Klasser och Objekt

Vecka 4, Bildserie 1

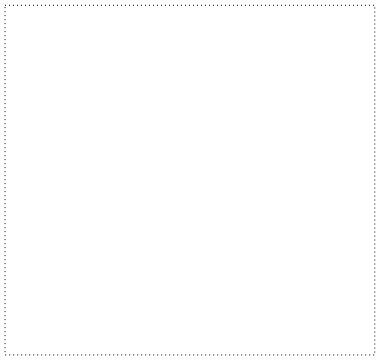
Innehåll

- Objekt
- Klass och instans
- Klassdeklaration
- Mer om Instansvariabler
- Instansiering
- Klass, typ och referens
- Punktnotation igen
- Instansmetoder
- this
- Konstruktorer
- Utskrifter av objekt
- Icke-muterbara objekt

Mål vecka 3

Kunna strukturera lösning till ett problem m.h.a klasser och objekt

Veckans Produkt

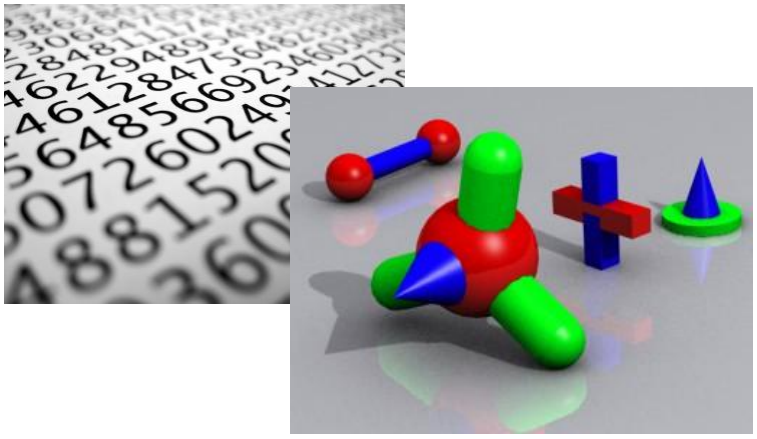


Finns ingen!. Gör /basic-uppgifter

Att Läsa i Boken

- 9.1-9.5
- 9.10
- 9.12-9.13
- 9.14.1

Sammansatt Data



Sammansatt data. Hittills har vi använt

- arrray:er av primitiva typer eller referenstyper: int[], int[][], Integer[], String[]
- array:erna har innehållit värden av samma typ

Mycket data är dock **sammansatt** d.v.s. sammansatta att av olika sorters (ev. primitiv) data

- T.ex. data för en person. Sammansatt av: ålder (int), namn (String), årsinkomst (double), o.s.v.

Många problem innehåller flera olika objekt med sammansatta data

- T.ex. en spelare, en karta, ett land… (alla är sammansatta)

Objekt

```
// Data spread out in arrays
String[] players = {"pelle", "fia", "lisa"};
int[] map = {114, 1024, 115, 146, 101357, 12487, 137, 14568, 157};
int[] owners = {0, 1, 2, 1, 2, 0, 0, 1, 2};
int[] dices = {2, 3, 2, 3, 1, 3, 1, 1, 3};
```

```
owner = "pelle";
neighbours = 114;
nDices = 2;
```

```
owner = "fia"
neighbours = 1024;
nDices = 3;
```

Country objekt

Vi har tidigare använt data som "hör ihop" utspridda i olika array:er (Dice Wars uppgiften)

- Skickade runt datan i "delar" till olika metoder.
  - Klumpigt med många parametrar.
- Möjligt problem: Felaktigt index ger **inkonsistent** data!

Bättre att samla sammanhörande data till en enhet, … ett objekt

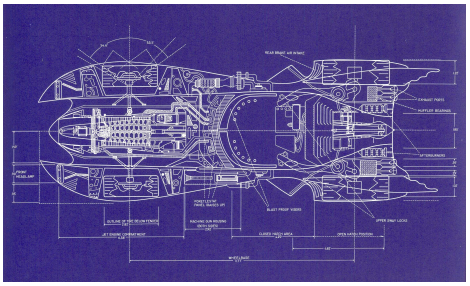
- Allt som hör ihop med ett land t.ex. en ägare, antal tärningar i landet, o.s.v. …, samlas i ett objekt

Genom att använda objekt får programmets data en mer logisk struktur

- Programmets data blir enklare att förstå och använda
- Dessutom kommer vi senare att ge objekten ett beteende (metoder) som kan använda datan.
  - Logiskt att samla data och bearbetning av datan till en enhet, … den som har datan jobbar med den.

Klass och Instans

Klass



Objekt (Instanser)

Vi kan inte skapa objekt "rakt av" i Java (med några undantag som t.ex. arrayer, strängar och enum)

- För att skapa objekt behöver vi en **klass** … en "ritning".
- Ritningen beskriver hur objekt skapade utifrån denna är konstruerade
  - Vilka egenskaper de har
  - Vad man kan göra med objekten
- Vi kan skapa ett godtyckligt (ändligt) antal objekt utifrån en och samma klass (ritning)
- En klass är en abstraktion
  - När vi skapar klassen väljer vi ut de egenskaper och det beteende som är relevanta för problemet
  - Exempel: Om vi skapar en personklass. Vad behöver vi? Skostorlek, hårfärg, ålder, personnummer …? Vi väljer!
  - Klasser hör ihop med allmänbegrepp (för den nyfikne)

Vi säger att ett objekt är en **instans** av en klass

- Tungt att hela tiden säga "En instans av klassen …" eller "Ett objekt av klassen …
- … ibland får man av kontexten förstå vad som avses
  - Vad åsyftas: Klass eller objekt? Fråga!

Klassdeklaration

```
class Country {
   String owner;  // Instance variables
   int neighbours;
   int nDices;

   // Instance method
   boolean hasBorder(int to) {
      ...
   }
}
```

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 void program() och ev. andra metoder).

Klassdeklaration

- Inleds med det reserverade ordet **class** + namnet på klassen.
  - Namn inleds med stor bokstav (CamelCase vid behov)
  - Namnet brukar vara ett substantiv.
- Därefter ett block med **klassmedlemmar**
- Klassmedlemmar är
  - **Instansvariabler**. Kallas även **attribut**
    - Fungerar som objektets minne, vad objektet måste komma ihåg.
  - **Instansmetoder**, de operationer vi kan utföra på objektets instansvariabler/attribut

Vid kompilering kommer det att skapas en class-fil för varje klasser vi deklarerar.

Instansieringsuttyck

```
// Instantiation expression
new Country()
```

När vi deklarerat klassen kan vi skapa instanser (objekt)

För att skapa en instans används ett **instansieringsuttryck**.

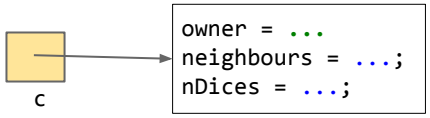
- Uttrycket använder **new-operatorn** tillsammans med ett metodanrop till en speciell metod med samma namn som klassen
  - Det finns alltid en sådan metod (osynlig i koden), mer senare …
  - … metoden kallas (förvald) **konstruktor** (default **constructor**) eftersom den används för att konstruera objekt.
  - Metoden måste heta exakt samma som klassen (skillnad mot vanliga metoder)
  - Tills skillnad från vanliga metoder anges inte heller returtyp eller void

Ett uttryck representerar ett värde (och har därmed en typ)

- Vilket värde representerar ett instansieringsuttryck? …
- … och vilken är typen?

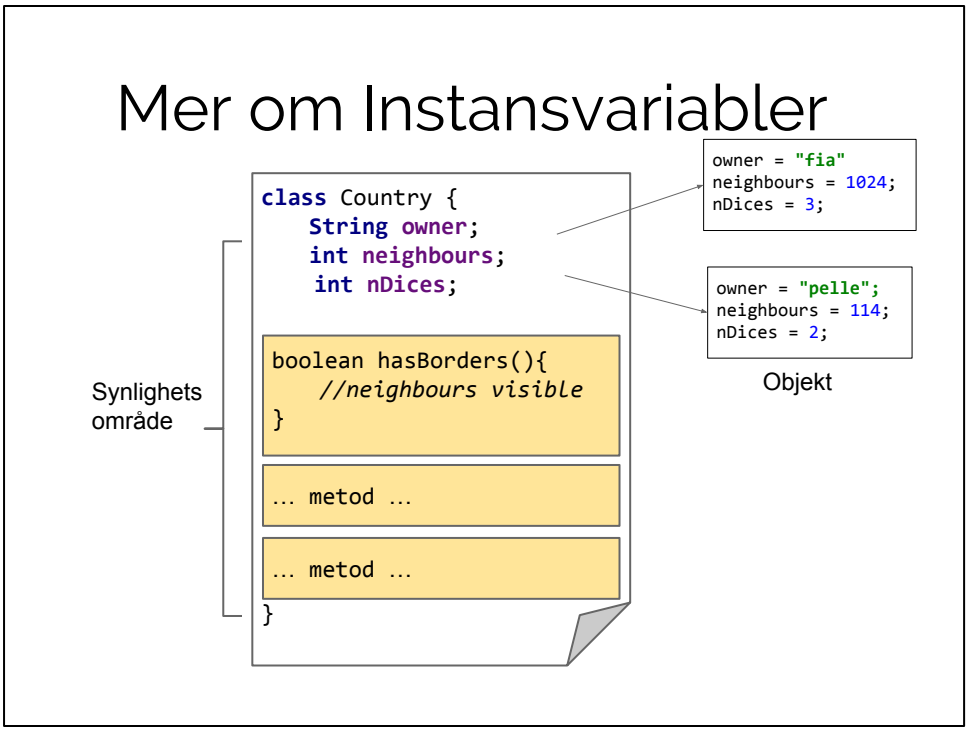
Klass, Typ och Referens

```
// Instantiate and assign
Country c = new Country();
```



En klass introducerar en ny referenstyp (på samma sätt som enum)!

- Typsystemet är utbyggbart för referenstyper!
- Om vi skapar en ny klass kan vi deklarera variabler av denna typ!
- Vi kan deklarera en referensvariabel, c, av typen Country!
  - c kan bara referera Country-objekt (eller kompatibla objekt)!
- Instansieringsuttrycket till höger returnerar en referens till ett Country-objekt …
  - Värdet är alltså en referens …
  - .. sidoeffekten är att ett (namnlöst) objekt skapas.
- Eftersom typ på värde (= referens till Country) och variabel (en referensvariabel för Country) är kompatibla fungerar tilldelning.
- Det enda sätter att komma åt objektet är via referensen c
  - Om vi inte tilldelar resultatet av instansieringsuttrycket får vi ett objekt vi inte kan komma åt
    - Dock kan vi tex. anropa en metod direkt på objektet, då metoden är klar är objektet oåtkomligt.
- Ofta använder vi klass och typ som synonymer!
  - Eftersom en klass introducerar en typ



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

Objekten som skapas utifrån klassen Country

- Har alla egna instansvariabler
  - Värden för instansvariabler kan naturligtvis variera för olika objekt.

Värdena för alla instansvariabler (mängden av värden) för ett objekt kallas objektets **tillstånd** (state)

- Viktigt begrepp, t.ex. vill vi att alla objekt skall ha ett giltigt tillstånd d.v.s. alla variabler har korrekta värden
- Ju fler instansvariabler desto större tillståndsrymd
  - Innebär att det är fler värden som måste hållas korrekta, svårt problem!
- Om ett objekt ändrar tillstånd kallas det att objektet **muterar (mutate)**

Instansmetoder


```
c1.hasBorder(1) // True
```

```
c2.hasBorder(1) // False
```

Instansmetoder använder instansvariablerna för det aktuella objektet

- För c1 gäller värdet 114 för neighbours, för c2 gäller 1024

Punktnotation Igen

```
// Instantiate
Country c = new Country();

// Dot notation
c.owner = "pelle";
c.nDices = 2;
c.neighbours = 114;

out.println( c.nDices );  // 2
out.println( c.hasBorder(4) ); // True
```

För att komma åt medlemmarna i objektet används punktnotation på referensen!

- Som för enum.
- Punktnotation betyder: gå till objektet och välj medlemmen (variabel eller metod) med angivet namn.
- c.nDices betecknar en int-variabel!
- c.hasBorder(4) betyder att metoden hasBorders i objektet c refererar skall exekveras.

Programmering

```
// To be use by Dice Wars
Dices ds = new Dices();

// Arg is number of dices
out.println(ds.roll(3)); // 3-18
```

Skapa en klass för tärningar vilken man kan använda som i bilden

- Tänkt att användas i t.ex. Dice Wars

Återblick

Instansiera slumptalsgenerator

```
Random rand = new Random();  // Instantiat a random object

int r = rand.nextInt(); // Use dot-notation to call method
```

Instansiera Scanner

```
Scanner sc = new Scanner(in);  // Constructor has arg

int r = sc.nextInt(); // Use dot-notation to call method
```

Detta har vi gjort tidigare utan närmare förklaring

- Vi instansierar objekt m.h.a. färdiga Java-klasser
- Scanner har en konstruktor som tar ett InputStream-objekt parameter, mer strax …

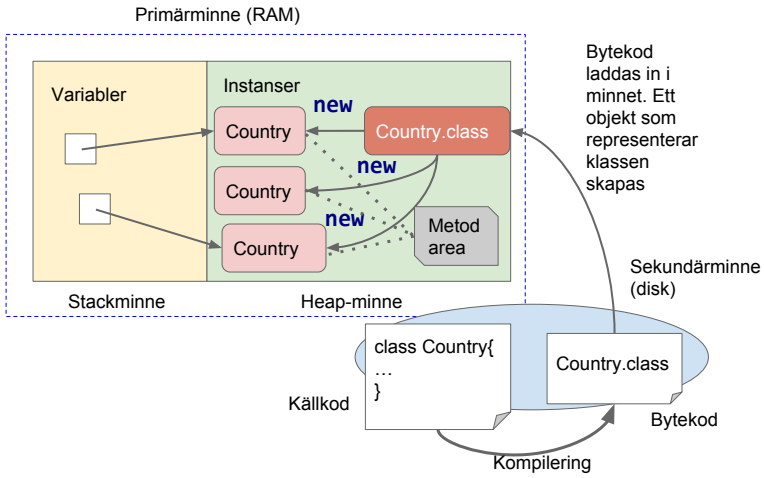
Programmering

```
TicTacToeBoard board = new TicTacToeBoard();
...
if (board.isEmpty(index)) {
   board.positions[index] = marks[actual];
   if (board.hasWinner(marks[actual])) {
       winner = actual;
       break;
   }
}
...
```

Skapa en klass för spelplanen i “Tre i rad” (Tic Tac Toe) som kan användas enligt bilden.

- marks[actual] är ‘X’ eller ‘O’
- index är någon ruta på spelplanen.

Klassladdning och Instansiering



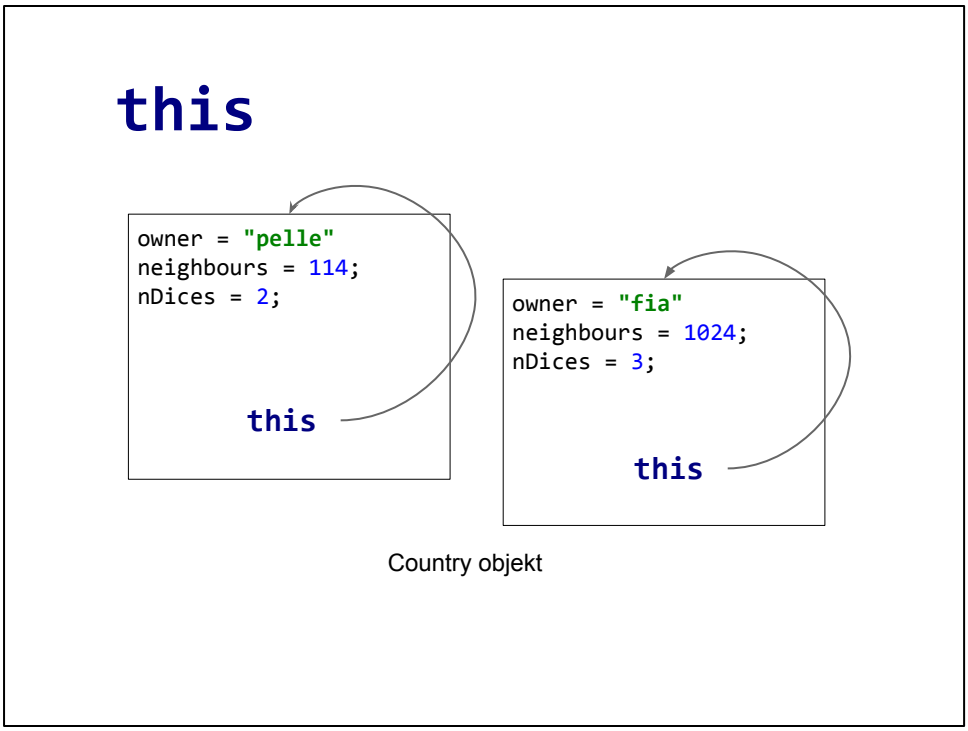
Lite mer i detalj vad som händer

- 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)
  - De läggs därför på en plats utanför objektet som alla objekt kan komma åt kallas “metodarean”.
  - Eventuella instansvariabler som används i metoden är, som sagt, specifika för det aktuella objektet
    - Alla metoder har en dold parameter till 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)
- Om objektet skapats i en metod kommer variabeln med referensen att försvinna då metoden kört klart
    - Objektet kommer då att skräpsamlas (om inte referensen returneras och sparas någon annanstans)



Om vi skriver ett namn i koden (i en klass), t.ex. namnet på en instansvariabel eller instansmetod, är det underförstått att vi syftar på det aktuella objektet

- … men ibland (t.ex. vid namnkrockar) måste vi uttryckligen ange att vi syftar på det aktuella objektet.

Hur åstadkomma detta i Java?

- Man använder det reserverad ordet **this**!
  - this i koden syftar på objektet självt (det aktuella objekt som skapats under exekvering)
  - this är en referens till objektet
- this kan bara användas i instansmetoder (och konstruktorer, mer strax …)
- this är ett reserverat ord

Mer om Konstruktorer

```
class Country {
    String owner;  // = this.owner
    int neighbours;
    int nDices;
    // Contructor
    Country(String owner, int neighbours, int nDices) {
        this.owner = owner;
        this.neighbours = neighbours;
        this.nDices = nDices;
    }
    ...
}
// Instantiation, use contructor
Country c = new Country("pelle", 2, 114);
```

Om vi vill initiera ett objekt på ett visst sätt skapar vi 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 förvald parameterlös konstruktor (default konstruktor), även om inget syns 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, forts. följer …

Konstruktoröverlagring

```
class Country {
    String owner;
    int neighbours;
    int nDices;
    // Contructor
    Country(String owner, int neighbours, int nDices) {
        this.owner = owner;
        this.neighbours = neighbours;
        this.nDices = nDices;
    }
    // Overloaded constructor
    Country(String owner) {
        this.owner = owner;
    }
}
```

Konstruktorer kan överlagras på samma sätt som vanliga 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 så måste parametrarna skilja sig åt

Kopieringskonstruktor

```
class Country {
    String owner;
    int neighbours;
    int nDices;
    // Copy contructor
    Country( Country other ) {
        owner = other.owner;
        neighbours = other.neighbours;
        nDices = other.nDices;
    }
    ...
}
Country c1 = new Country("pelle", 2, 114);
Country c2 = new Country( c1 );
```

Kopieringskonstruktor (copy constructor)

- Är en konstruktor som tar som enda parameter ett objekt av samma typ som klassen
- Ett smidigt sätt att skapa kopior utifrån existerande objekt.
- Behöver inte använda this här eftersom vi har other framför parametern

Utskrifter av Objekt

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

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

Om man skriver ut ett objekt “rakt av” med out.println(...) får man en utskrift liknande “Pair@7f31245a”

- Utskriften säger att det är ett objekt av typen Pair och att objektet har ett id (skrivet på hexadecimal form)
    - Vanligen inte vad man vill ha ...

I Java finns en speciell mekanism för att skriva ut innehållet i ett objekt

- Man skapar en metod toString som returnerar en strängrepresentation av objektet
- Man väljer själv hur man vill att objektet skall representeras (hur strängen skall se ut)
- Metoden anropas automatiskt vid out.println(...)
- Det finns redan en färdig toString()-metod i Java (finns i alla objekt). För att visa att inte den skall köras (utan vår egen) anger vi @Override framför
    - OBS! Av tekniska skäl måste det stå public framför metoden, mer senare ...

Vi fortsätter att skilja på utskrift och logik

- En klass representerar “något” (logiskt), däremot beskriver det INTE hur något skall visas
- De klasser vi arbeter med f.n. innehåller aldrig IO (out.println)

- Om vi vill skiva ut ett objekt så sköts detta av något annat objekt.
    - I vårt fall sköts all IO av själva programmet (IO-metoder i programmet).

Icke-muterbara Objekt

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

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

Icke-muterbara objekt är objekt som inte kan ändra tillstånd

- Alla instansvariabler ges ett värde då objektet skapas, därefter kan inget ändras.
- Åstadkoms genom att alla instansvariabler i klassen sätts final
    - Har vi referensvariabler räcker inte detta .. men vi bortser från detta just nu.
- Objekten är alltid i ett giltigt tillstånd (om vi initierat korrekt)
- Icke-muterbara objekt är säkra att jobba med, inga alias-problem eftersom tillståndet inte kan ändras m.m.!

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.

Programmering

```
ComplexNumb c1 = new ComplexNumb(1, 2);
ComplexNumb c2 = new ComplexNumb(-3, -5);
ComplexNumb c3 = new ComplexNumb(c2);

// Usage
out.println(c3.equals(c2));
out.println(c1.sub(c2));
out.println(c2.add(c1).add(c3).add(c3));
```

Skapa en klass för komplexa tal som kan användas som i bilden.

- Objekten skall vara icke-muterbara
- Hur skall strängrepresentationen se ut?