

# ex1classes

Joachim von Hacht

# Klassfiler

```
class Player {  
    String name;  
    int points;  
}  
  
String getName(){  
    return name;  
}
```

Player.java

2

I standard Java-program deklareraras klasser i egna filer, vanligen med en klass per fil.

- Klassens namn och filens namn måste vara samma (förutom att filen avslutas med .java).
- Lite förenklat.

# Åtkomst

```
public class Dice {    // In file Dice.java
    // Private, inaccessible from other classes
    private Random rand = new Random();
    private int nFaces;

    public Dice(int nFaces) {
        this.nFaces = nFaces;
    }
    // Public, for other objects to call
    public int roll() {
        return rand.nextInt(nFaces) + 1;
    }
}
```

3

Då man lägger en klass i en egen fil kan man specificera åtkomst (access) från andra klasser (filer).

- Man anger för åtkomst för klassen (skrivs framför class)
  - Vi anger alltid public (man kan komma åt klassen överallt i programmet)
- I klassfilen anger man dessutom åtkomst för alla instansvariabler
  - **public**, innebär att alla kan komma åt variabeln, variabeln är tillgänglig i all kod utanför klassen (om klassen är public)
  - **private**, ingen kod utanför klassen kan komma åt variabeln
    - Vi sätter normalt alltid private på alla instansvariabler
    - Genom att använda private skapar vi ett lokalt tillstånd i klassen
    - Vid felsökning behöver vi bara söka i klassen (om det inte handlar om referenser, ... vilket det tyvärr ofta gör)
  - **protected**, använder vi troligen inte (innebär att subklasser kan komma åt, se senare).
- Kontroll av åtkomst sker redan vid kompileringen, försöker vi använda private-variabler utanför klassen får vi ett kompileringsfel

Åtkomst för metoder anges på samma sätt som för variabler

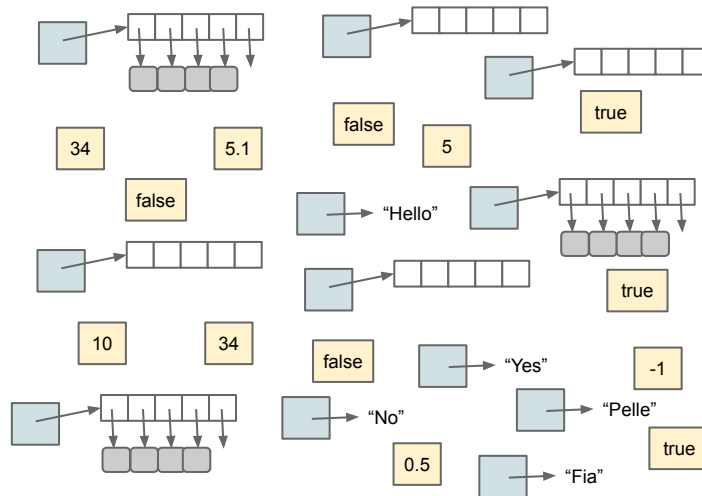
- public-metoder kan användas överallt i koden (om klassen är public)
- private-metoder kan bara användas inom klassen

- Används för interna hjälpmetoder (funktionell nedbrytning)
- En markering att metoden inte används någon annanstans.
  - Vid felsökning behöver vi bara söka i klassen.
- protected , som ovan.

OBS! Om vi befinner oss i samma fil (med flera klasser) så spelar åtkomst ingen roll, vi kan alltid komma åt allt i samma fil.

- Åtkomst gäller mellan klasser i olika filer

# Tillstånd



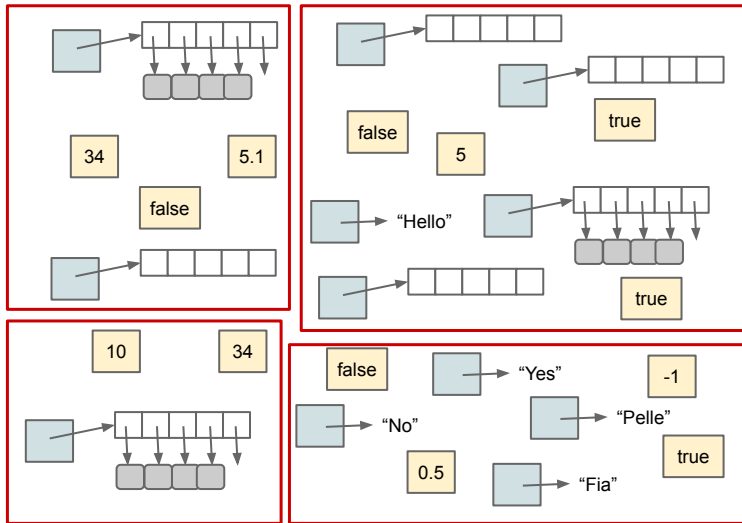
4

Tillstånd (state) mängden av alla värden för alla instansvariabler i programmet vid en viss tidpunkt under exekveringen

- Lokala variabler räknas ej (de kommer och går ...)
- Om allt fungera som tänkt innehåller variablerna korrekta värden, programmet befinner sig i ett giltigt tillstånd ... om EJ
- ... har vi ett ogiltigt tillstånd. Något är fel
- Att naivt försöka behärska tillståndet övergår mänsklig förmåga ...
  - ... vi måste utveckla tekniker för detta
  - Innebär t.ex. att vi alltid föredrar lokala variabler (eftersom de inte ingår i tillståndet)
  - Vi försöker också konsekvent att minska synlighetsområdet för variabler.

Ett fundamentalt problem inom imperativ programmering är att behärska tillståndet!

# Varför Åtkomstspecifikation?

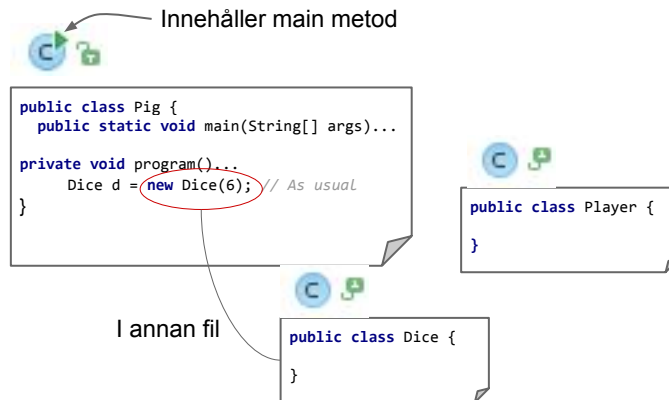


5

Ett sätt att försöka behärska tillståndet är att dela upp det totala tillståndet i mindre deltillstånd och på så sätt lättare kunna hålla dessa giltiga

- Kallas (bl.a.) **informationsgömning** ([information hiding](#))
- Konkret gör vi detta genom att dela upp tillståndet på olika klasser med privata instansvariabler.

# Exekvering med Klassfiler



6

Om man har ett Java program uppbyggt av ett antal klassfiler måste en av dessa innehålla metoden main

- Innan körning måste alla filer kompileras
  - Sköts av IntelliJ
- Instansiering påverkas inte av att klasser ligger i separata filer (så länge som klassen och konstruktorn är public)

# Get och Set Metoder

```
public class Player {  
  
    private String name;  
    private int points = 0;  
    ...  
    public String getName() { // Getter, NOTE name  
        return name;  
    }  
    public void setName(String name) { // Setter, NOTE name  
        this.name = name;  
    }  
  
}
```

7

Eftersom vi sätter alla instansvariabler till private kan ingen kod utanför klassen komma åt dem  
Leder till vissa problem...

Ibland måste vi läsa av tillståndet t.ex. vid utskrifter

- Att läsa av tillståndet är inte så riskabelt (inget skall ändras)
- För avläsning skapas get-metoder (getters).
  - Standard för Java är att de heter get + namnet på instansvariabeln (se bild)

Ibland måste vi kunna ändra tillståndet

- Ändring är mycket farligt, kan leda till ogiltigt tillstånd
- Vår strategi för ändring av tillstånd
  - Om möjligt sätt värden i konstruktorn
  - Om möjligt skapa metoder som gör förändringar internt i klassen t.ex. om en spelares poäng skall öka låt objektet sköta detta (inte läsa av, öka, och skriva tillbaks)
    - Om det verkligen behövs skapa en set-metod.
    - Heter alltid set + namnet på instansvariabeln

Generellt: Låt objektet som har datan, gör beräkningarna och skicka ut resultatet, istället för att skicka ut datan! Låt objekten ha sin data i fred!



# Konstruktoröverlagring

```
class Complex {  
  
    // ---- Overloaded Constructors -----  
    public Complex(double re, double img) { ... }  
  
    public Complex(Complex other) { ... }  
  
    public Complex(double re) { ... }  
  
}
```

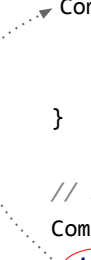
8

Konstruktorer kan överlagras på samma sätt som metoder.

- Ofta vill man initiera ett objekt på flera olika sätt
  - Vissa värden kanske skall vara förvalda etc.
  - En klass kan ha flera konstruktörer för detta ändamål
- Som tidigare vid överlagring så måste parametrarna skilja sig åt

# this(...)

```
class Complex {  
    ...  
    Complex(double re, double img) { // Constructor  
        this.re = re;  
        this.img = img;  
    }  
  
    // Same result as previous slide  
    Complex(Complex other) {  
        this(other.re, other.img); // Call other constructor  
    }  
    ...  
}
```



9

När man överlagras konstruktorer skapar man en baskonstruktor som verkligen sätter alla värden. Därefter skapar man konstruktorer med förvalda värden för vissa parametrar (av bekvämlighetsskäl)

- Konstruktorer med förvalda värden använder this() för att anropa andra konstruktorer (med fler parametrar)
- Genom att skriva this(...) , d.v.s. parenteser efter this, så avses någon konstruktor (med matchande parameterlista).

# Klassen Object

```
class MyOwnClass{
    // No methods here!!!
}

MyOwnClass m = new MyOwnClass();

// No methods in class but still able to call?!
out.println(m.toString());
out.println(m.getClass());
out.println(m.equals("abc"));
out.println(m.equals(5));
out.println(m.hashCode());
// Etc.
```

11

Om man tänker till så finns det metoder som antagligen alla objekt kan tänkas behöva t.ex. jämförelse.

- Istället för att alla alltid skall skriva dessa "generella" metoder i sina klasser har man i Java samlat dessa i klassen Object.
- Därefter låter man implicit alla klasser "ärva" metoderna från Object (inget syns i koden).
  - D.v.s. vilken klass vi än skapar så har den metoderna som finns i Object.
  - Detta är förklaringen till att Object är supertyp till alla referenstyper. Vi kan tilldela en Object variabel vilket objekt som helst eftersom alla kan utföra operationerna i typen Object (de ärver ju metoderna!)

Object innehåller bl.a. metoder.

- equals används då man vill jämföra objekt
- hashCode, används av samlingar t.ex. Map, se Samlingar
- getClass kan svara på vilken klass (typ) ett objekt har tillhör.
- toString är tänkt att ge en läsbar representation av ett objekt (ett objekt som en sträng)

Som synes ärver vi metoden toString.

- Tidigare lade vi till toString-metoden i en klass för att få en

- strängrepresentation av ett objekt (läsbar utskrift).
- Det som händer är då att vår version av metoden kommer att köras istället för den ärvda.
- För att det skall fungera måste vår metod ha exakt samma metodhuvud som den ärvda.
- Kallas överskuggning (override).

# Likhet för Objekt

```
public class Player {  
    ...  
    @Override  
    public boolean equals(Object other) {  
        // Same object (i.e. identity)?  
        if (this == other) { return true; }  
        // Only same types of objects allowed  
        if (other == null || getClass() != other.getClass()) {  
            return false;  
        }  
        Player = (Player) other;  
        // This is our definition of equals (others possible)  
        return this.points == other.points;  
    }  
}
```

11

Alla klasser ärver en metod equals() från klassen Object.

- Metoden ger referenslikhet.

Vill vi ha värdelikhet för objekt måste vi själva definiera vad vi menar med likhet

- När vi bestämt oss skapar vi en egen version av metod equals i klassen (på samma sätt som med toString).
  - Eftersom vi har skapat en egen equals() (överskuggat den ärvda) skriver @Override över, kompilatorn kontrollerar då att metodhuvudena är identiska (förutom namn på parametrar)
- I bilden har vi bestämt att två spelarobjekt är lika då de har lika poäng ( ... kanske inte så bra?)
- Som tidigare så kommer vår metod, inte den ärvda, att användas för alla Player-objekt

Om man skapar en egen equals-metod skall man alltid skapa en egen hashCode-metod.

- Hur detta görs går vi inte in på (normalt given i laborationer, övningar)
- Båda metoderna kan genereras av IntelliJ. Högerklicka > Generate > ...
- toString kan också genereras.

# Equals och Samlingar

```
List<Complex> list = new ArrayList<>();  
list.add(new Complex(5, -2));  
  
// Must have equals() else not found (by value)  
out.println(list.contains(new Complex(5, -2)));
```

12

Om man skall lagra objekten i någon samling måste man implementera equals och hashCode. Annars hittas inte objektet (värdelikhhet)

# Icke-Muterbara Objekt

```
public class Complex {  
  
    private final double re;    // Immutable (final)!  
    private final double img;  
  
    // Return a new Complex object (because we normally  
    // assume operands won't change, also in this case class immutable)  
    public Complex add(Complex other) {  
        return new Complex(this.re + other.re, this.img + other.img);  
    }  
}
```

Ett radikalt sätt att hantera tillståndet är att göra så att man inte kan förändra ett objekt tillstånd efter det att det instansieras.

- I klassen sätts alla instansvariabler till final
  - OBS! Om vi har instansvariabler av referenstyp inte säkert detta räcker!
- Initiering görs i konstruktorn (det är tillåtet att sätta final variabler i konstruktorn)
- Vi säger att objekten som skapas är **icke-muterbara**
- Icke-muterbara objekt är säkra att jobba med, inga alias-problem eftersom tillståndet inte kan ändras!

Vissa objekt uppfattas naturligt som icke-muterbara t.ex. operander

- Vi förväntar oss inte att operanderna i uttrycket  $a + b$  ändras, ...
- ... vi förväntar oss ett nytt värde,  $c$ , skilt från både  $a$  och  $b$
- .. vi skall inte kunna ändra  $a$  eller  $b$  och på så sätt ändra resultatet  $c$ .

Nackdelen med icke-muterbara objekt är att vi måste skapa nya objekt om vi vill ha ett annat tillstånd

- Kan bli kostsamt om vi har väldigt många (små) objekt

Det ovan lite förenklat, mer i senare kurser.

Om en klass representerar ett värde som man utför operationer på skapar man normal nya objekt för resultatet från operationerna.

- Någon har referenser till operanderna, vill inte att dessa skall ändras (helst använda icke-muterbara objekt).



# Ansvar



- En klass skall fånga ett koncept, ha ett ansvar.
- På samma sätt som metoder.