TDDD78
Objektorientering i Java, del 3

Ärvning av *implementation*, inklusive *abstrakta klasser*

Hur ska vi ärva?

När ska vi ärva?

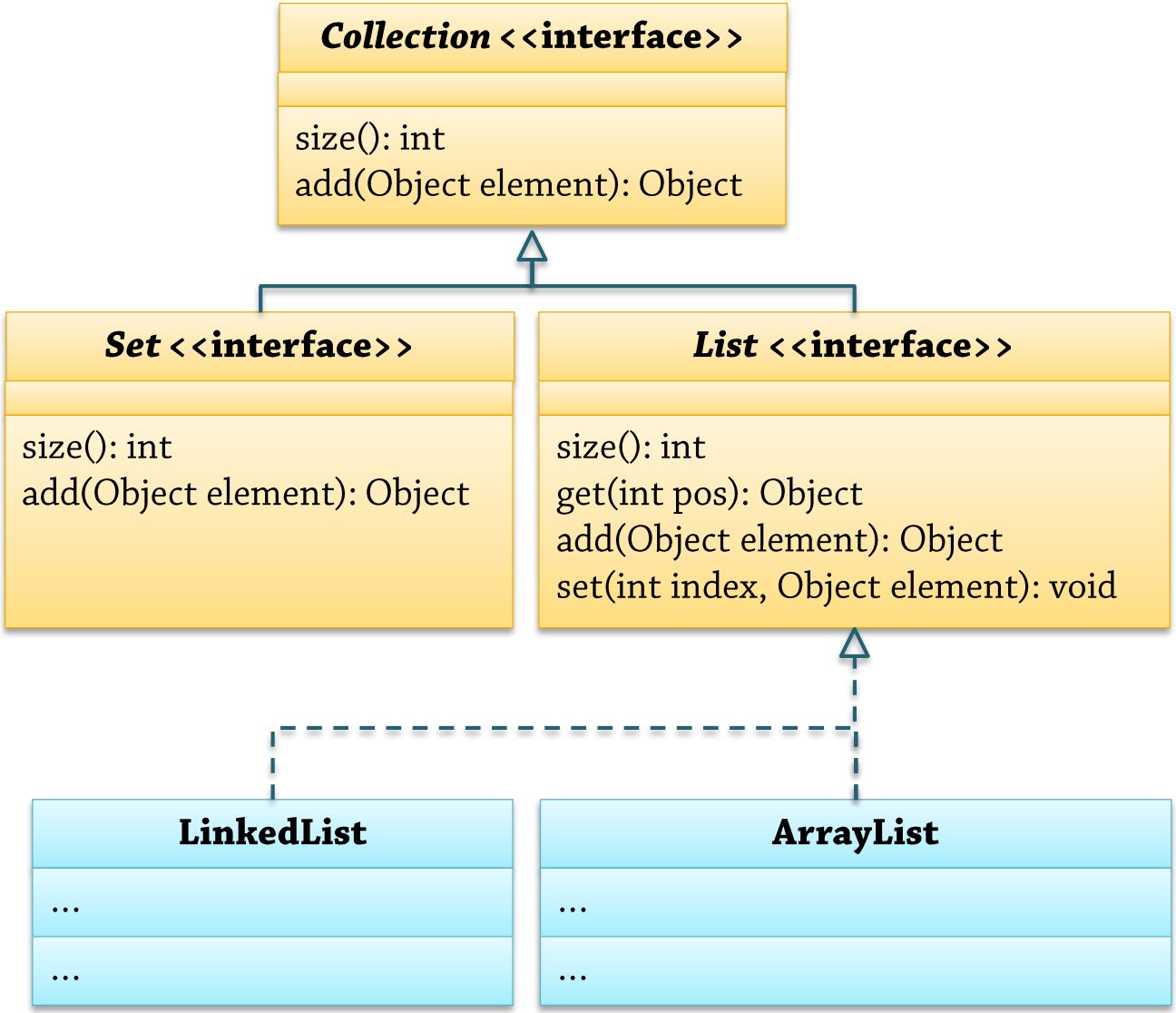
jonas.kvarnstrom@liu.se – 2017

Ärvning av implementation

Implementationsarv 1: Klassnivåer

jonkv@ida

- Nu kan vi ha godtyckligt antal nivåer…



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

Implementationsarv 2: Nya medlemmar

jonkv@ida

- 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

jonkv@ida

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

   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 x, y, r
Adderar outline, fill (Color-objekt)

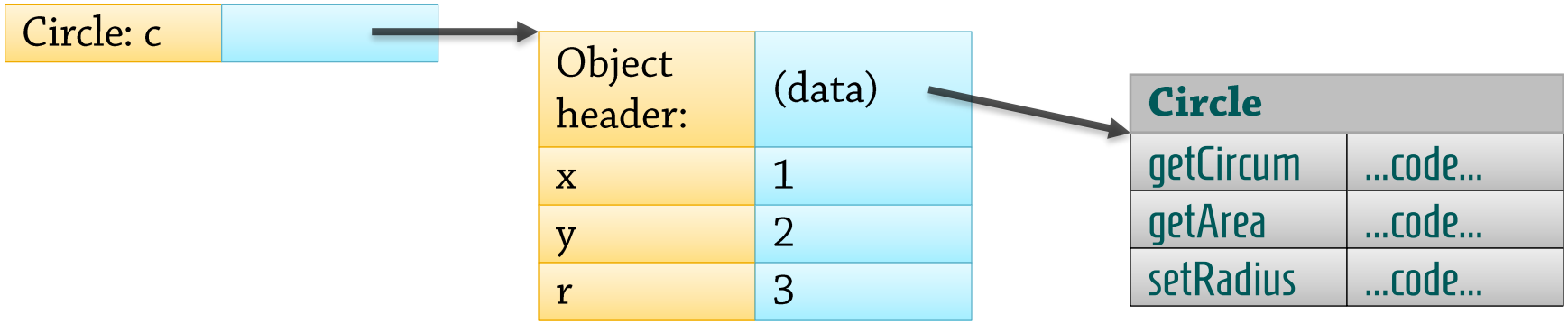
Ärver getRadius()
Adderar draw()

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

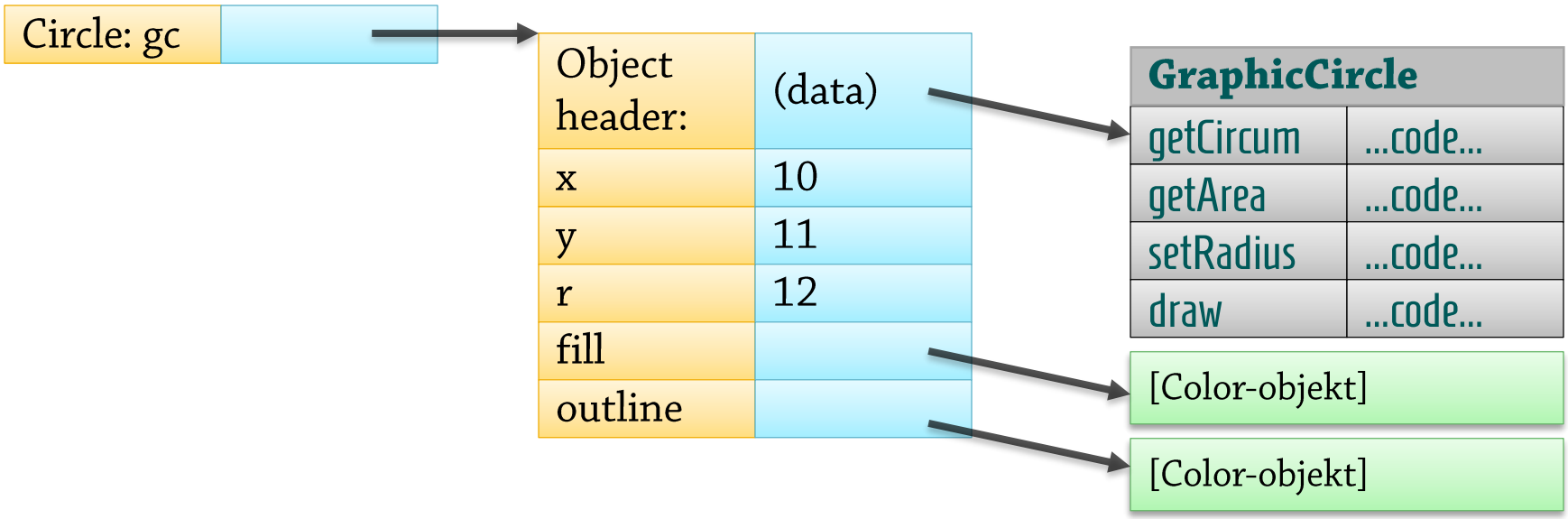
Implementationsarv 4: Visualisering

jonkv@ida

- **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);



Implementationsarv 5: Overriding

jonkv@ida

- 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 (!super.isValid()) 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**

Implementationsarv 6: super()

jonkv@ida

- 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 (!super.isValid()) return false;
    if (outline == null || fill == null) return false;
    return true;
  }
}
```

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

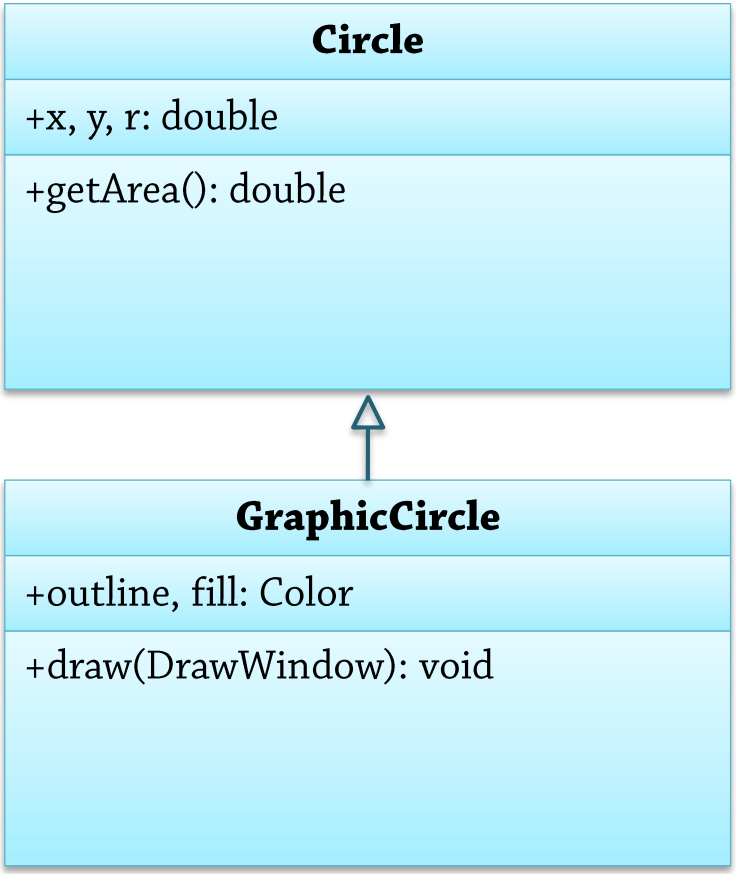
2. Om vi istället skrev **här**:
   if (r < 0) return false; …

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

UML-klassdiagram: Generalisering

jonkv@ida

- 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);
   }
}
```

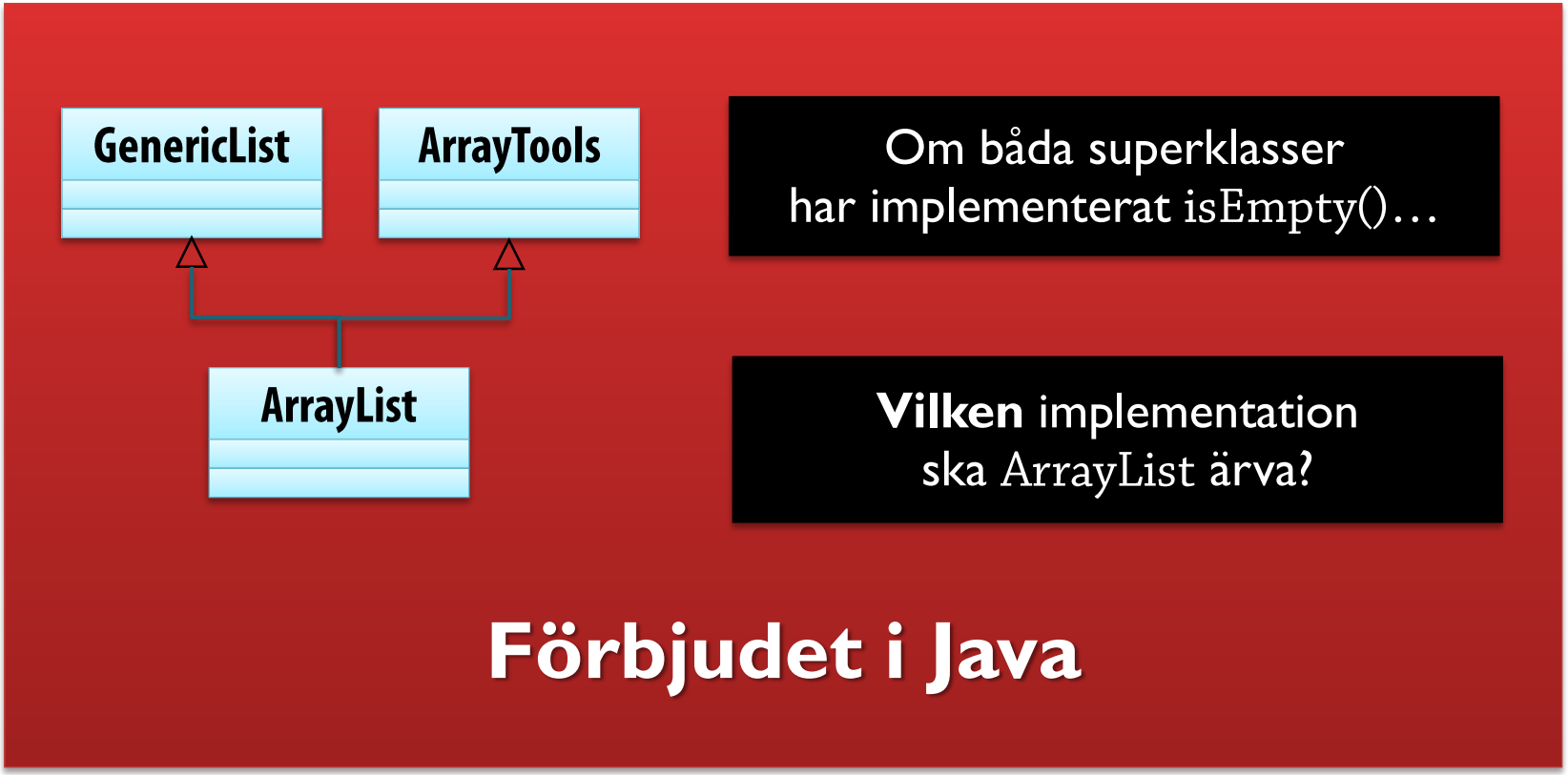
Den här implementationen *ersätter* en annan!

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

Multipel ärvning?

jonkv@ida

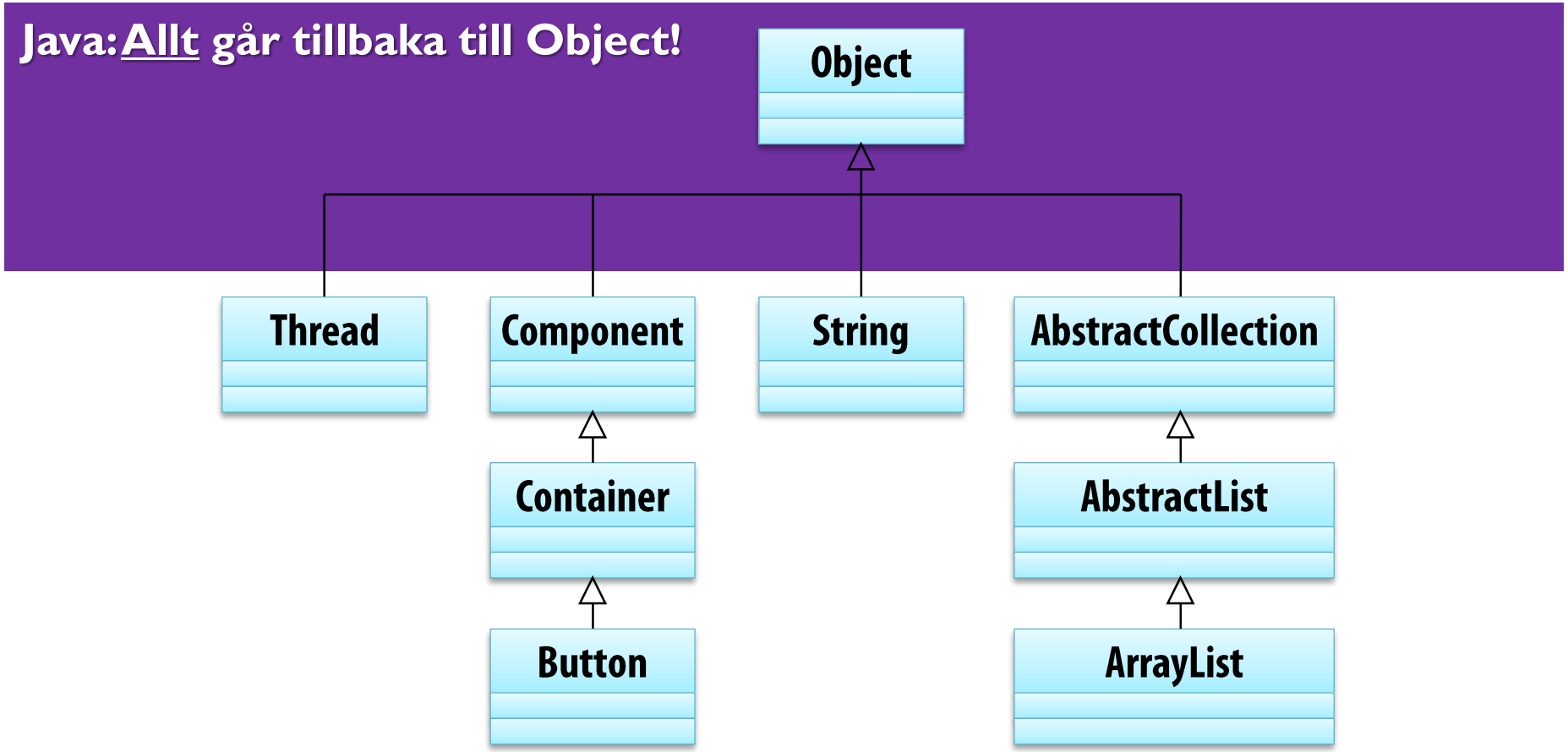
- **Multipel ärvning** i vissa språk: Utöka **flera klasser**
  - Användbart och förvirrande



Rötter i klasshierarkin

jonkv@ida

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



Enkelrotad klasshierarki

jonkv@ida

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

Implementationsarv och konstruktorer

Konstruktorer och arv 1

jonkv@ida

- **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 konstruktorer 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 subklass!

Konstruktorer 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!

Konstruktorer: Skicka vidare!

jonkv@ida

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

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

Tom Circle-konstruktor

```
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-konstruktor
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

Konstruktorer: Försvara klassen!

jonkv@ida

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

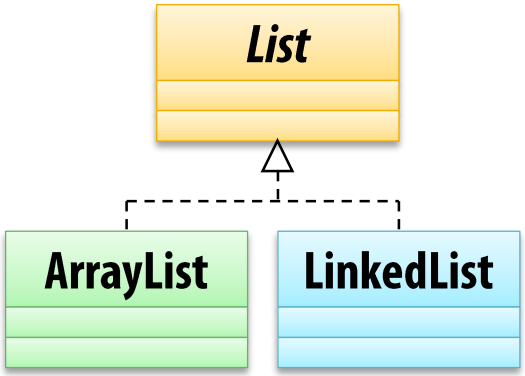
```
public class Circle {
  public double x, y, r;
  public Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }
}
```

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

Abstrakta klasser i Java

Motivation

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



”X är en speciell sorts Y, som…”

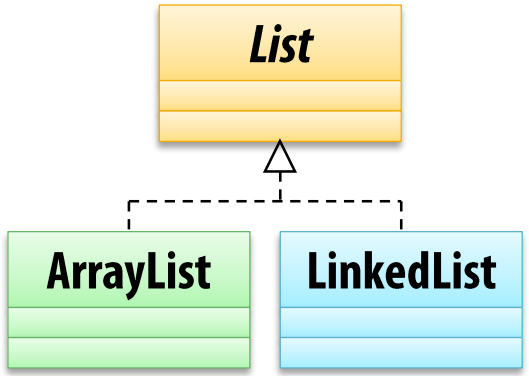
  - Naturligt att prata om *grafiska komponenter* i allmänhet
    - ➔ Vissa komponenter är *knappar*
    - ➔ Vissa komponenter är *textfält*
    - ➔ …

Men ibland vill vi helt enkelt 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?

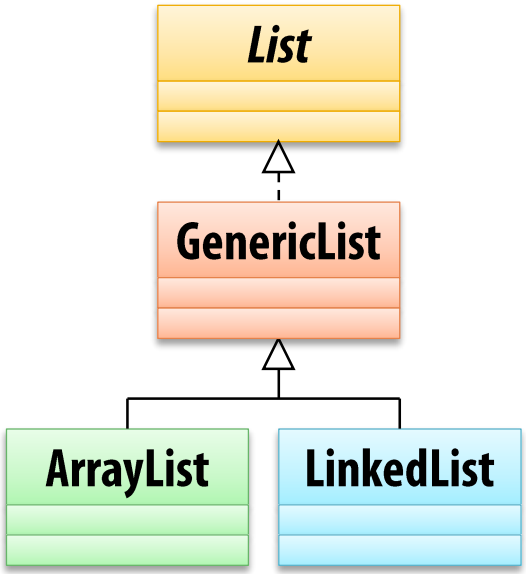
Inte flytta till **List**: Gränssnitt ska ange *krav*, inte *implementation*

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) { … }
}
```

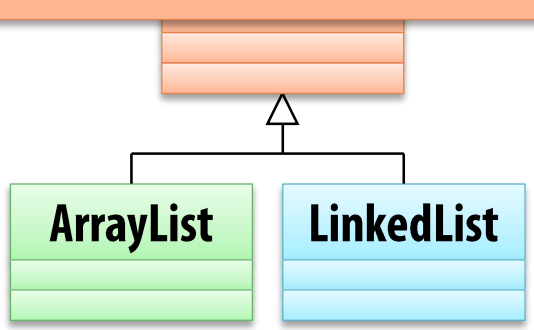
Försök 2: Dummy-implementation?

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*

Bara för att undvika redundant kod!

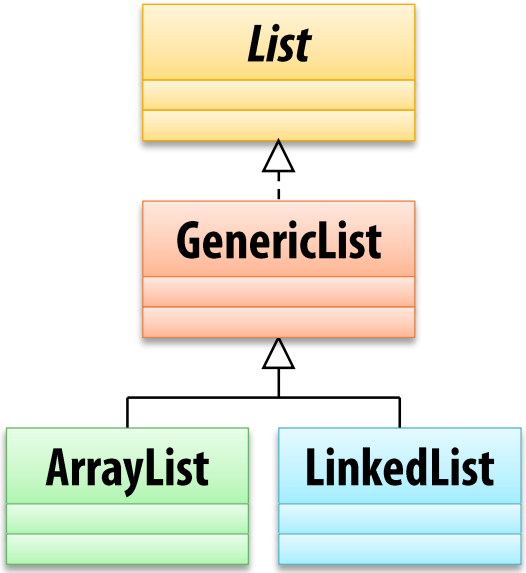
Abstract 2: För återanvändning!

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: 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;
   }
}
```

Gränssnitt –
eller abstrakt klass?

Abstrakt klass eller gränssnitt? (1)

jonkv@ida

Gränssnitt

- Kan implementera flera

class LinkedList
implements List, Queue, …

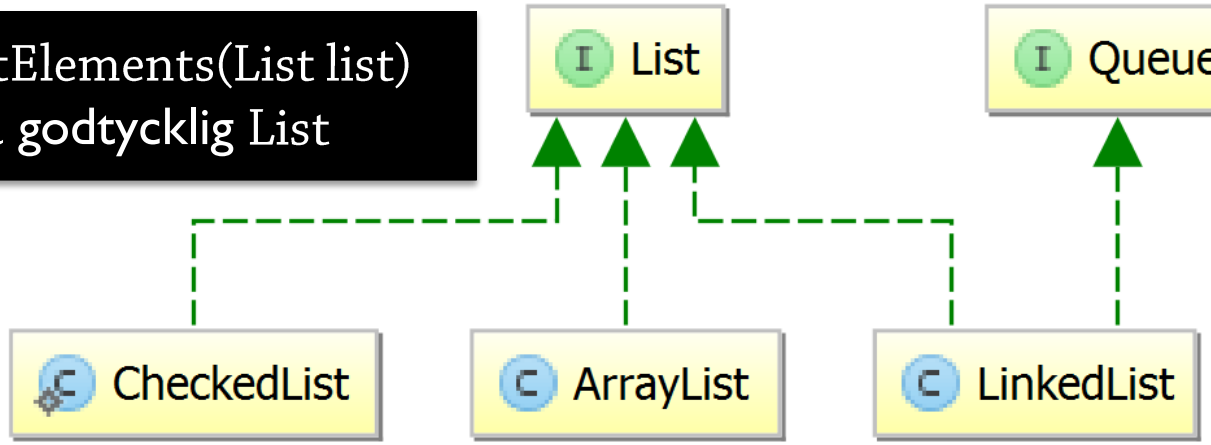
Kan vara både List och Queue

- Ingen kod, bara löften / kontrakt

interface List { … }

Alla som implementerar måste ha kod för alla metoder

void printElements(List list)
kan ta godtycklig List



Vem som helst kan implementera List:
Även om man också *implementerar* Queue, eller *utökar* en klass

Gränssnitt ger ingen gemensam kod att ärva

Abstrakt klass eller gränssnitt? (2)

jonkv@ida

Abstrakta klasser

- Ger löften *och* kod

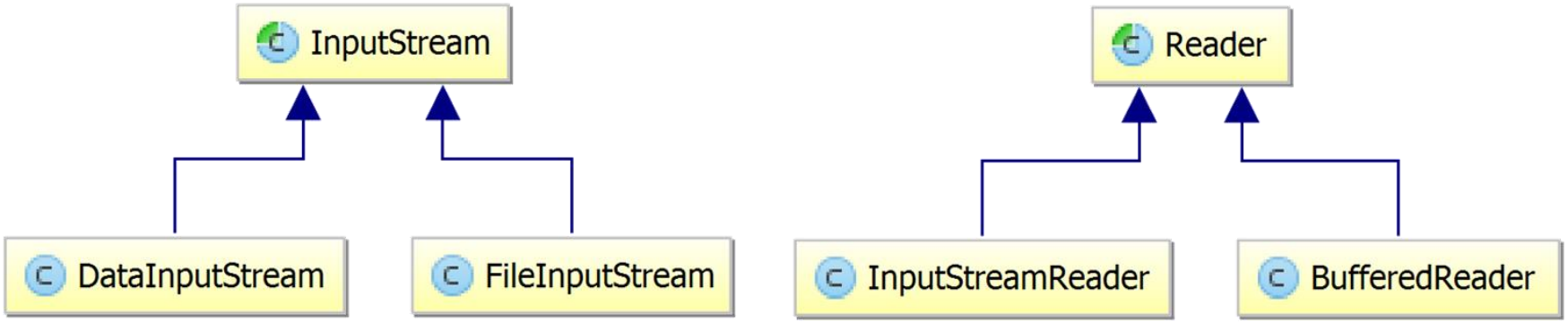
abstract class Reader { … }
abstract class InputStream { … }

Klasser tillhandahåller kod som alla subklasser använder

- Kan bara utöka *en*

class BufferedReader
extends Reader [och inget mer]

Ingen klass kan vara Reader *och* InputStream



Subklasser till InputStream och Reader
ärver användbar kod…

Abstrakt klass eller gränssnitt? (3)

jonkv@ida

- Javas I/O-klasser har suboptimal design…
  - "Basen" i dessa hierarkier är abstrakta klasser, inte gränssnitt
  - ➔ I/O-kod deklarerar parametrar, fält, variabler av *klasstyperna* InputStream och Reader!

```
List readElementsFrom(InputStream stream) {
    …
}
```

För att skapa en ny strömtyp som metoden kan använda, *måste* man utöka InputStream

Då kan klassen inte utöka något annat…

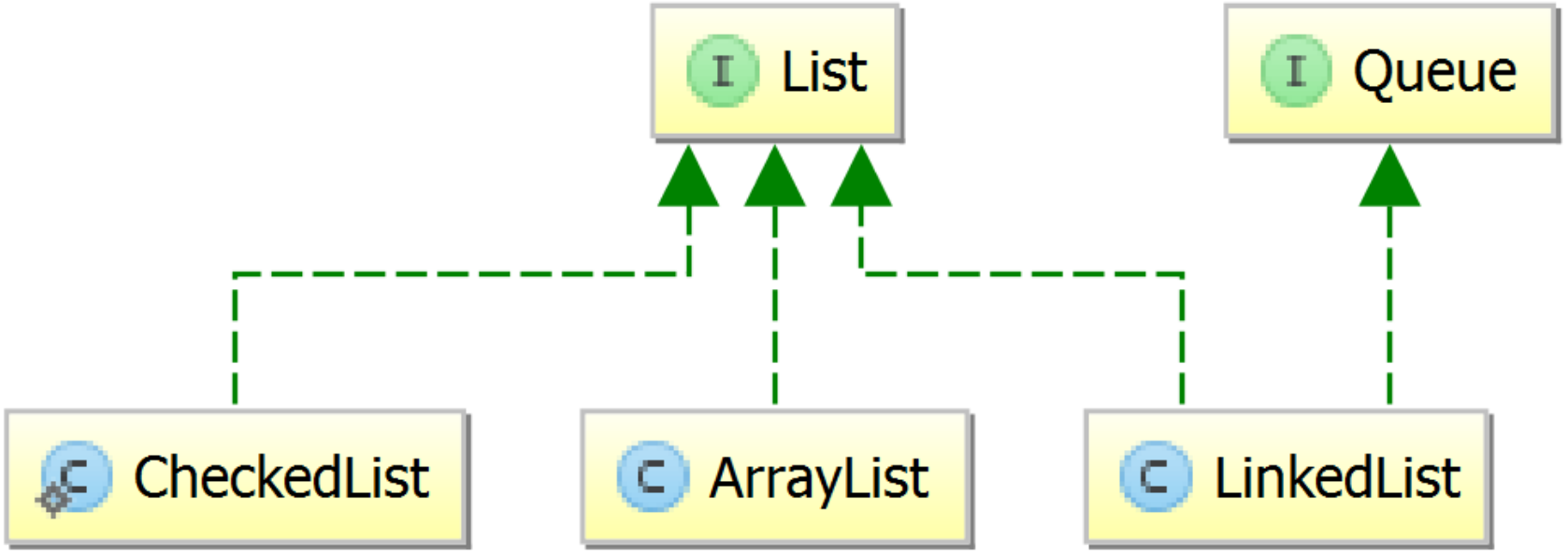
Hade List och Queue varit abstrakta klasser, kunde LinkedList inte ha varit både List and Queue!

Abstrakt klass eller gränssnitt? (4)

jonkv@ida

- Bra teknik: Använd båda!

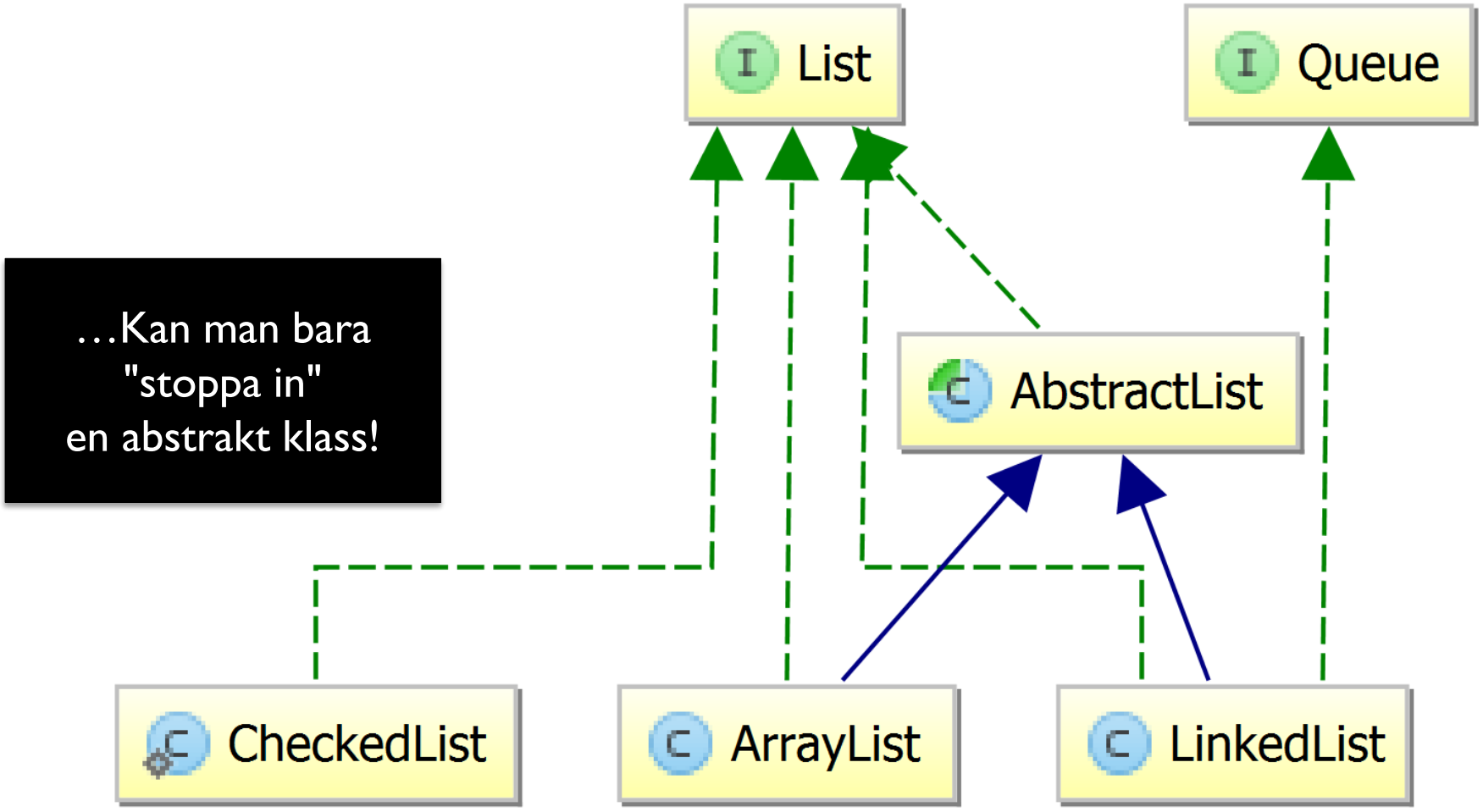
Hierarkiernas "baser" *är* gränssnitt
(List, Queue)
➔ används för deklarationer etc.



Man *kan* implementera dessa direkt
➔ full frihet

Om man har mycket gemensam kod,
och kan tjäna på att ha
en abstrakt klass…

Abstrakt klass eller gränssnitt? (5)



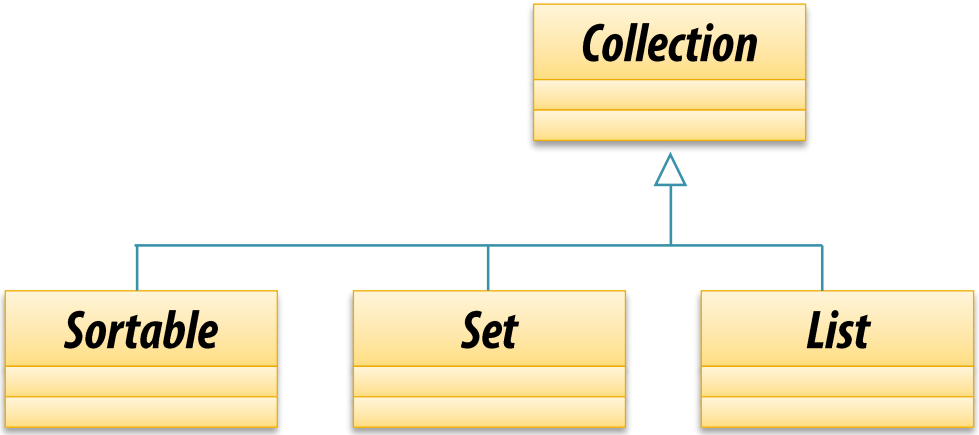
Metoder kräver/returnerar fortfarande **List**, inte **AbstractList**

➔ kan ta/returnera en CheckedList skriven *utan* den abstrakta klassen

Sammanfattning

Sammanfattning: Klasshierarkier

jonkv@ida

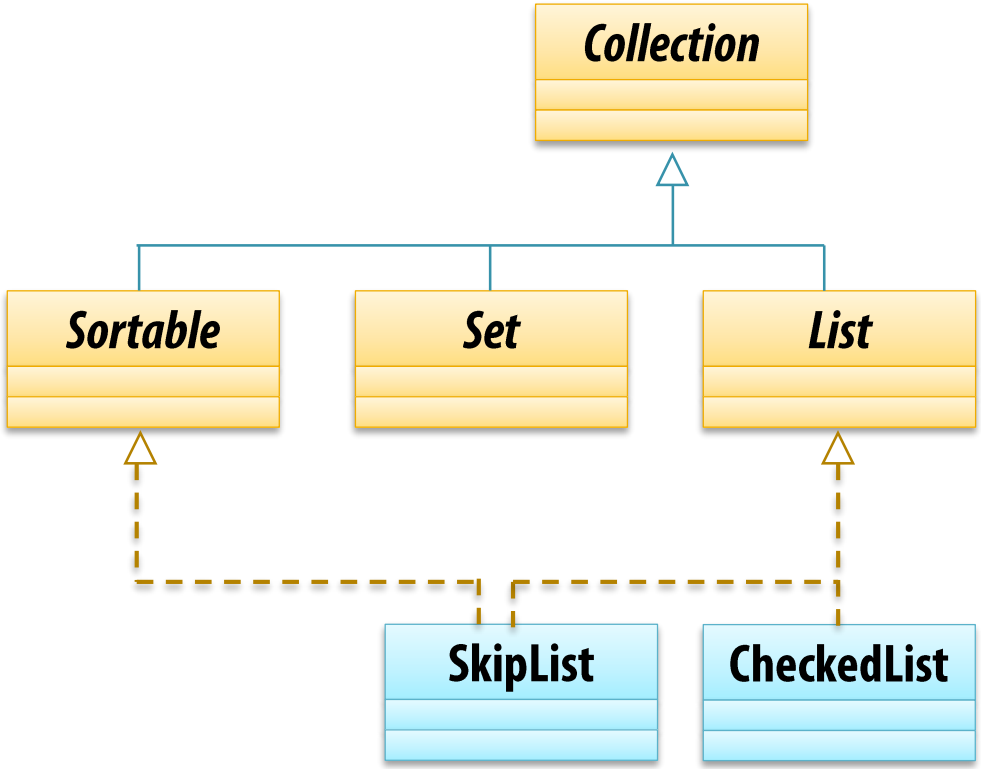


2. Ett gränssnitt kan
ärva från och ***utöka***
(löftena som gavs av)
ett annat gränssnitt

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

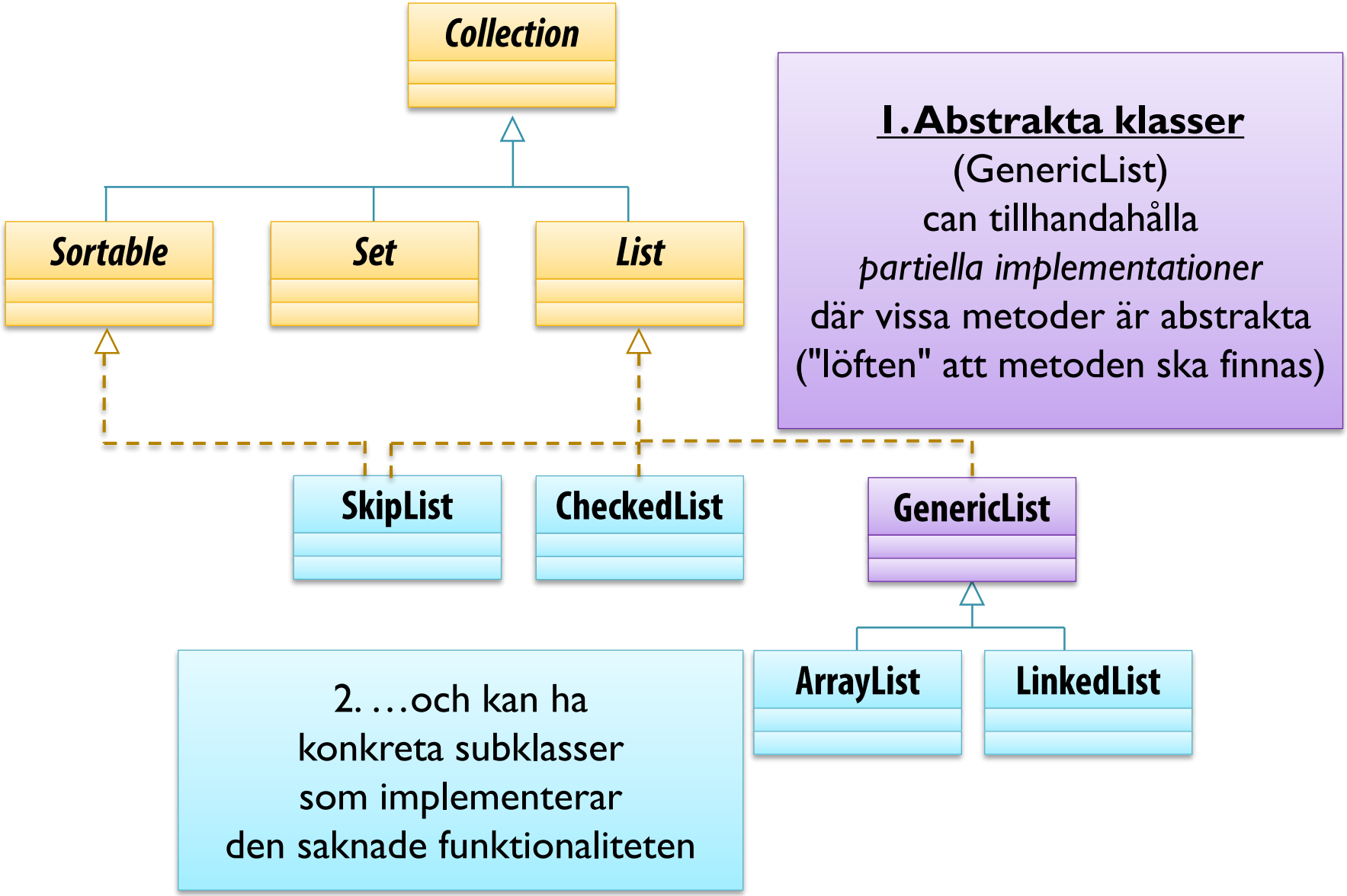
2. En metod kan ta ”godtycklig List” som parameter. Kan vara en SkipList eller CheckedList – spelar ingen roll!

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

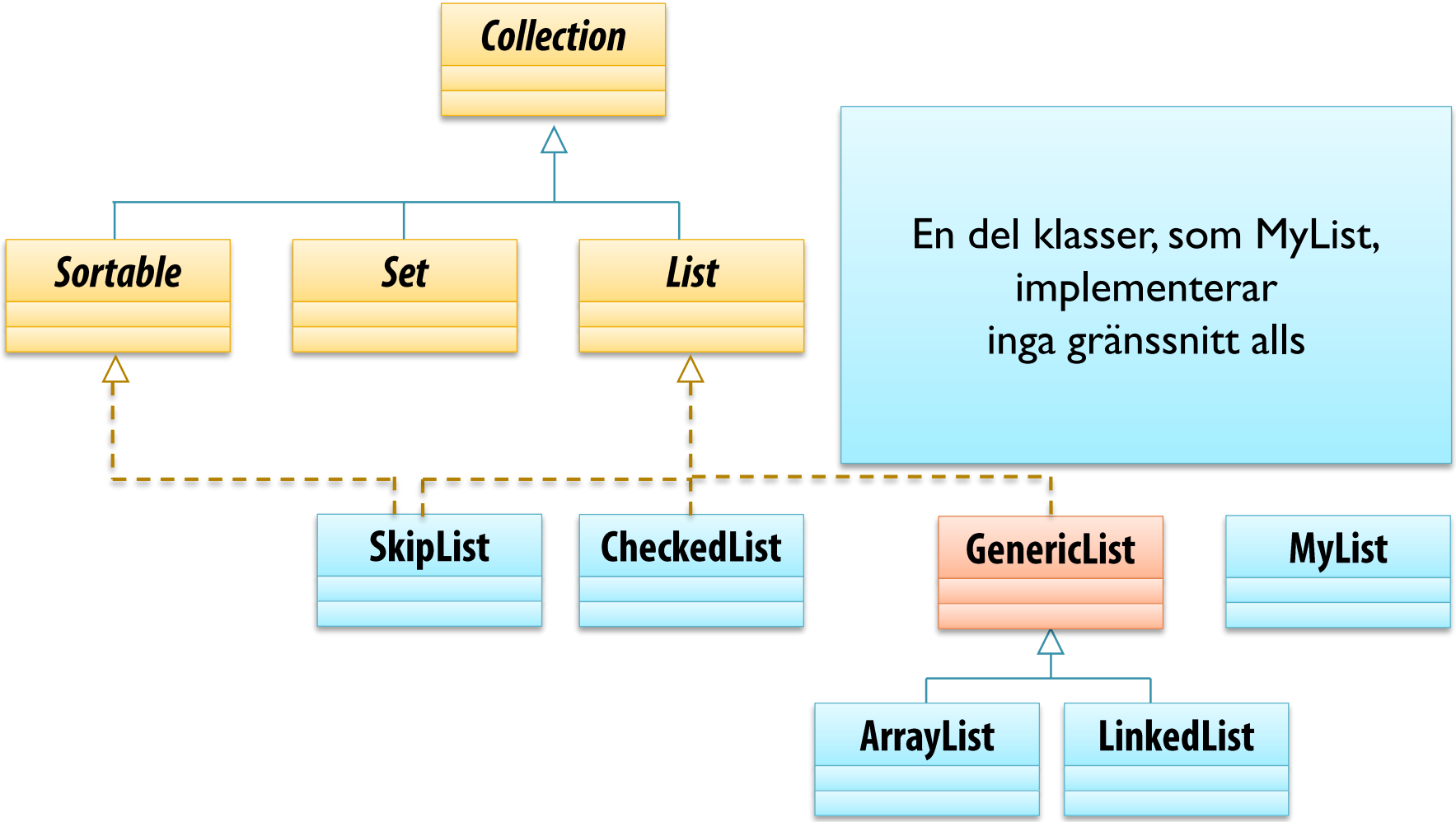


1. Abstrakta klasser
(GenericList)
can tillhandahålla
partiella implementationer
där vissa metoder är abstrakta
("löften" att metoden ska finnas)

2. …och kan ha
konkreta subklasser
som implementerar
den saknade funktionaliteten

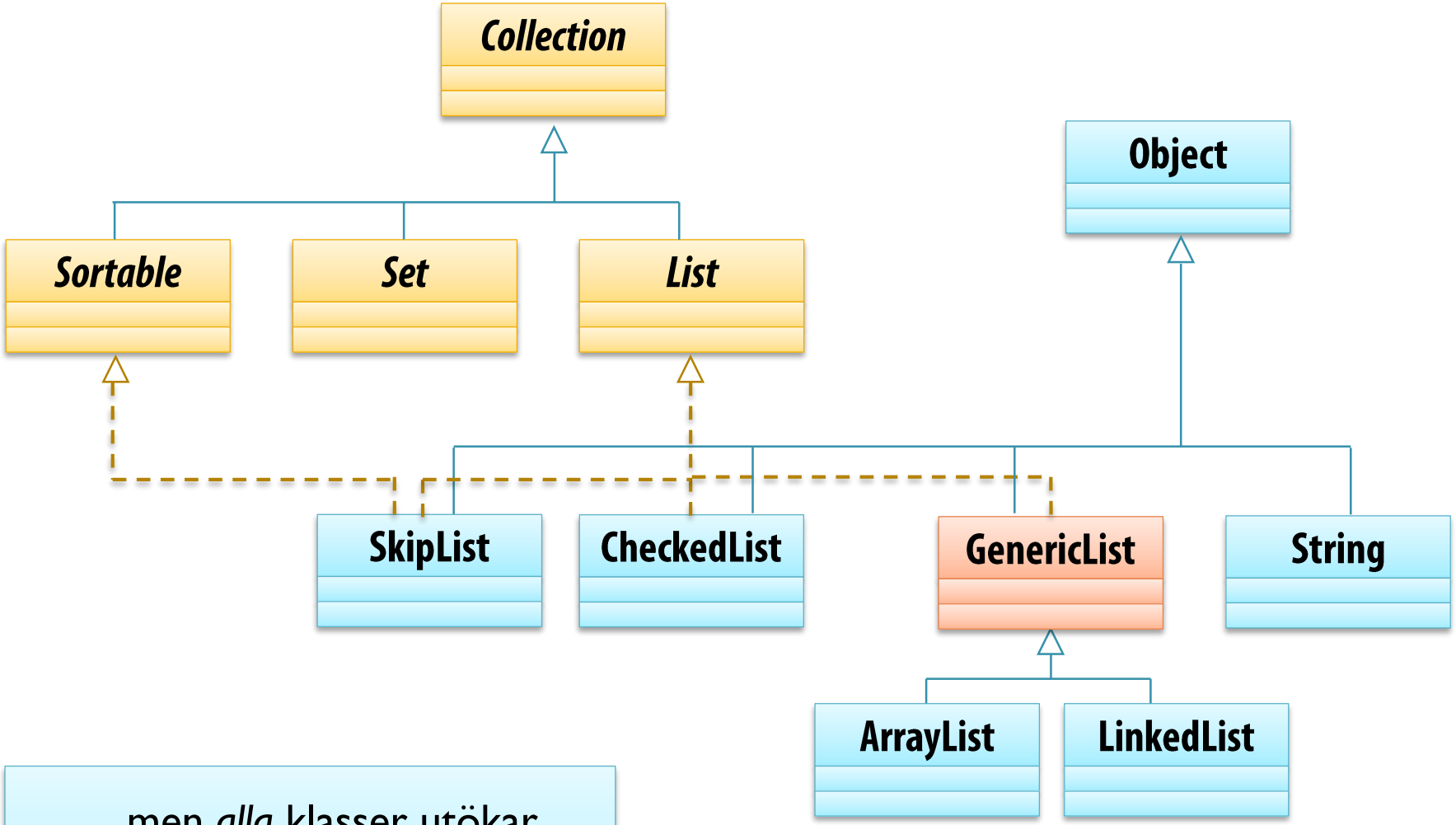
Sammanfattning: Klasshierarkier (4)

jonkv@ida



Sammanfattning: Klasshierarkier (5)

jonkv@ida



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

Att använda ärvning – eller inte

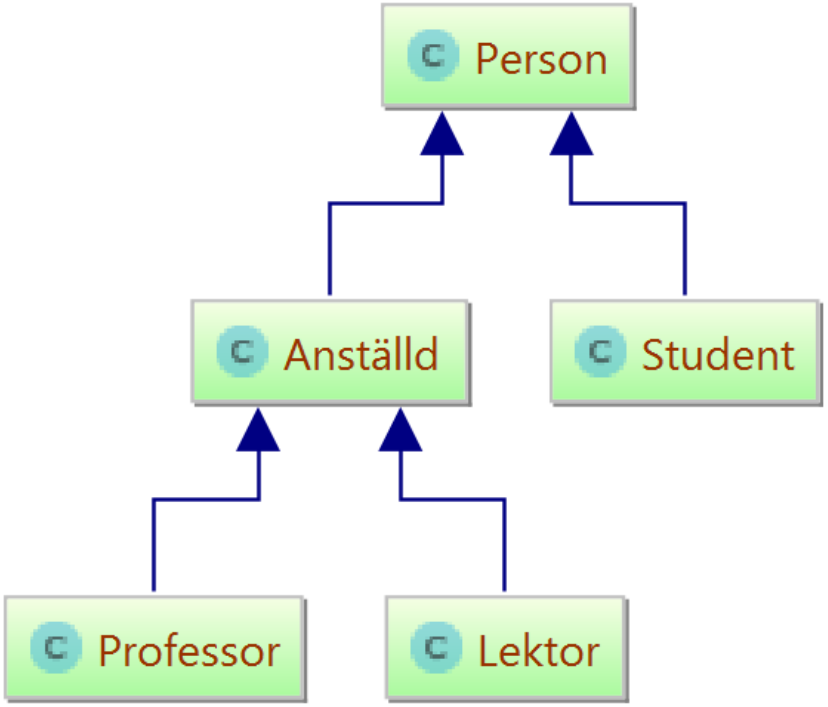
jonas.kvarnstrom@liu.se – 2017

Ärvning: Fördelar och nackdelar!

Personer 1: Klasser

jonkv@ida

- Vi vill ha en *persondatabas* för universitetet
  - **Ett** sätt att modellera: En klass för varje ”typ” av person

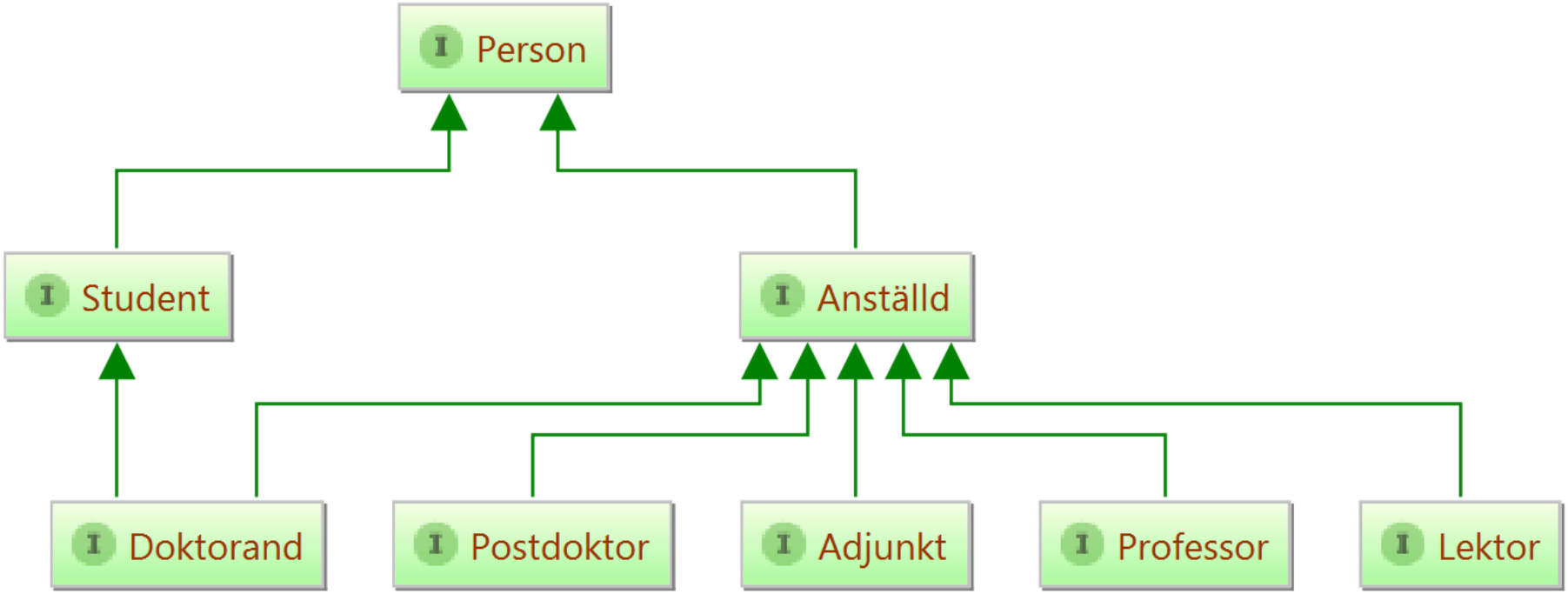


Problem: Doktorander är både anställda och studenter

Personer 2: Gränssnitt

jonkv@ida

- OK, vi använder gränssnitt – multipel ärvning!



Problem: Hur byter man typ?

Personer 3: Byta typ

jonkv@ida

- Många personer **byter ”typ”** under sin livstid
  - Student ➔ doktorand ➔ postdoktor ➔ lektor ➔ professor?

- Objekt **kan inte byta typ**
  - Vi måste skapa **nytt** objekt
    - Student jonas1 = new Student(…);
      …
      Doktorand jonas2 = new Doktorand(jonas1.namn, jonas1.personnummer, …)

- Praktiskt problem: Det gamla objektet kan lagras på många platser
  - Måste **bytas ut** mot det nya, *överallt*

- Begreppsproblem: **Jag** byttes inte ut mot en ny person!

Personer 4: Alternativ klassificering

jonkv@ida

- Varför dela upp enligt just **anställningsform**?
  - Andra klassificeringar:
    - **Prefekt, dekan, rektor** – en *roll* som en professor kan ha
    - Medlem i **utbildningsnämnd**, eller inte?
    - Medlem i **institutionsstyrelse**, eller inte?
    - …
  - Ska allt detta motsvaras av egna gränssnitt?

Personer 4: Annan lösning

- Försök inte ”tvinga in” ärvning för ärvningens skull!
  - Alternativ lösning:

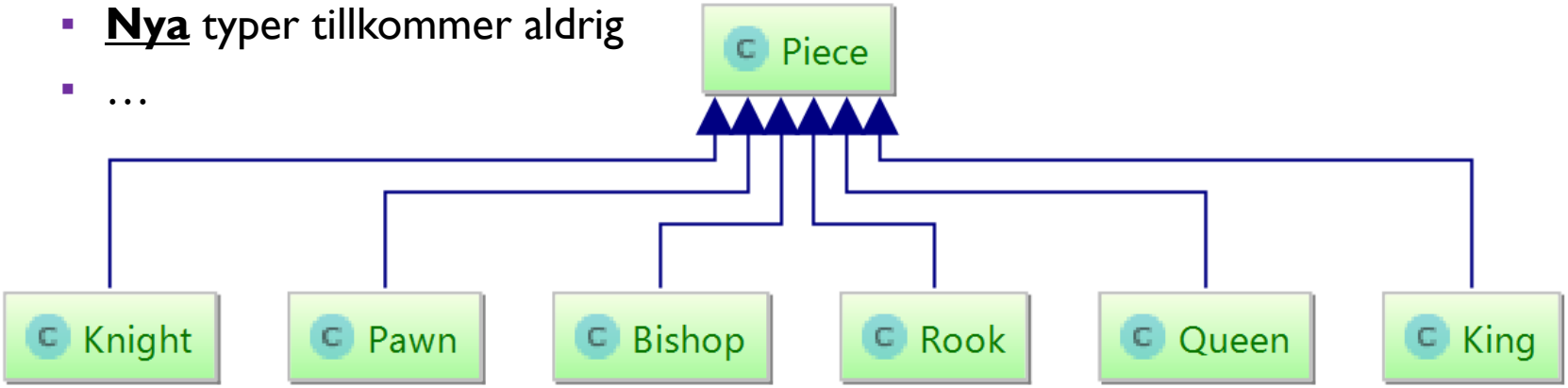
```
public class Person {
   private boolean student;
   private boolean anställd;

   public enum Anställningsform { ADJUNKT, DOKTORAND, PROFESSOR,  … }
   private Anställningsform anställning;

   public enum Roll { PREFEKT, DEKAN, UTBILDNINGSNÄMND, … }
   private Set<Roll> roller;
}
```

Anställningsform är en **egenskap** som kan ändras

En eller flera roller, kan också ändras

Schack: En kontrast

- Schackspel:
  - Kan ha flera klass för pjäser:
    - Varje pjäs har **en** typ
    - Eget **beteende** *(förflyttningsregler)*, kanske tillräckligt komplext för egen klass
    - Pjäser **byter** sällan typ – kan enkelt hanteras
    - **Nya** typer tillkommer aldrig
    - …



  - **Kan** välja att ha **en** klass
    - Avvägning: Vilket blir mest läsbart, modulärt?
  - **Inte** separata klasser för vitt/svart – alltför lika beteende

Om du bara har en hammare, ser alla problem ut som spikar

Men vi har flera verktyg än ärvning!

Ärvning:
Undvik onödiga underklasser

Skapa inte klasser för *klassernas* skull

Onödiga klasser 1.1

jonkv@ida

```
class GenericPlayer {
    int strength;
    Image img;
    GenericPlayer(int strength, Image img) {
        …
    }
}
```

```
class BasicPlayer extends GenericPlayer {
    BasicPlayer() {
        super(20, getImage("basicplayer.png"));
    }
}
```

```
class SuperPlayer extends GenericPlayer {
    SuperPlayer() {
        super(50, getImage("superplayer.png"));
    }
}
```

Inget beteende,
bara *data* (värden)

Subklasser ska ha eget **beteende**:
Nya eller ändrade metoder!

Här skiljer sig bara konstruktorn

Dålig lösning:
Onödiga klasser

Kan anropa **new** GenericPlayer(…)
med parametrar
➔ slipper två klasser
➔ fördelar i flexibilitet

Onödiga klasser 1.2: Fabriksmetoder

jonkv@ida

- Om man ändå vill ge namn för att förenkla:
  - Använd **fabriksmetoder**

```
public class PlayerFactory {
   public Player createBasicPlayer() {
      return new GenericPlayer(20, getImage("basicplayer.png"));
   }
   public Player createSuperPlayer() {
      return new GenericPlayer(50, getImage("superplayer.png"));
   }
}
```

  - Behöver inte alltid en egen "fabriksklass"
    - Anropas create* bara från GameMechanics?
    - Då kanske de ska ligga i GameMechanics istället

Onödiga klasser 2.1

jonkv@ida

```
class GenericPlayer {
    int strength;
    Image img;
    GenericPlayer(int strength, Image img) {
        ...
    }
}
```

```
class BasicPlayer extends GenericPlayer {
    void leapForward() {
        x += 10;
    }
}
```

```
class SuperPlayer extends GenericPlayer {
    void leapForward() {
        x += 20;
    }
}
```

Här finns **metoder** med olika kod
Men den enda skillnaden är i *värden*!
Ska **inte** ha olika klasser

Onödiga klasser 2.2: Lösning


```
class GenericPlayer {
    int strength;
    int leapLength;
    Image img;
    GenericPlayer(int strength, Image img, int leapLength) {
        ...
    }
}
```

Bra, generell lösning:

Kan lätt ange *godtycklig* hopplängd
Kan hitta på hopplängder *under programmets körning* (ingen ny klass krävs)

Använd inte klasser när värden räcker

Ärvning eller komposition (sammansättning)?

Att vara eller icke vara

jonkv@ida

- **Är den, eller har den?**

Gör vi så här, ärver Circle funktionalitet från Point…
Trevligt! Gratismetoder!

```
public class Circle extends Point {
  // Ärver x,y
  private double r;
  public Circle(double x, double y, double r) {
    super(x,y);
    this.r = r;
  }
  …
}
```

Är en cirkel en sorts punkt?

Nej!

Att vara eller icke vara (2)

jonkv@ida

- En cirkel **har** en punkt – mittpunkten
  - Vill vi erbjuda liknande metoder? Delegera!

```
public class Circle {
  private Point center;
  private double r;

  public Circle(Point center, double r) {
    this.center = center;
    this.r = r;
  }
  public double getDistFromOrigin() {
     return center.getDistFromOrigin();
  }
  …
}
```

Att vara eller icke vara (3)

- **Är den, eller har den?**

Gör vi så här, ärver Queue funktionalitet från ArrayList…
Trevligt! Gratismetoder!

```
public class Queue extends ArrayList {
    …
    public Object pop() {
        Object obj = get(size()–1);
        remove(size()-1);
        return obj;
    }
}
```

Är en kö en sorts lista?

Nej!

Listor kan stoppa in element var som helst, ta bort var som helst, …
En kö kan ha en lista (se labbar)

Om du bara har en hammare, ser alla problem ut som spikar

Använd komposition när det passar!

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

Typkontroll: Operatorn instanceof

jonkv@ida

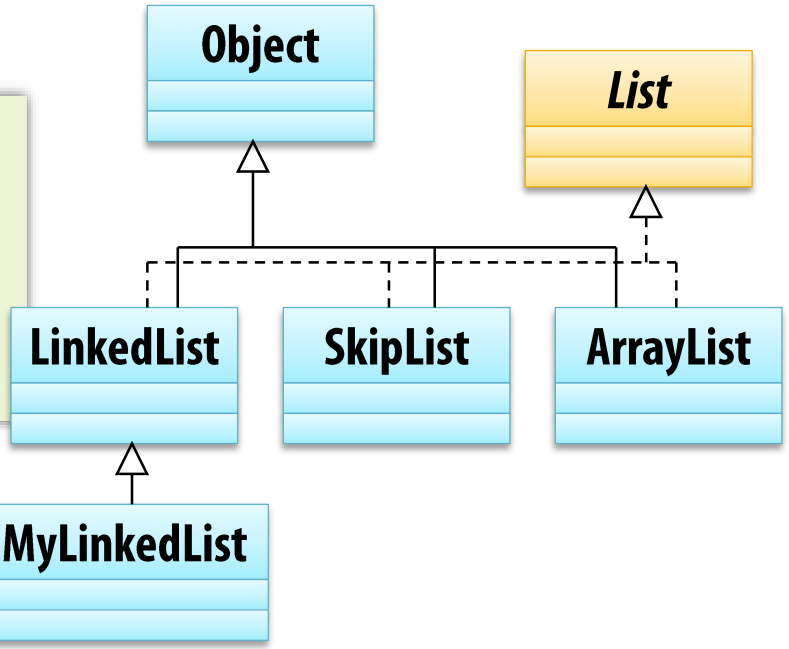
- Vilken typ har ett visst objekt?
  - **Exakt** typ: object.getClass() == TypeName**.class**

```
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)

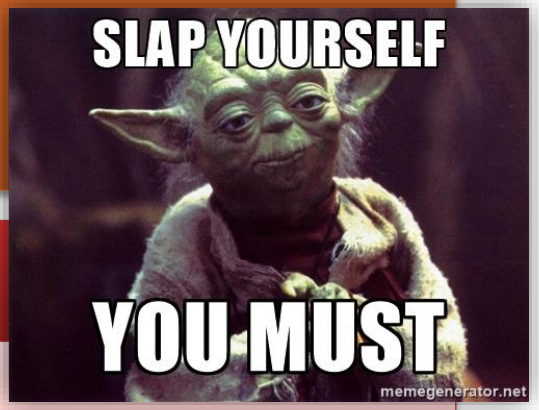
jonkv@ida

- **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);
}
```

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

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

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

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

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

En kodlukt: Upprepning

Diskuteras i labb 4

Kodlukt

- **Kodlukt / code smell**:
  - Inte nödvändigtvis en bugg i sig
  - Något man kan se "**på ytan**" på koden,
    som ofta indikerar att det finns ett **djupare problem**
  - Fixas ofta genom **refactoring**: Kod skrivs om, men gör exakt samma sak

Redundans och upprepning

Redundans?

jonkv@ida

- I vissa sammanhang kan redundans ha sina poänger…
  - *This parrot is no more. It has ceased to be.*
    It's expired and gone to meet its maker.
    This is a late parrot. It's a stiff. Bereft of life, it rests in peace.
    If you hadn't nailed it to the perch, it would be pushing up the daisies.
    It's rung down the curtain and joined the choir invisible.
    This is an ex-parrot.

Undvik redundans

jonkv@ida

- I programmering ska redundans och upprepning undvikas!
  - Gör det **svårare att förstå** koden
    - Mer att läsa
    - Måste se om upprepningen är exakt lika eller skiljer sig på något sätt
  - **Mer kod att underhålla**
  - **Risk att inte all kod uppdateras** på samma sätt

DRY-principen: Don't Repeat Yourself!
DIE: Duplication Is Evil

WET: Write Everything Twice
WET: We Enjoy Typing…

Onödigt många metoder

Onödigt många metoder

jonkv@ida

- Kod ska skrivas **effektivt** – exempel:

```
public void activateUp() {
    up = true;
}
public void deactivateUp() {
    up = false;
}
```

```
public void setUp(boolean up) {
   this.up = up;
}
```

```
public void moveLeft() {
    …;
}
public void moveRight() {
    …;
}
```

```
public void move(Direction dir) {
    …;
}
```

Onödig upprepning av kodmönster

```
public void checkKeyboard() {
    if (isPressed(Key.UP)) {
        player.directions.add(Direction.NORTH);
    } else {
        player.directions.remove(Direction.NORTH);
    }
    if (isPressed(Key.DOWN)) {
        player.directions.add(Direction.SOUTH);
    } else {
        player.directions.remove(Direction.SOUTH);
    }
    if (isPressed(Key.LEFT)) {
        player.directions.add(Direction.WEST);
    …
```

Extrahera en hjälpmetod!

```
public void checkKeyboard() {
    checkKey(Key.UP, Direction.NORTH);
    checkKey(Key.DOWN,Direction.SOUTH);
    checkKey(Key.LEFT, Direction.WEST);
    checkKey(Key.RIGHT, Direction.EAST);
}
```

```
private void checkKey(Key key, Direction dir) {
    if (isPressed(key)) {
        player.directions.add(dir);
    } else {
        player.directions.remove(dir);
} }
```

Upprepning av kodmönster (2)

jonkv@ida

- Extrahera **upprepade uttryck** till namngivna variabler

NameProcessor.java

```
static void printNames(final String fullName) {
        System.out.println(fullName.substring(fullName.indexOf(" ") + 1));
        System.out.println(fullName.substring(0, fullName.indexOf(" ")));
}
```

Multiple occurrences found

Replace this occurrence only

Replace all 2 occurrences

NameProcessor.java

```
static void printNames(final String fullName) {
    int firstNameEndIndex = fullName.indexOf(" ");
    System.out.println(fullName.substring(firstNameEndIndex + 1));
        System.out.println(fullName.substring(0, firstNameEndIndex));
}
```

Snabbare kod

Namnet ger en *förklaring* till deluttrycket

Lättare att se att delarna är *medvetet* identiska

IDEA kan hjälpa till: Refactor | Extract | Variable

Upprepning i konstruktorer

jonkv@ida

- Många konstruktorer ➔ upprepad kod?
  - En konstruktor kan anropa en annan – en "kedja"

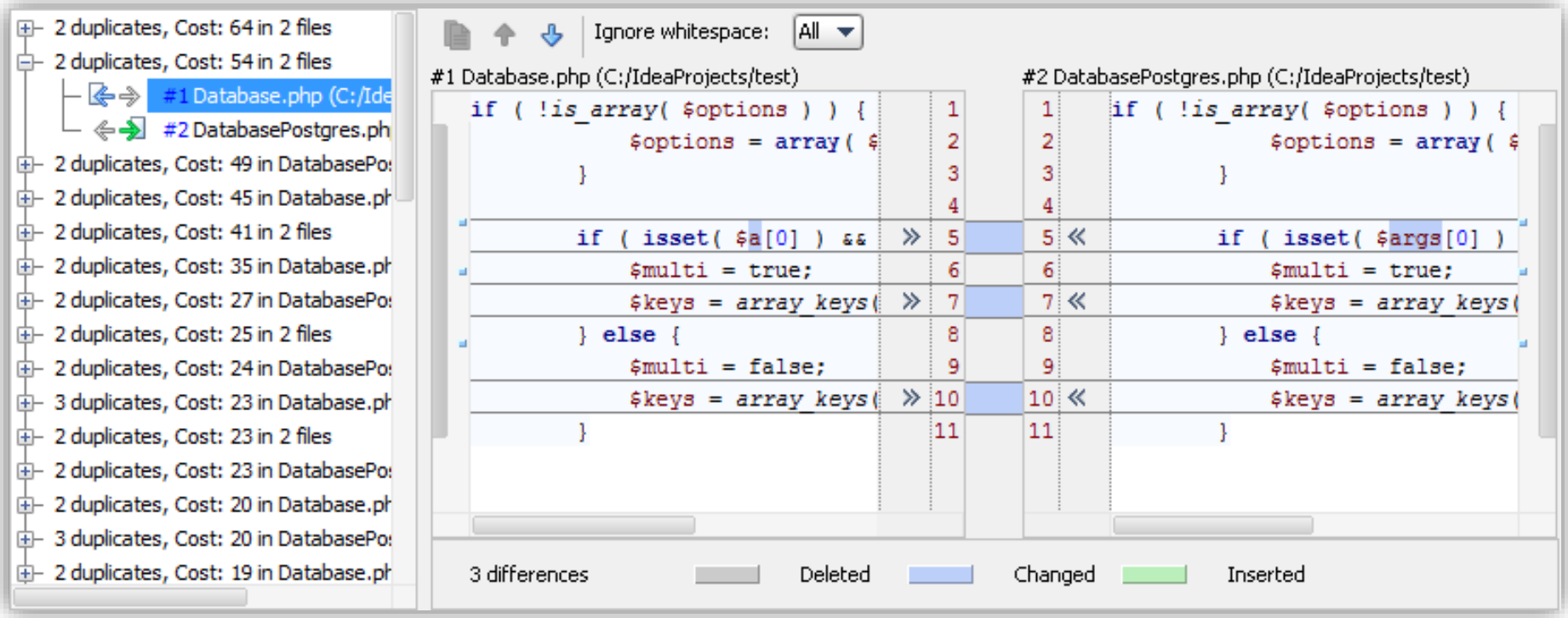
```
class Circle {
  double x, y, r;
  Circle(double x, double y, double r) {
    this.x = x; this.y = y; this.r = r;
    // Testa att r >= 0
    // Beräkna arean
    // Mer initialiseringskod...
  }
  Circle(Circle other) {
    this(other.x, other.y, other.r);
  }
}
```

Specialsyntax:
this(args)
som *första* sats i en konstruktor
vidarekopplar till en *annan* konstruktor

Att hitta upprepad kod

jonkv@ida

- IDEA kan hitta vissa former av **uppenbar repetition**
  - Analyze | Locate Duplicates



Ni gör resten – alltför ineffektiv kod ger komplettering!

Kan även hitta duplicerad kod ”on the fly”