typkontroll i Java (RTTI, Run-Time Type Identification)

Varning: Ska användas mycket sparsamt!

Typkontroll: getClass() och instanceof

jonkv@ida

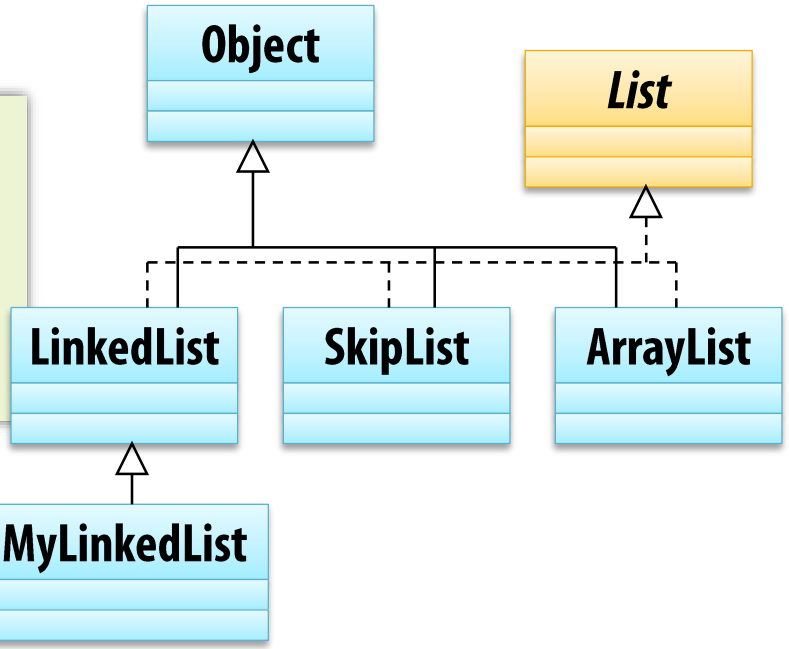
- Vilken typ har ett visst objekt?
  - **Exakt** typ: object.getClass() == TypeName**.class**

```
if (obj.getClass() == LinkedList.class) {
    // Exakt typkontroll:
    // obj pekar på ett objekt som skapats med "new LinkedList()"
}
```

  - **Subtyp**: object **instanceof** TypeName

```
if (obj instanceof LinkedList) {
    // Subtypskontroll:
    // obj pekar på ett objekt av typ LinkedList
    // eller en subklass som MyLinkedList
}
```

  - *Allt* är **instanceof** Object!



Typkontroll: Operatorn instanceof (2)

jonkv@ida

- **Kan** användas för att göra olika saker beroende på typ…
  -

```
void checkCollisions() {
    for (Thing t : thingsOnScreen) {
        if (player.intersects(t)) {
            // Vad var det vi kolliderade med egentligen?
            if (t instanceof Wall) {
                // Få spelaren att studsa tillbaka
                // …
            } else if (t instanceof Bullet) {
                // Få spelaren att förlora hälsa
                // …
            } else if (t instanceof Powerup) {
                player.addPU((Powerup)t);
                // …
            } else if … { … }
        }
    }
```

”För alla Thing-objekt t i samlingen thingsOnScreen”

Typkontroll: Operatorn instanceof (3)

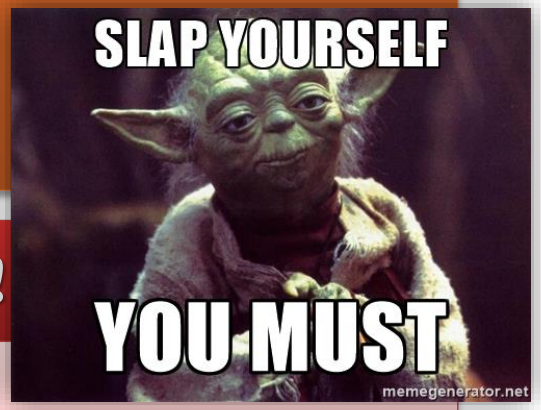
jonkv@ida

- **Kan** användas för att göra olika saker beroende på typ…

```
void checkCollisions() {
    for (Thing t : thingsOnScreen) {
        if (player.intersects(t)) {
            if (t instanceof Wall) {
                // Få spelaren att studsa tillbaka
            } else if (t instanceof Bullet) {
                // Få spelaren att förlora hälsa
            } else if (t instanceof Powerup) {
                player.addPU((Powerup)t);
            } else if …
        }
```

Göra olika beroende på typ…?

Det är ju detta subtypspolymorfism är till för!



UNDVIK! Polymorfism är oftast bättre!

Från *Effective C++*, av Scott Meyers :

"Anytime you find yourself writing code of the form 'if the object is of type T1, then do something, but if it's of type T2, then do something else', slap yourself. "

Vad är problemet?

jonkv@ida

```
void checkCollisions() {
   for (Thing t : thingsOnScreen) {
      if (player.intersects(t)) {
         if (t instanceof Wall) {
            // Make player bounce back
         } else if (t instanceof Bullet) {
            // Make player lose health
         } else if (t instanceof Powerup) {
            player.addPU((Powerup)t);
         } else if ...
      }
   }
}
```

Centraliserat: Lägger man till en ny sorts Thing, måste denna metod *också* ändras!

Bräckligt, inte modulärt!

***Single Responsibility Principle*:**
Kollisionshanteraren borde inte bestämma *hur* alla saker på skärmen interagerar med spelaren!

Varför är polymorfism (oftast) bättre?

jonkv@ida

```
void checkCollisions() {
   for (Thing t : thingsOnScreen) {
      if (player.intersects(t)) {
         t.handleCollisionWith(player);
      }
   }
}
```

1. Be saken hantera detta

4. Polymorfism och dynamisk bindning ➔ rätt implementation väljs!

```
interface Thing {
   void handleCollisionWith(Player p);
}

class Bullet implements Thing {
   void handleCollisionWith(Player p) {
      // Make player lose health
   }
}

class Powerup implements Thing {
   void handleCollisionWith(Player p) {
      p.addPowerUp(this);
} }
```

2. Thing lovar att alla Thing-klasser kan hantera kollision

3. Alla Thing-klasser har sin egen implementation, bestämmer sitt beteende

Icke-lösningar på problemet

jonkv@ida

- Ingen instanceof, men samma problem

```
void checkCollisions() {
   for (Thing t : thingsOnScreen) {
      if (player.intersects(t)) {
         if (t.isWall()) {
            // Make player bounce back
         } else if (t.isBullet()) {
            // Make player lose health
         } else if (t.isPowerup()) {
            player.addPU((Powerup)t);
         } else if ...
      }
   }
}
```

```
void checkCollisions() {
   for (Thing t : thingsOnScreen) {
      if (player.intersects(t)) {
         if (t.getType() == WALL) {
            // Make player bounce back
         } else if (t.getType() == BULLET) {
            // Make player lose health
         } else if (t.getType() == POWERUP) {
            player.addPU((Powerup)t);
         } else if ...
      }
   }
}
```

Problemet är typkontrollen, inte instanceof

Be objektet → använd polymorfism