

Typhierarkier del 3

När och hur vill vi använda dem?

Några "Best Practices"

Gränssnitt eller abstrakt klass?

Abstrakt klass eller gränssnitt? (1)



Gränssnitt

- Kan implementera flera

```
class LinkedList  
implements List, Queue, ...
```

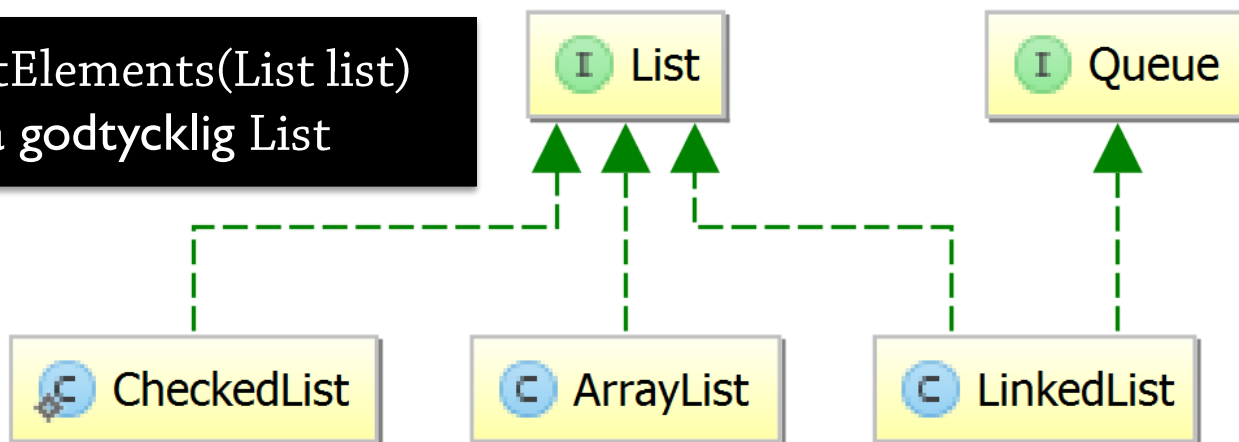
Kan vara både List och Queue

- 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)

Abstrakta klasser

- Ger löften och kod

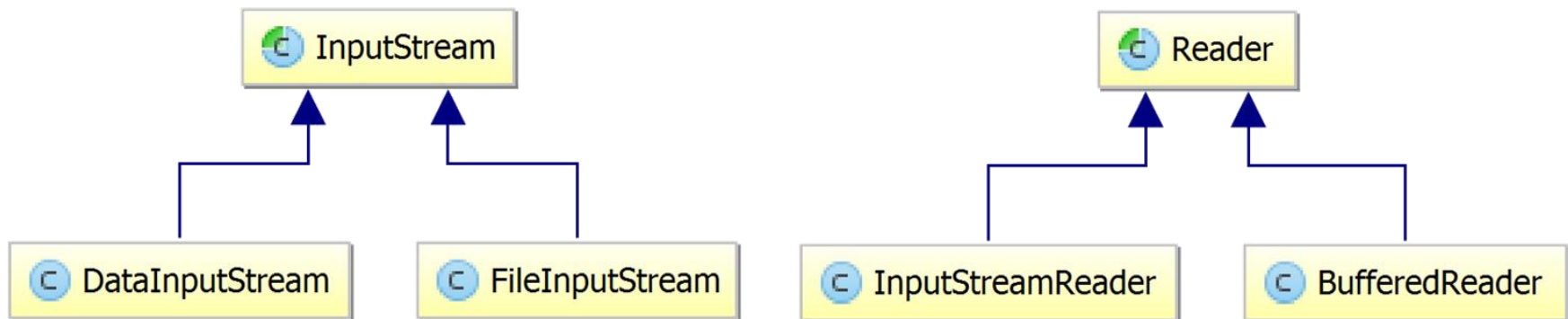
```
abstract class Reader { ... }  
abstract class InputStream { ... }
```

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)



- Javas I/O-klasser har suboptimal design...
 - "Basen" i dessa hierarkier är abstrakta klasser, inte gränssnitt
 - **InputStream** – läsa "råa" data
 - **Reader** – läsa text, avkoda teckenkodningar
 - ➔ I/O-kod deklarerar parametrar, fält, variabler av dessa **klasstyper**

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

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

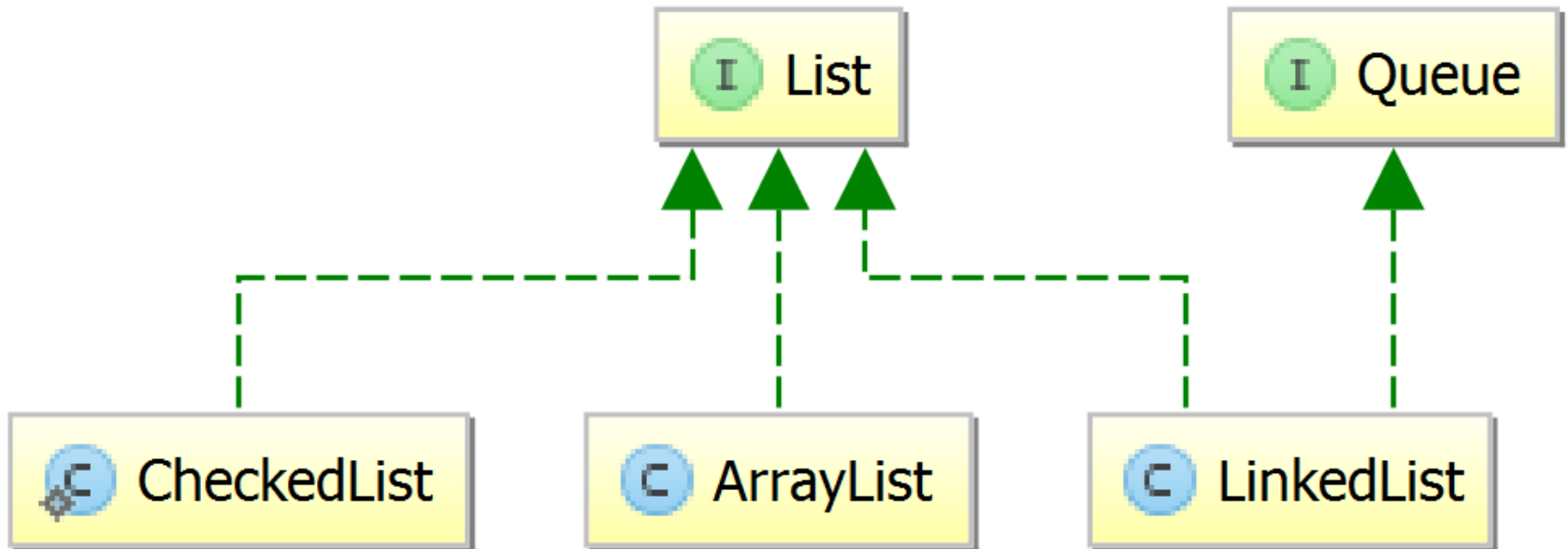
**Då kan klassen inte utöka
något annat...**

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

Abstrakt klass eller gränssnitt? (4)

- Bra teknik: Använd båda!

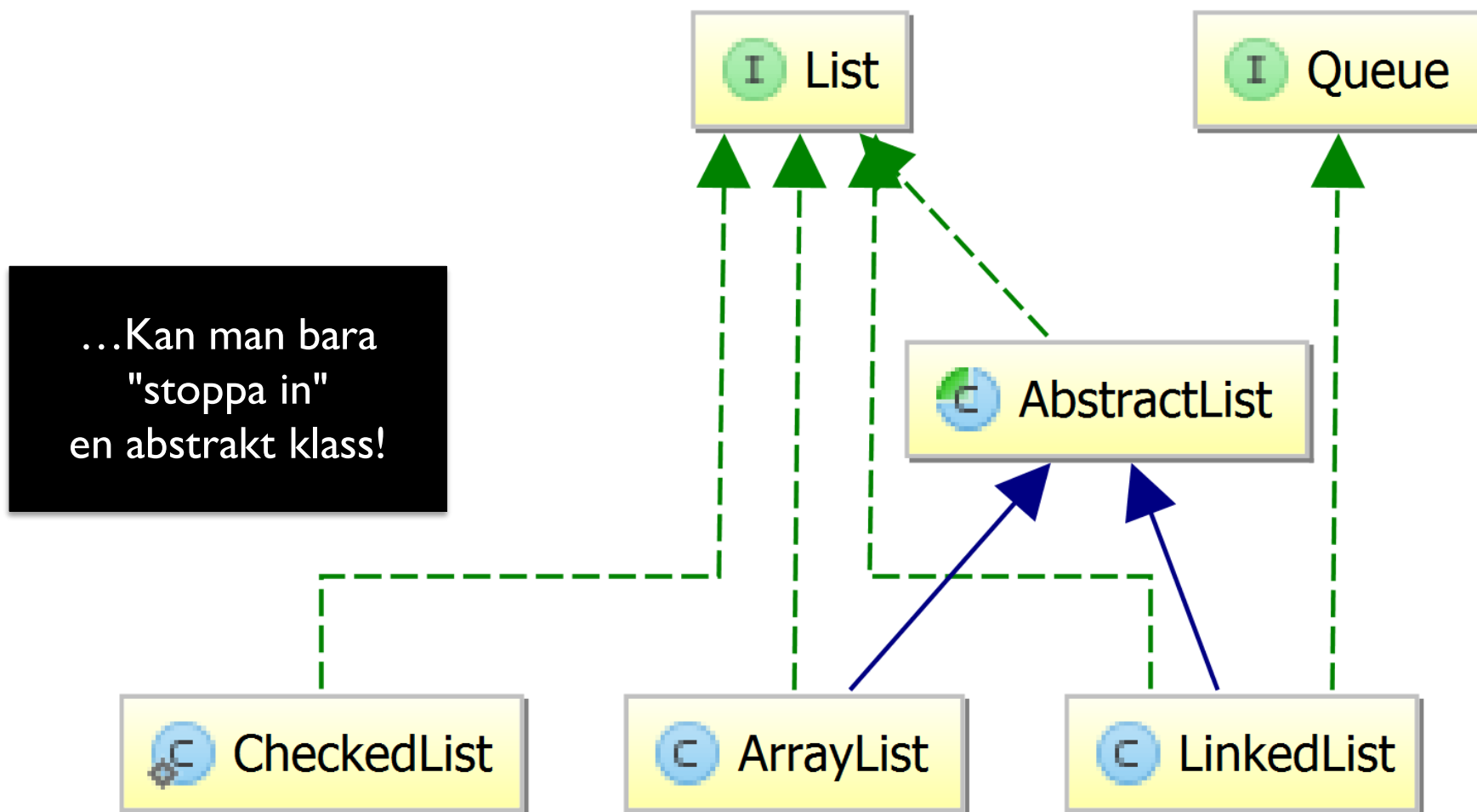
Hierarkiernas "baser" är gränssnitt
(List, Queue)
→ används för deklarationer etc.



Man *kan* implementera dessa direkt
→ full frihet

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

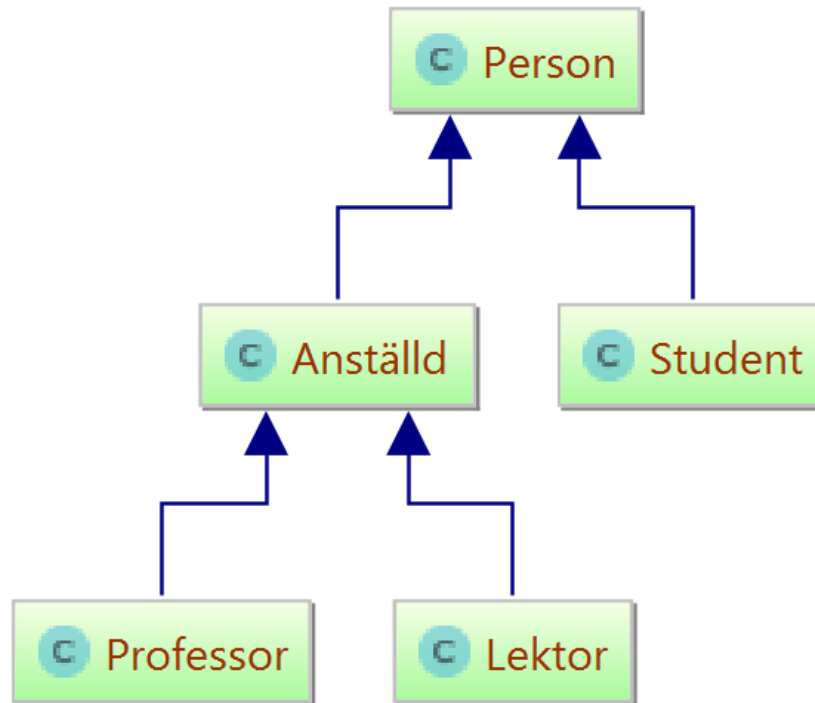
Abstrakt klass eller gränssnitt? (5)



Metoder kräver/returnerar fortfarande **List**, inte **AbstractList**
→ kan ta/returnera en **CheckedList** skriven *utan* den abstrakta klassen

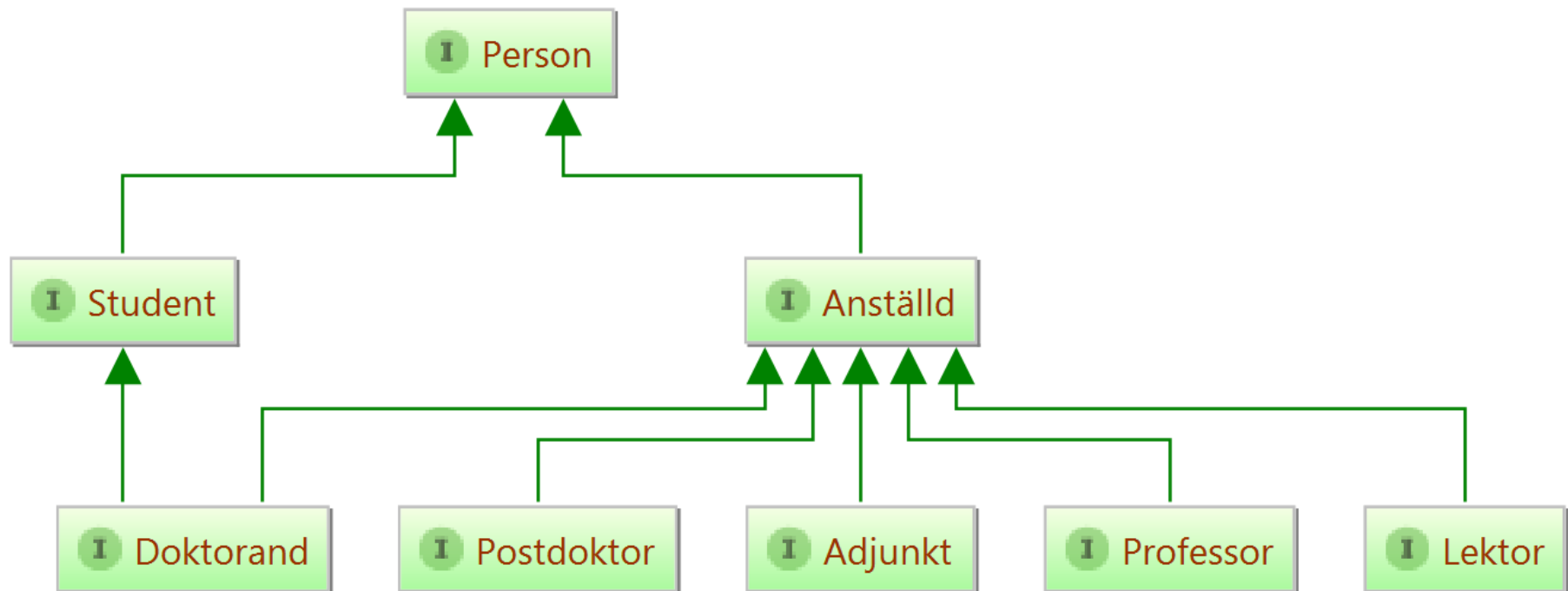
**Ska programmet använda ärvning
överhuvudtaget?
Fördelar och nackdelar!**

- 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

- OK, vi använder gränssnitt – multipel ärvning!



Problem: Hur byter man typ?

- 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 (pekas ut) på många platser
 - Måste **bytas ut** mot det nya, överallt
- Begreppsproblem: **Jag** byttes inte ut mot en ny person!

Personer 4: Alternativ klassificering



- 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;
```

Anställningsform är en egenskap
som kan ändras

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

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

```
}
```

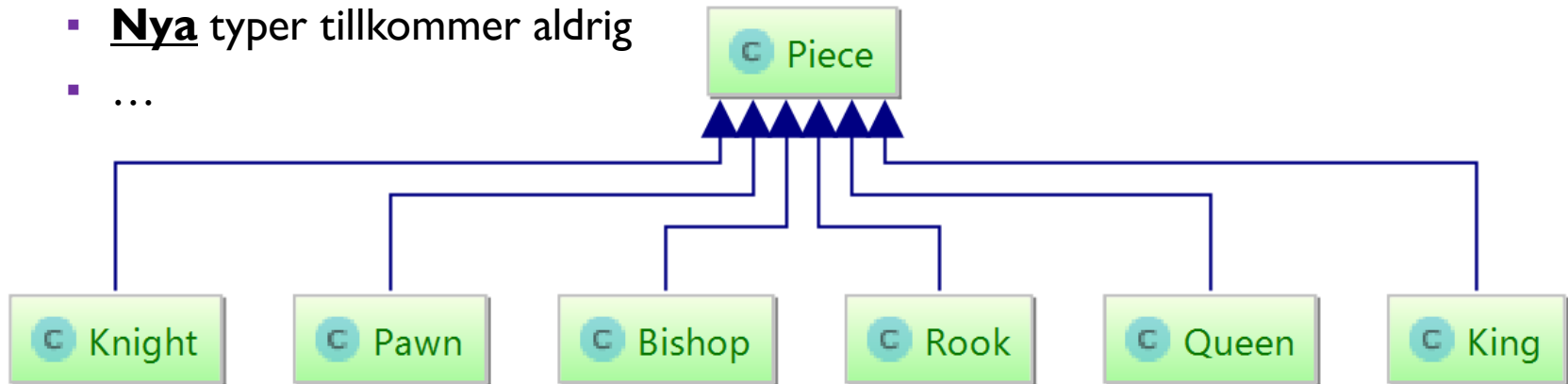
En eller flera roller,
kan också ändras

(Överkurs: Nästlad enum-klass – fullständigt namn **Person.Roll**,
användbart när en klass tydligt ”tillhör” en annan)

■ 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

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

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

Inget beteende,
bara *data* (värden)

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

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

Data är bättre än kod...

Onödiga klasser 1.2: Fabriksmetoder



- 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

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

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

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

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

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)?**

- 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;  
    }  
    ...  
}
```

Men är en cirkel en sorts punkt?

Nej!

Att vara eller icke vara (2)

- 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!