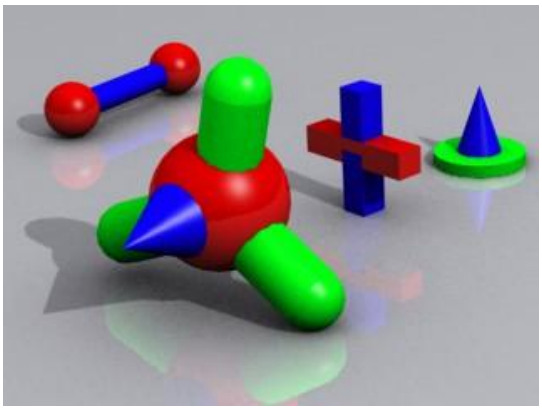
Klasser

TDA548/Joachim von Hacht

Flera men Olika



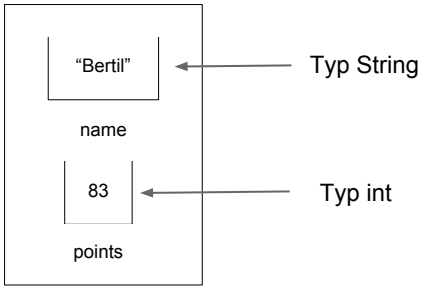
2

Arrayer används när vi behöver många variabler av samma typ.

Ibland behöver vi flera variabler men ev. av olika typ (vissa kan vara av samma typ)

- Om vi t.ex. vill beskriva en spelare i ett spelprogram så kanske vi behöver en variabel för spelarnamn och en för poäng
  - Krångligt t.ex. vid metodanrop att behöva skicka många variabler
- Istället för att ha enskilda variabler kan vi skapa ett **objekt** med flera variabler
  - Vi kan då skicka objektet, d.v.s. “alla” variablerna, på en gång till en metod.
- Dessutom hör ju de enskilda variablerna konceptuellt ihop, de används för att beskriva en spelare
  - Programmet blir lättare att förstå

Konceptuell bild



Objekt med två variabler av olika typ

3

Ett objekt är en "förpackning" av variabler som hör ihop (förenklat, mer senare)

- I bilden har vi samlat flera variabler som hör ihop med en spelare.

Vi kan inte skapa objekt direkt

- För att skapa objekt behöver vi en **klass**

OBS Bilden! Att värdet "Bertil" egentligen inte ligger i variabeln (Se Strängar).

Klassdeklaration

```
// Very basic class
class Player {
   String name;  // Instance variables
   int points;
}
```

4

För att skapa en klass i Java gör man en klassdeklaration

- Vi kan lägga klassdeklarationer var som helst i programmet (utanför alla metoder).
- Vi lägger dem vanligen mot slutet i programmet (efter metoderna).
    - Eller i egna filer, se senare.

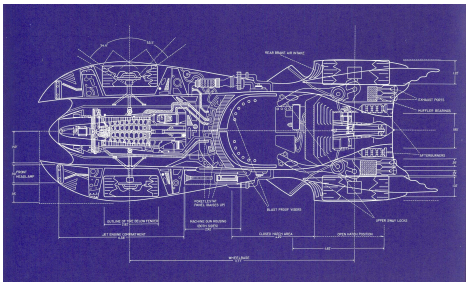
Klassdeklaration

- Inleds med det reserverade ordet **class** + namnet på klassen.
    - Namn inleds med stor bokstav (CamelCase vid behov)
    - Namnet brukar vara ett substantiv och skall förklara vad objekten skall representera.
- Därefter ett block
- I blocket deklarerar vi de variabler vi vill skall finnas i objekten, dessa kallas **instansvariabler** (alt. **attribut**)

Alla instansvariabler kommer att initieras med förvalda värden, t.ex. 0 för int.

Klass och Instans

Klass



Objekt (Instanser)

5

Vi kan skapa ett godtyckligt (ändligt) antal objekt utifrån en och samma klass

- Man kan se en klass som en ritning
- Vi säger att ett objekt är en **instans** av en klass
- Alla instanser får sin egen uppsättning av variabler
    - Vi kanske behöver två spelarobjekt, ett för “Pelle” och ett för “Fia”, båda innehåller egna variabler name och points.

Analogier:

- Pannkaksrecept (klass) och pannkaka (objekt)
- Pepparkaksform (klass) och pepparkaka (objekt)

Variabler för Objekt*)

```
// Declare, instantiate and initialize
Player p = new Player();
```

Ny typ!

Metod med samma namn som klass

```
Player p;   // Declare ...
...
// ... later instantiate and assign
p = new Player();
```

*) Förenklat kommer mer

6

Alla variabler måste ha en typ!

- En klass introducerar en ny typ
  - Typsystemet är utbyggbart för klasser
  - Om vi skapar en ny klass kan vi deklarera variabler av denna typ!
- När vi deklarerat variabeln kan vi initiera den
  - För att initiera variabeln måste vi instansiera ett objekt.
  - Instansieringen sker genom att använda **new**-operatorn tillsammans med ett metodanrop till en metod som heter som klassen (mer senare)
    - Det är instansieringsuttrycket (till höger om =) som skapar objektet
    - Vi kan inte göra som med arrayer t.ex. { “olle”, 87 } eller liknande går inte.

Analys av kod (vänster till höger)

- Player är den nya typen som klassen introducerat
- p är namnet på variabeln
- = new Player() anropar en metod som heter som klassen och som instansierar ett objekt (den finns alltid). Därefter initieras variabeln p med objektet (mer senare).

Punktnotation

```
// Declare instantiate and initialize
Player p1 = new Player();

// Dot notation to access variables in object
p1.name = "pelle";
p1.points = 2;

out.println( p1.name );  // "pelle"
out.println( p1.points ); // 2
```


Ett namn på en array-variabel betecknade hela arrayen

- För att komma åt enskilda variabler använde vi indexering

Ett namn på en objektvariabel (p1 i bilden) betecknar hela objektet

- För att komma åt de enskilda variablerna används punktnotation d.v.s. objektvariabelnamn . instansvariabelnamn

Arrayer med Objekt

```
// Declare and initialize array AND instantiate
// objects (each new .. creates an object)
Player[] players = { new Player(), new Player() };

// First indexing then dot notation
players[0].name = "pelle";
players[0].points = 23;
players[1].name = "fia";
players[1].points = 45;

out.println( players[0].name );  // “pelle”
out.println( players[1].points ); // 45
```

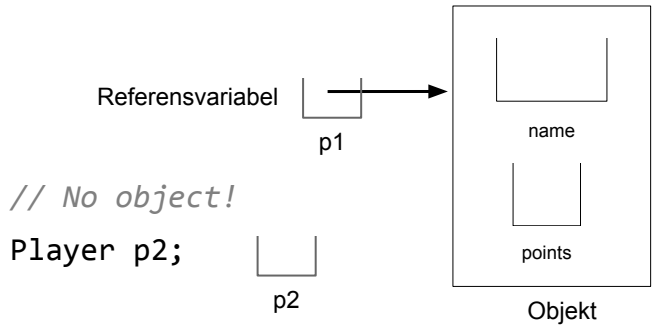
8

Om vi har en typ kan vi skapa arrayer utifrån typen

- Så också med klasstyper!
- För att komma åt enskilda variabler används …
- … först indexering eftersom vi började med en array (ger hela objektet) …
- … därefter punktnotation för att komma åt variabeln i objektet.

Mer om Variabler för Objekt

```
// Reference variable p1 and object
Player p1 = new Player();
```




```
// No object!
Player p2;
```

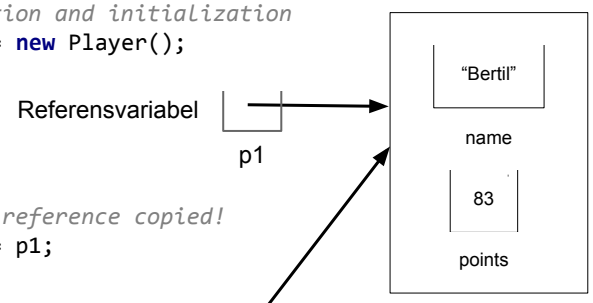

9

Detta fungerar på samma sätt som för arrayer!

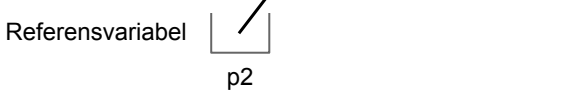
- Variabler för objekt är referensvariabler
  - En deklaration ger en referensvariabel, instansieringen (instansieringsuttrycket) ger en referens till objektet
  - Sker detta samtidigt hamnar referensen i den deklarerade variabeln
  - Värdet av instansieringsuttrycket är en referens, sidoeffekten är att ett objekt skapas.
- En deklaration av en variabel ger en referensvariabel (bara), inget objekt!

Tilldelning

```
// Declaration and initialization
Player p1 = new Player();
```




```
// Assign, reference copied!
Player p2 = p1;
```

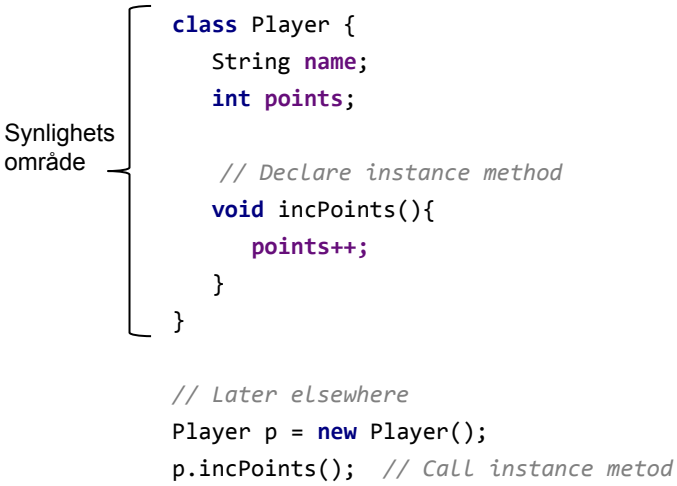



```
out.println(p1 == p2);   // true
```


På samma sätt som för arrayer, det är referensen som kopieras!

- D.v.s vi får två olika variabler som refererar samma objekt
- Ger samma aliasproblematik som för arrayer

Instansmetoder



11

Instansmetoder är metoder som är deklarerade i en klass.

- De kan anropas på alla objekt skapade utifrån klassen (förenklat, mer strax …)
- Det måste finnas ett objekt för att kunna anropa metoderna..

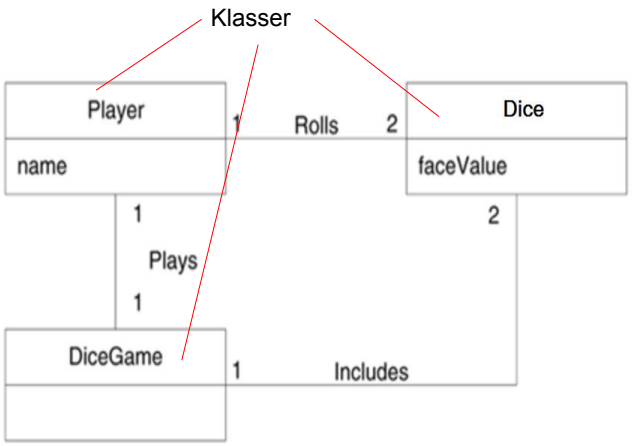
Instansvariabler deklareras som sagt utanför all metoder (vi skriver dem ofta överst i klassen)

Synlighetsområdet för instansvariabler är hela klassen (i alla metoder)

- Ett problem om något blir fel, vilken metod orsakade felet??
- Undvik instansvariabler om det går (ofta behövs de …)

Instansvariabler och lokala variabler kan ha samma namn!

Klasser och Programstruktur



15

12

Att använda klasser i ett program ger många fördelar

- Klasser blir ett nytt, mer övergripande, sätt att strukturera vår program
  - Klasser ger en naturlig enhet att samla data (instansvariabler) och metoder (instans metoder) som använder denna.
  - Metoderna får en naturlig hemvist.
- Vi kan implementera en klass i taget och testa den.
- Klasser har namn, vi får begrepp på högre nivå än metoder (abstraktion!).
  - Klasserna representerar fenomen på ett naturligt sätt, t.ex. representeras en spelare med en spelarklass (spelarobjekt när vi kör) o.s.v.
- Klasser kan ev. återanvändas i andra program

Klassfiler

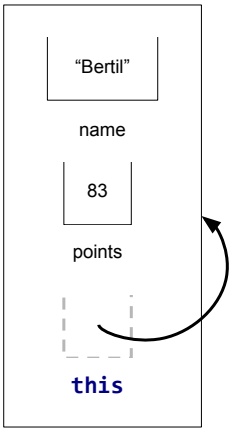
```
class Player {
   String name;
   int points;
}

String getName(){
   return name;
}
```

Player.java

Vanligen deklareras klasser i helt egna filer med en klass per fil.

- Annars blir det för mycket i själva programmet, m.m. (… mer strax)
- Klassens namn och filens namn måste vara samma (förutom att filen avslutas med .java)

this




```
// Implicit "this" used
out.println(name);
out.println(points);

// Using this explicit
// Same output as above
out.println(this.name);
out.println(this.points);

// No! this is constant
this = ...
```

14

Alla objekt i Java har en dold konstant referens till sig själv.

- Referensen används automatiskt (osynligt) då man t.ex. anger en instansvariabel, underförstått är det variabeln för det aktuella objektet som avses.
- Ibland använder man referensen explicit genom att skriv **this** i koden.
  - Används bl.a. vid namnkrockar, mer strax
- this är ett reserverat ord

Konstruktor

```
class Player {
   String name;
   int points;
}
Player p = new Player();
// Tiresome to set values
p.name = "pelle";
p.points = 2;
```

```
// Better, use constructor
class Player {
   String name;
   int points;
   // Constructor to set values
   Player(String name, int points){
      this.name = name;
      this.points = points;
   }
}

// Using constructor with arguments
Player p = new Player("pelle", 2);
```

15

För att initiera ett objekt på ett visst sätt skapar vi en speciell metod, en **konstruktor**

- M.h.a. konstruktorn sätter vi värden för instansvariabler

En konstruktor

- Är en metod som automatiskt anropas i samband med instansiering
  - Måste ha samma namn som klassen.
  - Parameterlista som vanlig metod (ev. tom).
  - Får inte ange returtyp.
  - Kan inte anropas som en vanlig metod
- Ofta är det naturligt att parametrarna till konstruktorn har samma namn som instansvariablerna de skall tilldelas (så att vi slipper hitta på olika namn för samma koncept).
  - För att skilja på parametrarna och instansvariablerna anger vi explicit **this**.

Finns alltid en parameterlös konstruktor (default konstruktor)

- Syns inte i koden.
- När vi skapar egna konstruktorer "försvinner" den förvalda
- Vill vi ha en parameterlös konstruktor får vi skriva dit den, d.v.s. vi kan ha flera konstruktorer, de kan vara överlagrade.

Konstruktoröverlagring

```
class Complex {
    …
    Complex(double re, double img) {  // Constructor
        this.re = re;
        this.img = img;
    }

    Complex(Complex other) {   // Another constructor
        re = other.re;
        img = other.img;
    }
     …
}
```

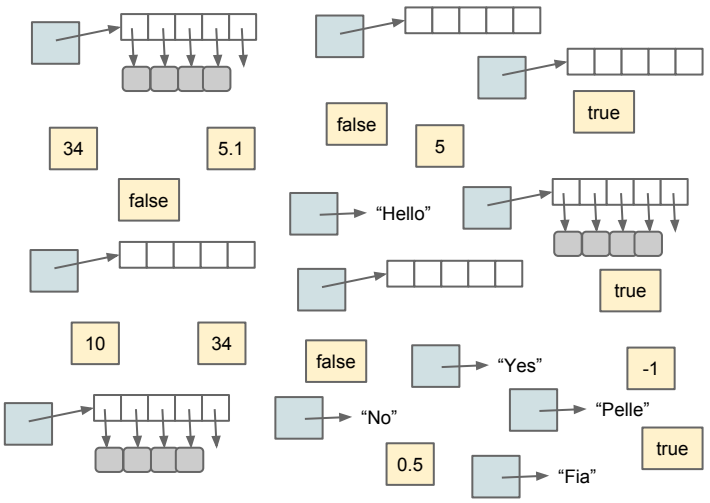
16

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 konstruktorer för detta ändamål
- Som tidigare vid överlagring så måste parametrarna skilja sig åt

IntelliJ kan generera konstruktorer: Högerklicka > Generate …

Mer om Tillstånd



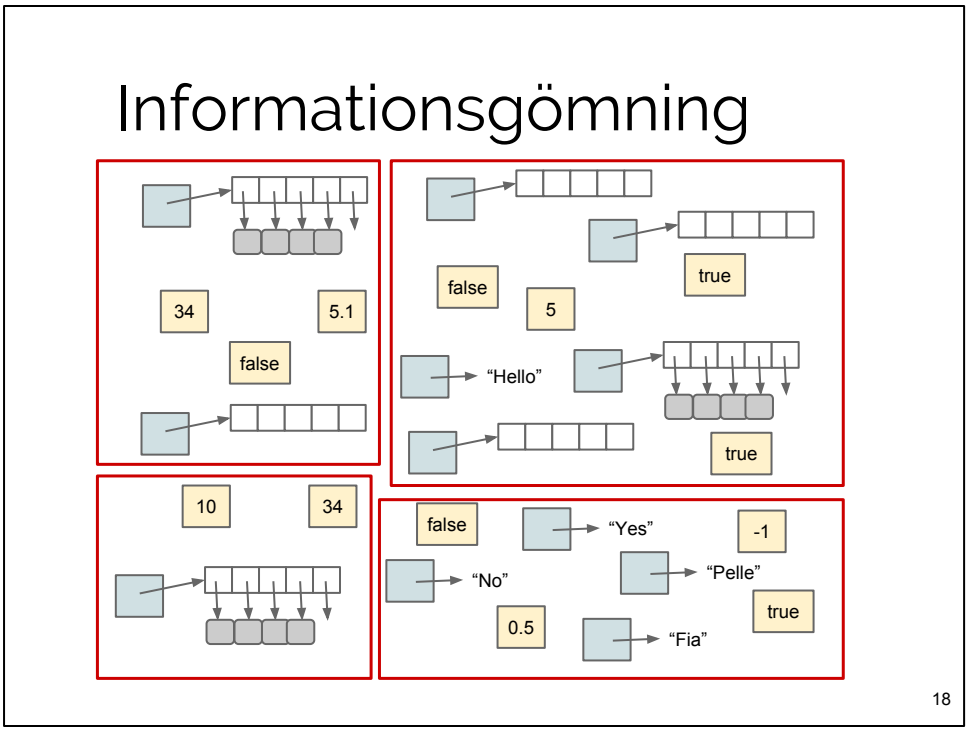
17

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, forts. …
- Om ett objekt ändrar tillstånd kallas det att objektet **muterar (mutate)**

Ett fundamentalt problem inom imperativ programmering är att behärska tillståndet!

- 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 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 **informationsgömning** (information hiding), lite svävande terminologi, … men vi säger så)
  - Vi ser till att instansvariabler (i möjligaste mån) bara används inom en viss del av programmet
  - Övriga delar skall inte komma åt
- Dock ingen garanti … lokalt giltiga tillstånd ger inte automatiskt globalt giltigt tillstånd

Exempel : Antag totala tillståndet skall noll

- Deltillstånd A skall alltid vara negativt och deltillstånd B skall alltid vara positivt
- Även om deltillstånd giltiga så inte säkert totalt tillstånd giltigt

Klassfiler och 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(6) + 1;
   }
}
```

20

I Java kan man åstadkomma informationsgömning genom att dela upp programmet i klassfiler (klassdeklarationer i egna filer)

- Man anger **åtkomst (access)** för klassen (skrivs framför class)
  - Vi anger alltid public class (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)
  - **protected**, mer senare …
  - **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, …)
- 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)
- protected , mer senare …
- 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.

Anges ingen åtkomst alls gäller "package visibility", d.v.s. alla klasser (filer) i samma mapp kan komma åt

- Vi använde inledningsvis detta för att slippa ange åtkomst …
- … i fortsättningen anger vi alltid åtkomst.

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

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;
   }

}
```

22

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).
    - De heter alltid 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 att skicka ut datan!

- Låt objekten ha sin data i fred!

Icke-muterbara Objekt

```
// Immutable class for pairs
class Pair {
    final int x;  // final for all instance variables
    final int y;

    Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
    …
}
```

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 sätt till final
  - Om vi har referenser 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 op 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.

Klassen Objekt

```
// Simplified view of class java.lang.Object with
// a few methods
class Object {
    …
    boolean equals(Object obj) { … }

    int hashCode() { … }

    Class<?> getClass() { … } // More in later courses

    String toString(){ … }
    …
}
```

22

I Java finns en färdig klass, Object

- Tanken är bl.a. att klassen skall innehålla metoder som alla objekt (överhuvudtaget) kan tänkas behöva
  - 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 tillhör
  - toString är tänkt att ge en läsbar representation av ett objekt (ett objekt som en sträng)
  - m.fl.

Alla objekt som vi skapar ärver automatiskt alla metoder i Object

- Metoderna syns inte i koden men finns där
- För vilket objekt o som helst kan man t.ex. skriva o.toString()

Utskrifter av Objekt

```
class Pair {
    final int x;
    final int y;
    // MUST have public first and should have override
    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}

Pair p = new Pair(1, 3);
out.println(c);  // toString() automatically called
                 // Output: (1, 3)
```

26

Alla klasser ärver metoden toString från Object-klassen

- Använder man den ärvda toString-metoden får man en utskrift liknande "Pair@7f31245a"
  - toString() metoden anropas automatiskt vid t.ex. out.println()
- För att få en mer läsbar utskrift skapar vi en egen version av toString()
  - Man väljer själv hur man vill att objektet skall representeras (hur strängen skall se ut)
  - Eftersom vi skapat en egen version måste vi skriva public framför (mer senare)
  - För att informera kompilatorn om att vi gjort en egen version skriver vi dessutom @Override över metoden (kallas en **annotering**)
    - Kompilatorn kontrollerar då att vår metod och den ärvda metoden har samma metodhuvud.
    - Ett krav för att det skall fungera enligt nedan (mer i senare kurser)!
- Eftersom vi gjort en egen metod i klassen Pair kommer vår metod automatiskt att köras istället för den ärvda. Detta beteende är inbyggt i Java.
  - Kallas **override (överskuggning)**, mer i senare kurser.
- IntelliJ kan genererar toString() metoden, Högerklicka > Generate > ...

Vi fortsätter att skilja på utskrift och logik

- toString returnerar en strängrepresentation av objektet …
- .. alltså inga utskrifter direkt i objektet!

Likhet för Objekt

```
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;
    }
}
```

24

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 metod equals i klassen.
  - OBS! Att parametern måste vara Object
  - I bilden har vi bestämt att två spelarobjekt är lika då de har lika poäng (… kanske inte så bra?)
- Eftersom vi har skapat en egen equals() måste vi lägga till public framför och bör skriva @Override över (på samma sätt som för toString()
- 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 > …

Klassmetoder

```
class ArrayUtils  {
    static int[] reverse (int[] a){       // Class (static) method
        int[] tmp = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
            tmp[arr.length - i - 1] = arr[i];
        }
        return tmp;
    }
}
 // Call directly on class
int[] revArr = ArrayUtils.reverse( arr );

// Class methods in Java classes
Character.isDigit(...)
String.valueOf(...)
Math.sqrt(..)
```


För vissa metoder gäller att man inte behöver något objekt

- Metoden har inget behov av någon data förutom parametrarna
  - De är rena funktioner

Om så, verkar det onödigt att skapa objekt…

- .. detta kan man lösa i Java genom att använda klassmetoder.
- Anges med static i metodhuvudet.
- Metoderna tillhör klassen (inte något objekt)
- För att komma åt metoderna använder man punktnotation direkt på klassnamnet (som för klassvariabel)!

Om man lägger till import static … så slipper man skriva klassnamnet.

Rent Statiska Klasser

```
 // Simplified view of class Math
public final class Math {
   /**
    * Don't let anyone instantiate this class.
    */
    private Math() {}  // private!

    public static double cos(double a) { … }
    public static double tan(double a) { … }
    public static int abs(int a) { … }
    …

}
```

26

Vissa klasser är bara rena metodsamlingar t.ex. den färdig Java klassen Math.

- Man skall inte kunna skapa några Math-objekt …
- … därför görs konstruktorn private. D.v.s. ingen utanför klassen kommer åt konstruktorn
    - Inga objekt kan skapas

Man skall inte kunna skapa några Math-objekt …

- … därför görs konstruktorn private. D.v.s. ingen utanför klassen kommer åt konstruktorn
- Inga objekt kan skapas

Klassvariabler

```
class Order {
    static int orderNumberCounter;  // Class variable
    final int orderNumber;  // Instance variable
    Order(){
            this.orderNumber = orderNumberCounter;
            orderNumberCounter++;
    }
    static int getLastOrderNumber(){  // Class method
            return orderNumberCounter;
    }
}

Order o = new Order();
int i = Order.getLastOrderNumber();  // Call directly on class
```

27

En klassvariabel tillhör inte något objekt, den delas av alla objekt av samma klass

- Anges med static vid deklarationen
- För att komma åt variabeln använder man punktnotation direkt på klassnamnet d.v.s. klassnamn.klassvariabel
- Att en klassvariabel delas av alla objekt är riskabelt
  - På samma sätt som att instansvariabler delas av alla metoder i en klass, delas en klassvariabel av alla instanser
  - … om det blir fel, vart uppstod felet (vilket objekt som helst kommer åt variabeln)?!?!?
  - Klassvariabler bör vara vara final!
- Klassvariabler är specialfall, behövs sällan …

Klass kontra Instans

```
private static int i;  // Class
private int j;  // Instance

public void doIt() {  // Instance
   out.println(i);
   out.println(j);
   this.doOther();  // Ok
}

public static void doOther() { // Class
   out.println(i);
   //out.println(j);  // Bad
   //this.doIt();     // Bad
}
```

28

Instansmetoder kan använda klassvariabler och anropa klassmetoder

Klassmetoder kan inte använda instansvariabler eller anropa instansmetoder

- Vilket skulle objektet vara i så fall?? Klassmetoden kan inte veta!
- Kan speciellt inte använda this i klassmetod, finns ingen sådan referens.

Mer om Initiering av Objekt

```
class MyClass {
    int i1 = i2;
    static int i2 = 4;
    final static int i3; // Ok, assigned in constructor
    MyClass() {
        i3 = i1 + i2;
    }
}
MyClass m = new MyClass();
out.println(m.i1);    // 4
out.println(m.i2);    // 4
out.println(m.i3);    // 8
```

29

Objekt initieras enligt (förenklat):

- Klassvariabler i skriven ordning
  - Innan någon instans har skapats
- Instansvariabler i skriven ordning …
- … därefter körs konstruktorn.

Instansvariabler som är final måste ges ett värde vid deklarationen, eller i konstruktorn, eftersom de inte kan ändras.

Konstanter

Konstanter i Java-klasser

```
Integer.MAX_VALUE;        // 2147483647
Integer.MIN_VALUE;        // -2147483648
Math.PI
```

Egen konstant

```
public class CatchTheRain {
    public final static int MAX_DROPS = 10;
    ...
}
```

30

Konstanter är ett speciellt begrepp i Java.
En konstant:

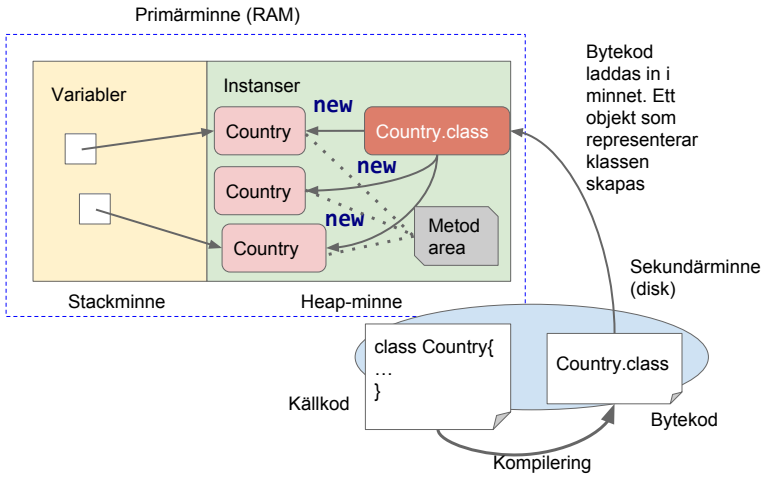
- Deklareras som public static final (och delas därmed av alla instanser)
- Primitiva variabler inget problem deklarera som ovan
- Referensvariabler
  - Objektet skall motsvara ett fixt värde (icke-muterbara)
  - Objektet skall inte användas som ett objekt d.v.s.inte anropa metoder eller indexera, bara användas som ett värde
- Konstanter skrivs med stora bokstäver avdelade med "_" t.ex. public static final int MAX_PLAYERS = …

Sammanfattning Objekt

Ett Objekt har:

- Unik identitet (referensen, (adressen))
- Tillstånd (alla instansvariabler)
- Beteende (metoderna)

Klassladdning och Instansiering



Lite mer i detalj vad som händer

- När vi försöker skapar en instans kontrolleras om klassen för objektet finns i minnet …
  - … om ej så söker Java rätt på .class-filen för klassen och läser in den i minnet (det skapas ett objekt som representerar själva klassen, Country.class)
- Instanserna skapas med klassobjektet som mall då vi använder new-operatorn
- I samband med detta körs konstruktorn
- Metoder delas av alla objekt (det är ju samma kod som skall köras, bara värden för instansvariabler skiljer)
  - De läggs därför på en plats utanför objektet som alla objekt kan komma åt kallas “metodarean”.
  - Metoderna har en dold första referens till this, så att de vet vilka instansvariabler som gäller.
  - Eventuella instansvariabler som används i metoden är, som sagt, specifika för det aktuella objektet

Heap-minne

- Objekt skapas på en plats i minnet kallat heap:en (till skillnad mot lokala variabler som finns på stacken)
- Objektet existerar så länge det finns en referens till det

- Finns inga referenser alls till objektet kommer det att skräpsamlas (d.v.s. raderas ut minnet)

main-metoden

```
public class MyClass {
    // args is an array of command line arguments
    public static void main(String[] args) {
        out.println(args[0]);   // Hello
        out.println(args[1]);   // world!
    }
}


// Executing program supplying
// command line argument after program name
java MyClass Hello World!
```

33

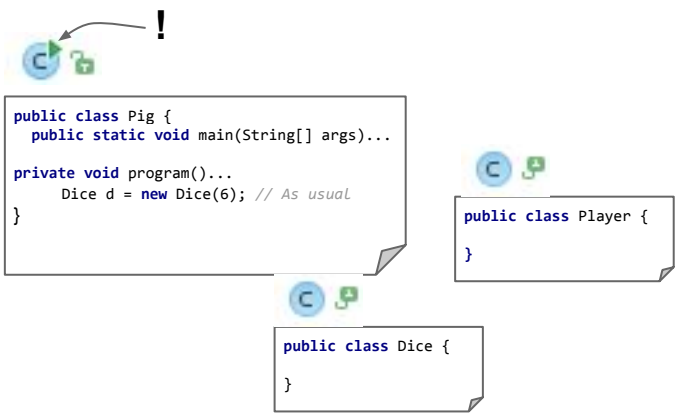
main-metoden måste finnas i alla Java-program (förenklat)

- Metoden anropas automatiskt först av allt då programmet startas
- Eftersom metoden är en klassmetod behövs inga objekt
    - Metoden anropas direkt på klassen som har metoden...
    - .. efter att klassen har laddats.
    - Då vi startar den virtuella maskinen måste vi ange vilken klass som innehåller main
        - Kan vara flera t.ex. i övningarna
        - IntelliJ visar en grön triangel på klassikonen om klassen innehåller en main-metod
            - Om så kan den exekveras (annars visas inget Run-alternativ i menyn)

I vår programmall (övningarna) har vi alltid instantiserat ett objekt av "programmet" i main-metoden

- Alla metoder vi skrivit har på det sättet blivit instansmetoder.

Exekvering med Klassfiler



34

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
  - Är klassen public så kommer vi åt den och kan därmed instansiera

Konstruera Objektmodeller

```
@Override
public void init(){   // JavaFX initialization method, run before graphics
   // Create object model here (if possible)
   int MAX_DROPS = 10;
   // Create an object
   Bucket bucket = new Bucket(width / 2,
                              height - 40, 40, 40, Color.BLUE);
   // Supply the bucket as argument to next object
   catchTR = new CatchTheRain(bucket, MAX_DROPS, width, height);
}

// Main model class (keep model objects connected)
public CatchTheRain(Bucket bucket, int nDrops, int width, int height) {
   this.bucket = bucket;  // This object will have reference to bucket
  ...
}
```

35

Ett program består av (ofta väldigt många) samverkande objekt av olika typer.

- För att kunna samverka måste de känna till varann (ha referenser till varann eller på annat sätt kunna anropas)
- Hur åstadkomma detta?

Vår metod är att:

- Försök skapa och koppla ihop så mycket som möjligt på ett ställe i programmet
  - Vi använder t.ex. init()- metoden i JavaFX
- Använd konstruktorer för att koppla ihop objekten
  - Skapa objekten vartefter och skicka dessa som konstruktorargument till andra objekt som behöver känna till dem..

I bilden: Objektet catchTR (av typen CatchTheRain) behöver en referens till ett bucket-objekt (av typ Bucket)

- När programmet senare körs kan catchTR använda hinken, t.ex. se till att den flyttar sig.
- Ofta anropar man först catchTR som i sin tur anropa bucket (för att flytta den).