

Mer om OO-programmering och modellering

Från principer till konkreta projektråd

Vad ett program gör är viktigt

Hur det görs är lika viktigt!

- Ett dåligt exempel:

- **package** Hello;
class Hello {
 Hello Hello(Hello Hello) {
 Hello:
 for (;;) {
 if (Hello.Hello(Hello) == Hello) **break** Hello;
 }
 return Hello;
 }
}

■ Ett extremt exempel:

```
■ class Div{static $1 $_;class $1{void _$(String $_){System.//  
out.print($_);}$1(){_();}void _(){int _,$_,$$,__,ä=(1<<5),  
å=100,ö=12,åö=ä*ö;å-=1<<4;while(åö>0){for($$=_$_=__=$=(int)  
å;_$_>($$-(1<<2));_$((""+(char)(_$_)),_$_-=1<<1)for(_=$=__-9;_>(($_-6);_-=1<<1,_$((""+(char)(_$_!=å?_:ä)));char S$=(char)(å+ö+1)  
;_$(("te"+S$+(char)(ö+(int)S$));åö--;}}}Div(){$_=new $1();}  
public static void main(String []$){Div å=new Div();}}
```

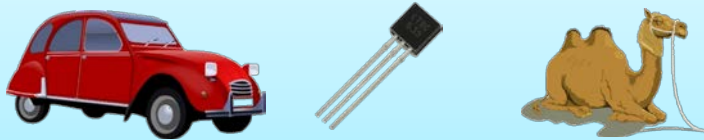
Skriver ut "TIGERteam" 384 gånger

Uppdelning i klasser och metoder

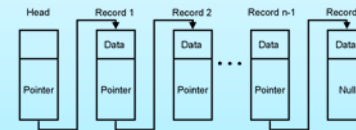
Hur vet vi
vilka klasser vill vi ha?

- Analys: Vilken funktionalitet behövs i programmet?

Objekt i verkligheten/domänen



Datastrukturer



Vad är viktigt?

Klasser:

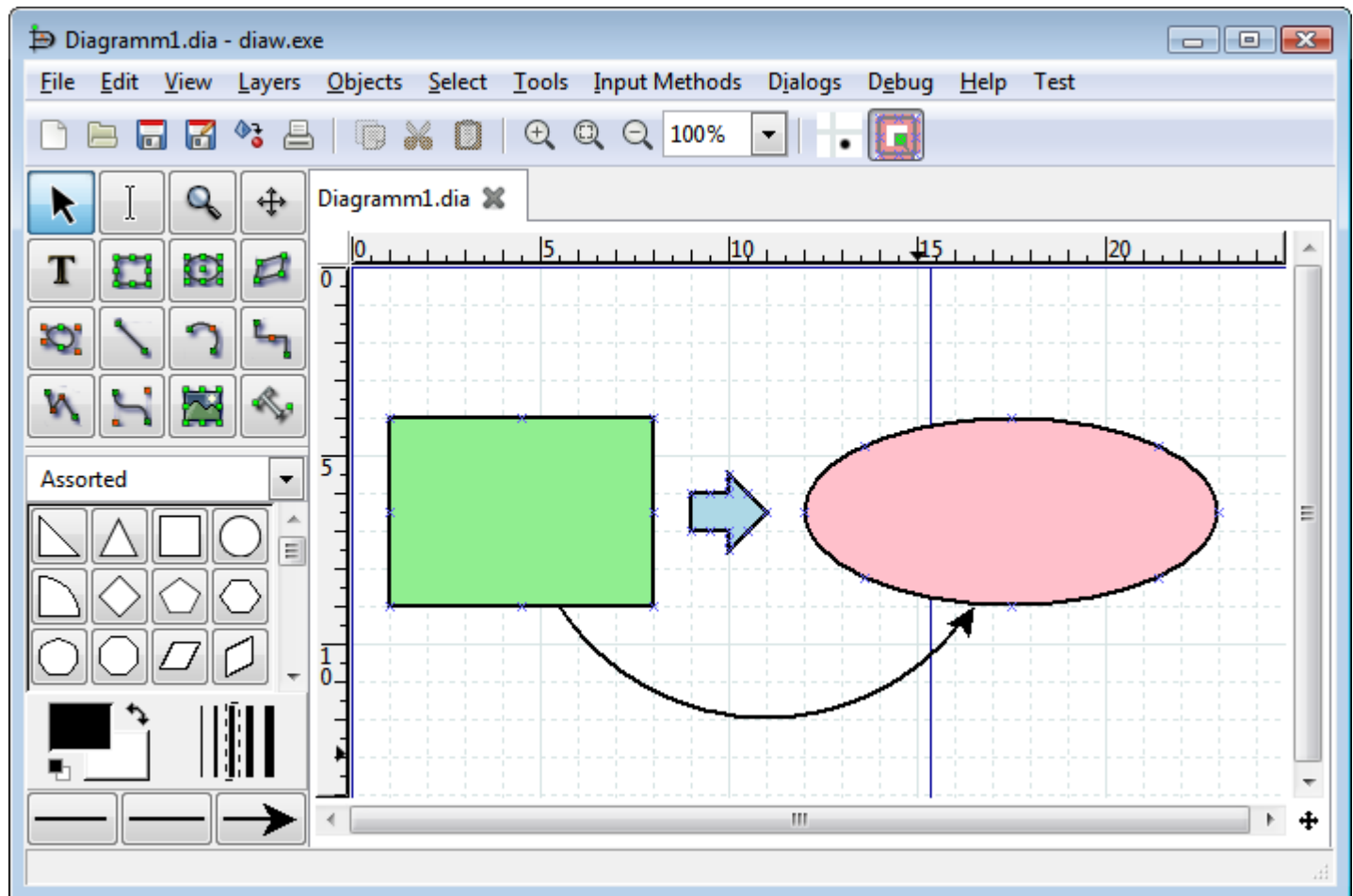
Car, Transistor, Camel
(Vehicle, Component, Animal)

Klasser:

List, Map, File, InputStream
RentalSchedule, ...

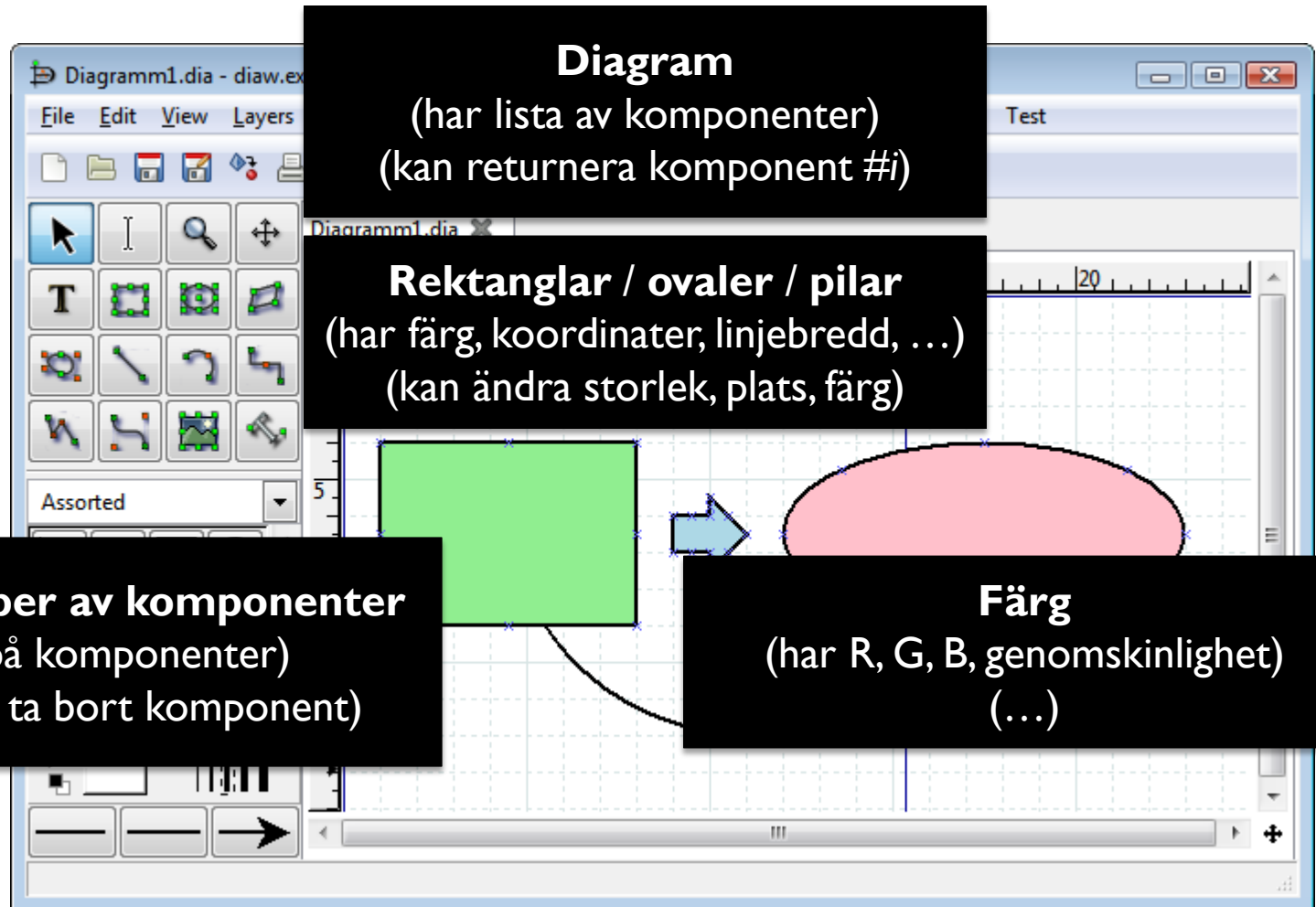
Exempel (1)

- Ett vektorbaserat ritprogram:



Exempel (2)

- I applikationsdomänen vektordiagram:

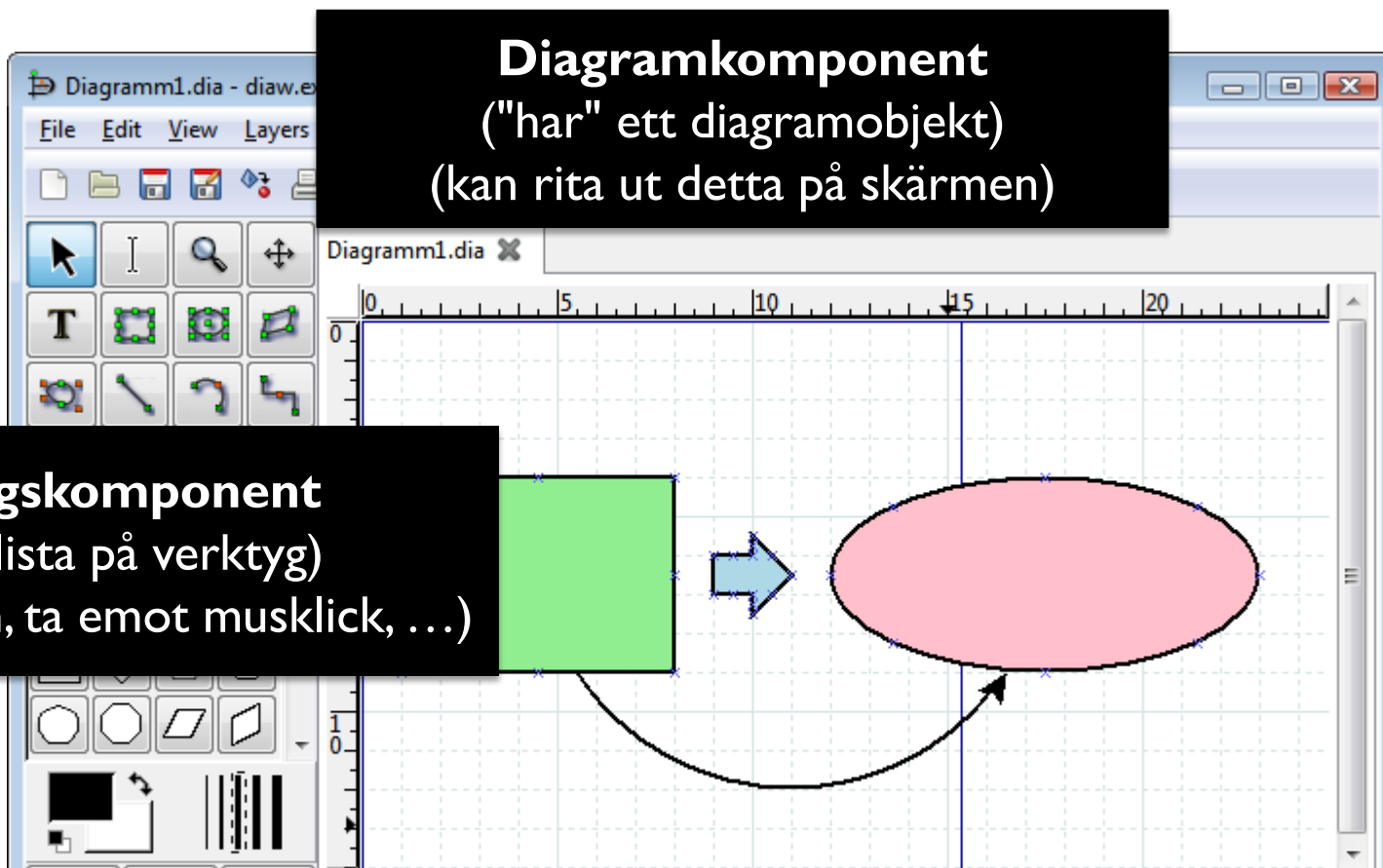


Exempel (3)

- Vi behöver ett grafiskt gränssnitt

Diagramkomponent
("har" ett diagramobjekt)
(kan rita ut detta på skärmen)

Verktygskomponent
(har en lista på verktyg)
(kan rita ut den, ta emot musklick, ...)



...och så vidare

När gör vi uppdelningen?

- **När** kan vi "välja" klasser?

- Vattenfallsmodellen?

1. Kravspecifikation
2. Design → mjukvaruarkitektur
3. Implementation
4. Integration
5. Testning, felsökning
6. Installation
7. Underhåll

- "Kontinuerlig design"?

- Liknande begrepp finns i:
 - Extreme programming
 - Agile software development
 - ...

1. Tänk ut exakt vad som behövs, i detalj
2. Programmera exakt dessa klasser

Tänk lagom mycket!

1. Fundera på övergripande design
2. Fundera mer detaljerat på en delmängd
3. Börja programmera
4. Vid behov: Tänk om, *refactoring*!

Gör så gott du kan från början – men var inte rädd att ändra dig!

- Ett av många sätt att tänka:



Hur vet vi om vi valt rätt uppdelning?

Några principer kan ge ledning...

Bra: Hög cohesion = sammanhållning

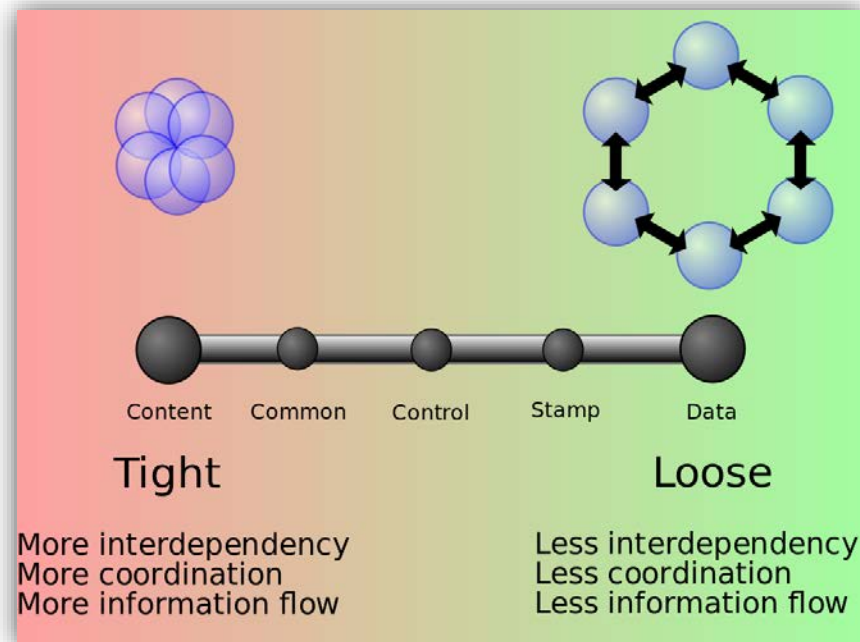
- Klassens funktionalitet ska vara meningsfull och hänga ihop
 - Bör gå att ge en kort sammanfattning av klassen
 - "En List är en sekvens av godtyckliga element"
 - "En LayoutEngine kan göra en specifik typ av layout för ett Diagram"
 - "En MiscWorker genererar slumpal, sparar filer och visar hjälptexter?"
 - ➔ Något är troligen fel...



Men: Undvik splittring, minska koppling

■ Men gå inte för långt!

- Varje extra klass ger ytterligare ett begrepp att hålla reda på
 - `ListStructure`, `ListPrinter`, `ListSorter`, `ListRandomizer`, ... → för mycket!
 - Listfunktionerna är troligen tillräckligt sammanhängande för att vara en klass
- Varje extra koppling (coupling) till en annan klass kan ge problem
 - Täta beroenden mellan klasser → varje ändring påverkar många klasser – dåligt



Låg coupling = få kopplingar mellan klasser

- Låg coupling = färre kopplingar mellan klasser
 - Funktionalitet som hör ihop bör vara samlad
 - (Bra om det räcker att läsa en klass för att förstå hur listor fungerar)
- Som alltid måste man hitta en balans: Ansvarsområden ska vara
 - Sammanhängande
 - Lätta att beskriva
 - Lagom stora



Hör ihop med:

Single Responsibility Principle

1. En klass bör ha ett enda ansvarsområde
 - "Göra en sak" – inte generera slumpstal + spara filer
2. Ansvarsområdet bör hanteras helt inom klassen
 - Inte vara utspritt

Ökar
cohesion

Minskar
coupling

```
class GameComponent extends JComponent {  
    public void paintComponent(...) {  
        player.drawImage(...);  
        if (player.getY()+50 < ball.getY()+32) { ... }  
        ...  
    }  
}
```

50 rårkar vara höjden på spelarfiguren.
Varför ska GUI-komponenten veta detta,
medan spelarobjektet innehåller bilden + koordinater?



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

SRP och sammanhållning för metoder



- **SRP** + önskan om hög sammanhållning gäller även metoder!
 - Varje metod ska ha ett tydligt ansvar, "göra en sak"

```
class MyGameComponent {  
    void tick() { // Called 50 times per second  
        ... A lot of code updating object positions ...  
        ... A lot of code checking for collisions ...  
        ... A lot of code updating scores ...  
    }  
}
```

3 separata uppgifter,
svårt att få översikt

```
class MyGameComponent {  
    void tick() { // Called 50 times per second  
        updatePositions();  
        checkCollisions();  
        updateScores();  
    }  
    void updatePositions() { ... }  
}
```

Lätt att se
vad som händer under tick()

En metod som gör en sak

Modellering och ärvning

Liskov Substitution Principle

Är en kvadrat en sorts rektangel?

- 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 .*
- Förenklat (och till viss del försvagat):
 - En subtyp måste uppfylla supertypens kontrakt

- Anta dessa kontrakt:

Square

Har 4 sidor

Alla 4 sidor är garanterat lika långa

Alla vinklar är 90 grader

Immutable (oföränderlig)

Rectangle

Har 4 sidor

Motstående sidor är lika långa

Alla vinklar är 90 grader

Immutable (oföränderlig)

En kvadrat är en sorts rektangel

Alla krav från rektangelklassen är uppfyllda

Allt vi kan bevisa om rektanglar kan vi också bevisa om kvadrater

OK att mer specifik *immutable* klass ärver från mindre specifik klass

LSP mellan föränderliga objekt

- Anta istället:

Rectangle

Har 4 sidor

Motstående sidor är lika långa

Alla vinklar är 90 grader

Har setWidth(), setHeight()

Square extends Rectangle:

Måste också ha setWidth(), setHeight()

Ärv set*() från Rectangle?

```
square.setHeight(200);  
square.setWidth(100);  
square.getHeight() != square.getWidth()
```

Bryter mot Square:s kontrakt!

Låt set*() ändra höjd och bredd?

```
square.setHeight(200);  
square.setWidth(100);  
square.getHeight() → 100
```

Bryter mot Rectangle:s kontrakt!

Inte OK att mindre flexibel klass ärver från mer flexibel klass

Kodlukt

■ Kodlukt / code smell:

- Något man kan se "på ytan" på koden, som ofta indikerar att det finns ett djupare problem
- Inte nödvändigtvis en bugg i sig, men ska ändå åtgärdas
- Ofta genom refactoring: Kod skrivs om, men gör exakt samma sak



Redundans och upprepning

- 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.*



- 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
 - Leder till mer kod att underhålla
 - Leder till 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...



Redundans på klassnivån (1)

```
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;  
    }  
}
```

Tidigare exempel:
För mycket kod
upprepas!

```
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;  
    }  
}
```

Inträffar ofta i projekt...

Redundans på klassnivån (2)

```
abstract 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;  
    }  
}
```

Tidigare diskuterad lösning:
Flytta kod till gemensam superklass
(abstrakt eller konkret!)

```
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) { ... }  
}
```

```
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) { ... }  
}
```

Redundans på klassnivå (3)

```
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];  
    ...  
}
```

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

Flytta till superklassen!

Redundans på klassnivå (4)

```
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;  
    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;  
    ...  
}
```

Onödiga klasser

Onödiga klasser (1)

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

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

Här ett extremt exempel:
Bara konstruktorn skiljer sig

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

Detta anger inte beteende,
bara *data* (fältvärden)

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

Dålig lösning:
Onödiga klasser

Kan bara anropa
new GenericPlayer(...)
med parameter istället
→ slipper två klasser

Onödiga klasser (1): Fabriksmetoder



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

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

```
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 data!
Ska inte ha olika klasser

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

Onödiga klasser (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

Onödigt många metoder

- Även på andra sätt ska kod skrivas effektivt – möjliga 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

Upprepning av kodmönster (1)

```
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);  
    }  
    ...  
}
```



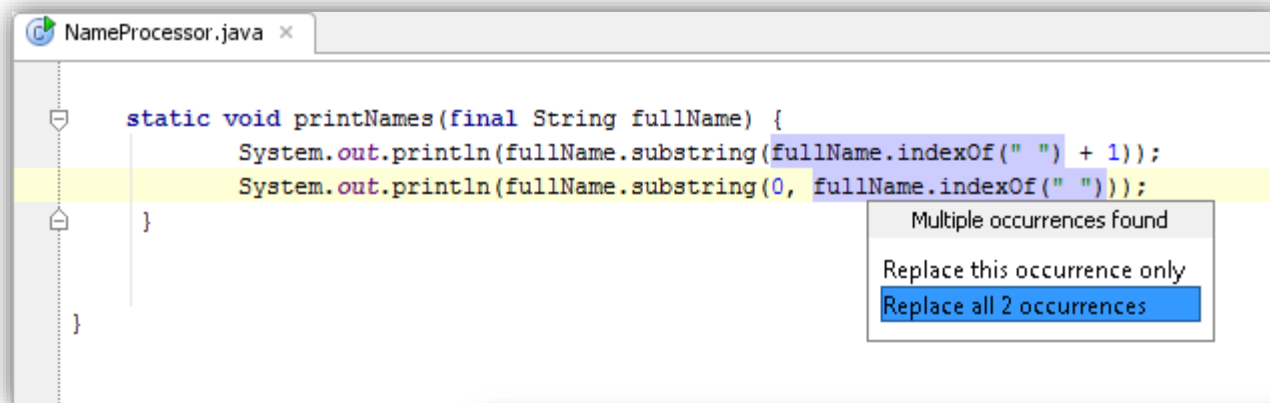
Extrahera en hjälpmetod!

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

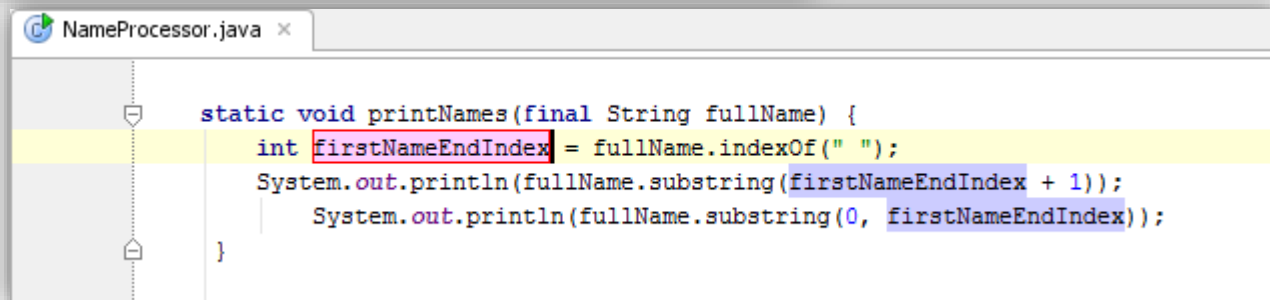
```
private void checkKey(Key key, Direction dir) {  
    if (isPressed(key)) {  
        player.directions.add(dir);  
    } else {  
        player.directions.remove(dir);  
    }  
}
```

Upprepning av kodmönster (2)

- Extrahera upprepade uttryck till namngivna variabler



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



```
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

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

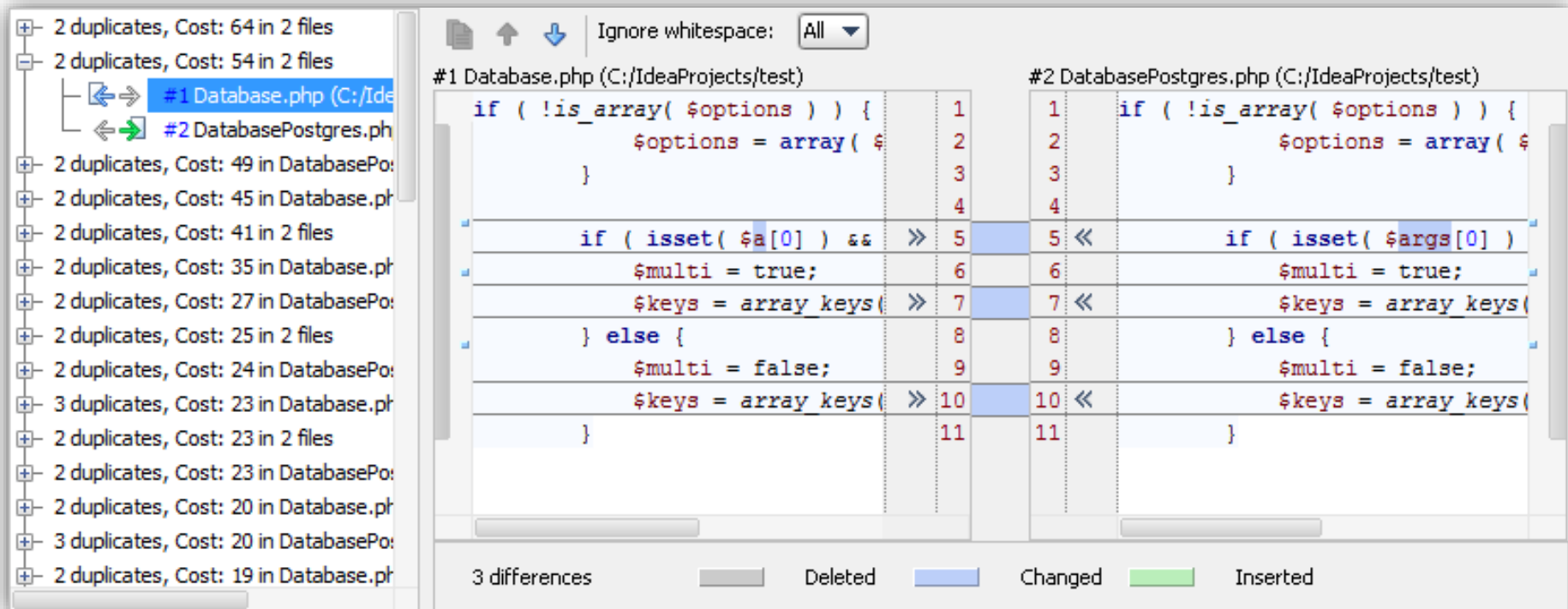
```
class Circle {  
    double x, y, r;  
    Circle(double x, double y, double r) {  
        this.x = x; this.y = y; this.r = r;  
        // Lots of common initialization code  
    }  
    Circle(double r) {  
        this(0.0, 0.0, r);  
    }  
}
```

Specialsyntax:

this(args)

som *första* sats i en konstruktor
anropar en annan konstruktor

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



Ni gör resten – alltför ineffektiv kod ger komplettering!

Redundant lagring

Redundans i lagring (fält)

```
class GenericPlayer {  
    // Can't move left and right at the same time,  
    // so some values are forbidden  
    // → we could get an inconsistency  
    // if we update incorrectly  
    boolean movingLeft;  
    boolean movingRight;  
}
```



```
enum HorizontalDir {  
    LEFT, STILL, RIGHT;  
}  
  
class GenericPlayer {  
    HorizontalDir horizMove;  
}
```

Alla värden rimliga

→ vi har gjort en
feltyp omöjlig

Redundans i lagring (fält)

```
class GenericPlayer {  
    // Can't move left and right at the same time,  
    // so some values are forbidden  
    // → we could get an inconsistency  
    // if we update incorrectly  
    boolean movingLeft;  
    boolean movingRight;  
}
```



```
enum HorizontalDir {  
    LEFT, STILL, RIGHT;  
}  
  
class GenericPlayer {  
    HorizontalDir horizMove;  
}
```

Alla värden rimliga

→ vi har gjort en
feltyp omöjlig

Globala tillstånd

- **Lokalt tillstånd:**

- Information lagrat i vanliga objekt / lokala variabler

- **Globalt tillstånd:**

- Information lagrat i static-variabler (i Java)
 - Andra språk har andra former av globala variabler
- Har bara en kopia

Varning: Används med måtta!

Exempel 1: Poänglista i Tetris

- Labb 4, Tetris:
 - Alla som spelar parallellt ska ha samma highscore-lista
 - Ska aldrig ändras:
Vi kommer aldrig att vilja dela upp spelare i grupper med olika listor
 - ➔ Globalt tillstånd *kan* vara OK, t.ex. en *singleton*:

```
public class HighscoreList {  
    // Skapa bara objekt av denna typ EN gång, när klassen laddas.  
    private static final HighscoreList INSTANCE = new HighscoreList();  
  
    // Privat konstruktor, så ingen annan kan skapa fler objekt.  
    private HighscoreList() { ... }  
  
    // Låt andra komma åt det unika objektet  
    public static HighscoreList getInstance() {  
        return INSTANCE;  
    }  
  
    public void addScore(...) { ... }  
}
```

Men vem som helst kan
komma åt listan
utan att ha "fått" den
➔
Kan bli svårare att förstå
programmets struktur

Exempel 2: Spelbräde i Tetris

- Labb 4, Tetris:
 - Just nu spelas bara ett spel i taget
 - Frestande att låta Board vara en singleton
 - Alla kan direkt komma åt den, behöver inte skickas som metodparameter
 - Senare: Vill kunna spela flera spel parallellt, över nätet
 - Varje spel ska ha sitt eget Board
 - Måste göra om koden
 - ➔ Globalt tillstånd gör förändringar svårare!



Namngivningsprinciper

Namngivning 1: Varför är det viktigt?

Att använda namn, och ge beskrivande namn,
ersätter till viss del
kommentarer och dokumentation!

Lättare och
snabbare
att skriva

Lättare och
snabbare
att läsa

Lättare att hålla
uppdaterat – man
ser om namnet är
"ur synk"

Använd namn!

if (state == 14)



if (state == State.RUNNING)

paint(xpos + 42, ...)



paint(xpos + block.getSize(), ...)

paint(xpos + 41, ...)



paint(xpos + block.getSize() - 1, ...)

Förbättrar läsbarhet, förenklar ändringar

IDEA kan ibland hjälpa till: Refactor | Extract | Constant

Använd namn!

```
if (state == 14) {  
    jump();  
}
```



```
if (state == State.RUNNING) {  
    jump();  
}
```

Var är det enklast att se felet?

Använd namn!

```
class MyGameComponent {  
    void tick() {  
        // Updating object pos  
        ...  
        // Checking for collisions  
        ...  
        // Updating scores  
        ...  
    }  
}
```



```
class MyGameComponent {  
    void tick() {  
        updatePositions();  
        checkCollisions();  
        updateScores();  
    }  
    void updatePositions() {  
        ...  
    }  
}
```

Förbättrar läsbarhet om segmenten är "komplexa/långa" (balans!)

IDEA kan ibland hjälpa till: Refactor | Extract | Method

Använd namn!

**Magiska konstanter luktar illa:
Påverkar läsbarhet, försvårar underhåll → *komplettering***

```
if (jumpCounter < 50) ...  
if (wobbleValue + xCoord < 500) ...
```

```
if (changePicture == 0) {  
    if (i == 4) {  
        i = 0;  
    }  
    changePicture = 3;  
    this.setImage(imageList.get(i));  
    i++;  
}
```

Namngivning 6: Information



- Använd **informativa** namn!
 - En metod som flyttar alla fiender ett steg framåt:
 - `update()` // Moves all enemies
→ `moveEnemies()`
 - En lista som innehåller alla uppkopplingar i en FTP-klient:
 - `c` // Contains all current connections
→ `connections, currentConnections`
 - En boolean-variabel som är sann om spelet är igång:
 - `if (startGame)` → `if (gameRunning)`

Namngivning 7: Namngivningsstandard



- (Och följ Javas namngivningsstandard...)
 - **Paket:** `se.liu.ida.jonkv`
 - **Klasser:** `Raster`, `ImageSprite`
 - **Gränssnitt:** `RasterDelegate`, `Storing`
 - **Metoder:** `run()`, `runFast()`, `getBackground()`
 - **Variabler:** `i`, `myWidth`
 - **Konstanter:** `static final int MIN_WIDTH, MAX_WIDTH`

Kommentarer

- Kommentarer och dokumentation ska finnas
 - Perspektiv: Annan programmerare ska ändra i programmet efter ett par år
 - ”En annan – till exempel du själv”!
 - Översikt över programmets design
 - Översikt över varje klass (Javadoc-format)
 - Kommentera inte det som man *omedelbart inser* när man läser koden
 - Kommentera metoder, fält, algoritmer, designval som inte är uppenbara eller triviala
 - Men försök först att göra dem uppenbara!
 - Förklara varför något görs
 - Inte bara vad som görs

(Men vi ser oftare
projekt som *saknar* dokumentation
än projekt med *för mycket* dokumentation)

Organisation via namnrymder (Paket i Java)

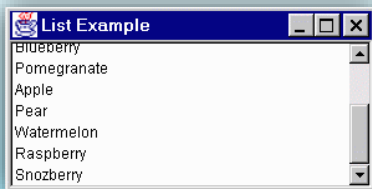
■ Enkla klassnamn kan ge namnkollisioner

- Datastrukturen `List` och GUI-elementet `List`?
 - Kan döpa om to `ListStructure`, `ListComponent`...
 - ➔ Måste alltid använda långa namn, även om vi inte jobbar med GUI
- Din liststruktur och någon annans?
 - Döp om till `JonasListStructure`, ...?

Många språk har distinkta namnrymder

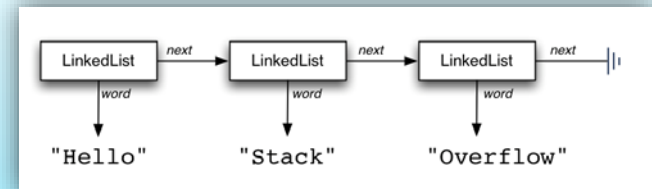
“GUI-namnrymd”

List ➔



“Datatyps-namnrymd”

List ➔



Namnrymder ger oss enklare namngivning, men också organisation

Repetition: Namnrymder i Python

■ Från Python:

```
>>> import newton
```

This program tries to find the square root of a number.

Enter a number: **2**

1.5

1.4166666666666665

1.4142156862745097

1.4142135623746899

1.414213562373095

```
>>> newton.find_root()
```

This program tries to find the square root of a number.

Enter a number: **10**

3.5

3.178571428571429

3.162319422150883

3.1622776604441363

3.162277660168379

**Gör "newton" till en namnrymd
(med innehållet från filen newton.py)**

**Fullständigt namn:
namnrymd.funktion**

**Ger möjlighet till t.ex.
"gui.List" och "datatypes.List"**

Paket 1: Namnrymder



- I Java ingår **varje klass** i ett **paket** (som är en namnrymd)

- **package** se.liu.ida.jonkv.graphics.shapes;
class Circle {
 ...
}

■ se.liu.ida.jonkv.graphics.shapes.Circle c1;

Standard för unika namn:
Börja med domännamn,
baklänges

Python

Om vi gör "import"
skapar vi en namnrymd,
där filens innehåll placeras

Kod i fil A hittar bara kod i fil B
om den "importeras"

Java

Klassen deklarerar
vilken namnrymd den tillhör
– detta gäller alltid!

Skriv paketnamn.klass
så hittas klassen utan "import"

Hur hittar Java klassen?

- Javas källkod lagras i en katalogstruktur motsvarande paketen
 - rotkatalog ~/mysource
 - katalog se
 - katalog liu
 - katalog ida
 - katalog jonkv
 - katalog graphics
 - katalog shapes
 - fil Circle.java
 - fil GraphicCircle.java
- Ange rotkatalogen ~/mysource – så hittas allt annat från den
 - På kommandoraden: Miljövariabeln **CLASSPATH**
 - I övrigt: Beror på utvecklingsmiljön

Kan bli många nivåer

**Utvecklingsmiljöer
hjälper till att
hålla reda på detta...**

Paket 3: Existerande

- Många paket finns i Javas klassbibliotek

java.lang:

Object, Class, String,
Thread, ...

java.util:

List, ArrayList, Date,
Calendar, Timer,
Formatter, ...

java.io:

Reader, InputStream,
...

Paket 4: Import förkortar namnen



- Men nu blir det väl lite jobbigt...
 - `javax.swing.filechooser.FileFilter` filter =
`new javax.swing.filechooser.FileFilter();`

- Undvik med **Javas** variant av **import**

- **package** se.liu.ida.jonkv.io;
import javax.swing.filechooser.FileFilter;

```
class FileSaver {  
    ...  
    FileFilter filter = new FileFilter();  
    ...  
}
```

Talar bara om var FileFilter finns!

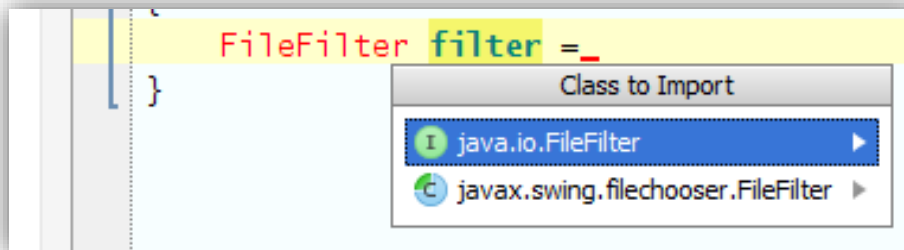
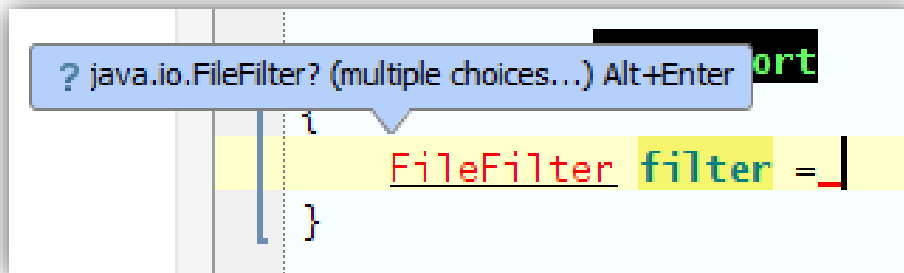
Om jag säger FileFilter
menar jag
`javax.swing.filechooser.FileFilter`

Ungefärlig motsvarighet i Python:

```
from javax.swing.filechooser  
import FileFilter
```

Paket 5: Automatisk import

- Vissa klasser importeras automatiskt
 - Alla klasser i samma paket
 - Alla klasser i java.lang ("systemklasser" som *Object*, *String*)
- Moderna utvecklingsmiljöer hjälper till med annan import



Paket 6: Import med "*"

- Kan importera alla klasser i ett paket med "*"
 - **import** javax.swing.filechooser.*;
FileFilter filter = **new** FileFilter();

Motsvarighet i Python:

```
from javax.swing.filechooser
import *
```

Undvik: Kan ge problem
om nya klasser adderas till ett importerat paket...

Skriv i Java 1.0...

```
import java.awt.*; // Har en List-klass
import java.util.*; // Ingen List-klass...
class MyClass {
    List list; // Unikt!
}
```

Kompilera med Java 7...

```
import java.awt.*; // Har en List-klass
import java.util.*; // Har också en...
class MyClass {
    List list; // ??? Kompileringsfel!
}
```

- Om inget paket anges:
 - Klassen hamnar i "unnamed package"
 - **Använd inte detta** – ange alltid paketnamn!

- För att:
 - Undvika namnkollisioner
 - Förbättra läsbarhet, underhåll, organisation:
Vilka klasser är relaterade?
 - Undvika problem med verktyg som antar att paketnamn finns