Informationsgömning, Klassvariabler och metoder, Strängar

Vecka 5, Bildserie 1

Innehåll

- Informationsgömning
- Klassvariabler och metoder
- Konstanter
- String
- StringBuilder
- Kort for-loop

Mål

Informationsgömning

Kunna arbeta med texter

- Mer om standardklassen String och StringBuilder

Kunna hantera undantag

Filhantering

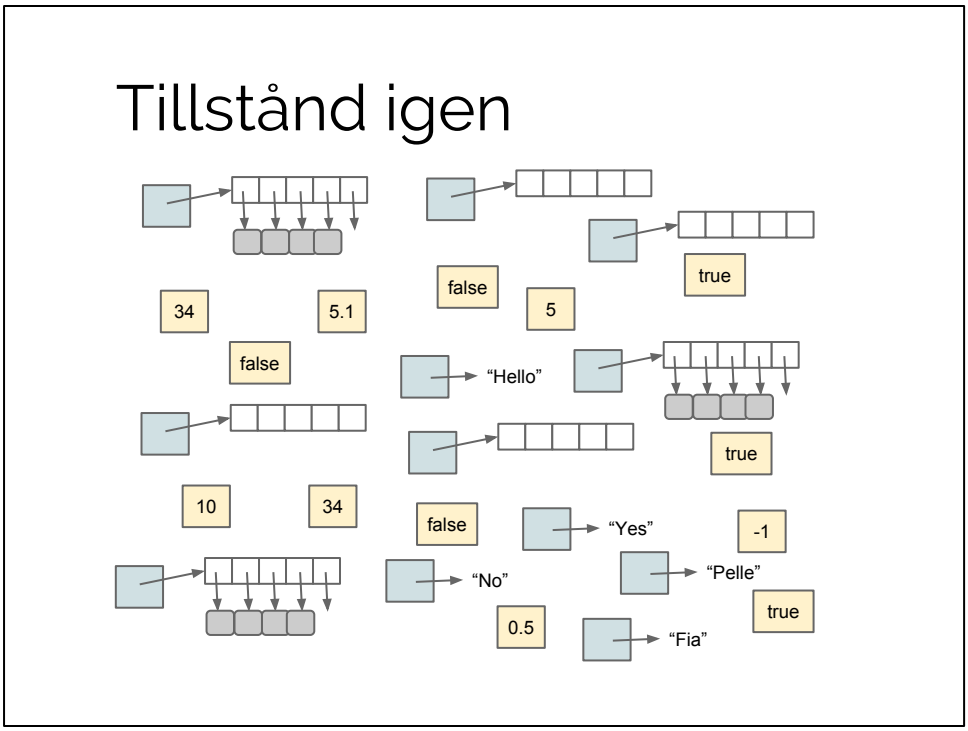
Veckans Produkt

```
...
Guess a char > k
---|
   |
   O
  /|\


k _ _ s t _ _ k t _ _
Guess a char > a
---|
   |
   O
  /|\
  /

k _ _ s t _ _ k t _ _
Guess a char > ...
...
```

Ett spel igen, hänga gubbe (hangman)

Att Läsa i Boken

- 9.7
- 4.3-4.4
- 10.10 (ej 10.10.7)
- 10.11
- 7.2.7



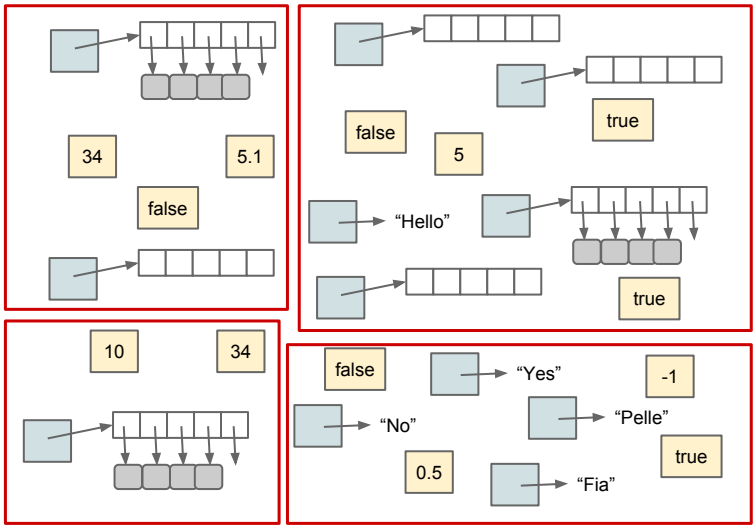
Vi har tidigare berört tillstånd (state) mängden av alla värden för alla instansvariabler i programmet vid en viss tidpunkt under exekveringen

- 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. …

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.

Informationsgömning



Ett sätt att 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 till ståndet skall 0

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

Klassfiler

```
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;
   }
}
```

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

- Namn för klass och fil som förut (samma namn etc.)
- 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 användsas 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 inledningsvid 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

```
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;
   }

}
```

Eftersom vi sätter alla instansvariabler till private kan ingen annan kod komma åt dem
Leder till vissa problem…

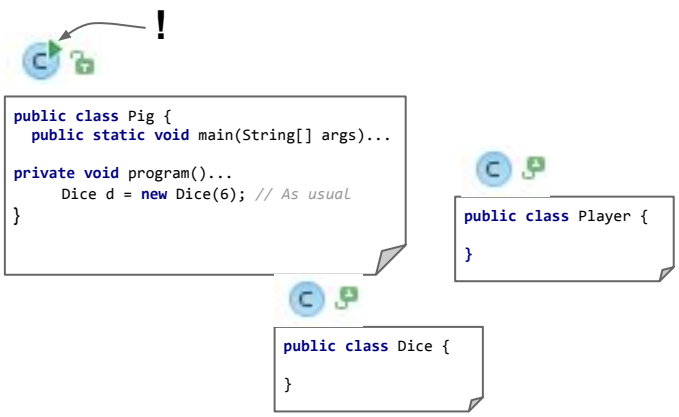
Ibland måste vi läsa av tillståndet t.ex. vid utskrifter

- 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
- Eftersom objektet har datan, gör beräkningar i objektet och skicka ut resultatet, istället för att att skicka ut datan! Låt objekten ha sin data i fred!
- 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

Exekvering med Klassfiler



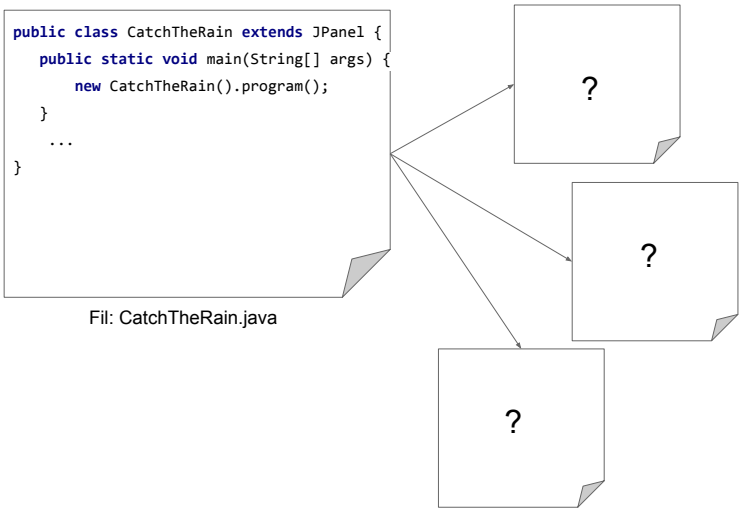
Om man har ett Java program uppbyggt av ett antal klassfiler måste en av dessa innehålla metoden main

- main-metoden anropas automatiskt först av allt då ett program startar.
    - Vi har också vår program()-metod i samma fil
- 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

IntelliJ visar en grön triangel på klassikonen om klassen innehåller en main-metod

- Om så kan den exekveras (annars visas inget, inget Run-alternativ i menyn)

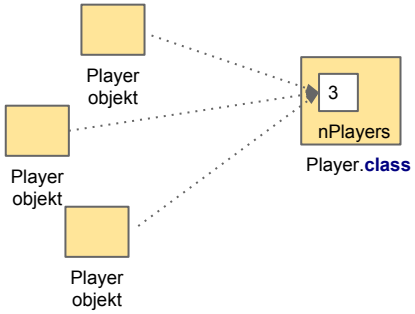
Programmering



Dela upp programmet CatchTheRain (finns på kurssida) i klassfiler.

- En fil får innehålla huvudprogrammet (med all IO) och main-metoden.

Klassvariabler



Antag att alla spelarobjekt (se bild) behöver veta hur många spelarobjekt det finns.

- Om all objekt hade en variabel för detta skulle alla alltid ha samma värde.
- … onödigt.
- Bättre att låta alla instanser dela på en variabel (eftersom värdet ju alltid är detsamma för alla)
  - Ett sätt att göra detta är att använda en **klassvariabel.**
  - Klassvariabeln finns i "klassen", inte i något enskilt objekt
  - Alla objekt måste naturligtvis kunna komma åt klassvariabeln.

I vissa (sällsynta) situationer kan klassvariabler vara användbara

Att en klassvariabel delas av alla 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 skall därför alltid (om möjligt) anges som final.
- Klassvariabler (icke final) räknas in i tillståndet (alltså mindre bra)

Initiering av klassvariabler sker då klassen laddas (från en *.class-fil).

- Alltså innan någon instans har skapats (med new). Kan vara användbart i vissa sammanhang.

Deklaration av klassvariabler

```
public class Player {

   private static int nPlayers;
   …
}
```

Anges med det reserverade ordet static

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;
    ...
}
```

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-muterbart)
    - 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 = …
    - Används sparsamt för applikationsglobala värden.

Klassmetoder

Klasmetoder från Math.

```
double d = Math.sqrt(25); // Dot notation on class
d = Math.pow(5, 2);
```

Klassmetoder för omslagsklasser

```
String s = String.valueOf(123); // int to String
Integer.min(45, 67);
Intger i = Integer.valueOf("123"); // String to int
boolean b = Character.isDigit('9');
b = Character.isSpaceChar(' ');
```

För vissa metoder gäller att man inte behöver något objekt

- Metoden har inget behov av någon data förutom parametrarna
- T.ex. rena beräkningar (rena funktioner med indata->utdata).

Om så, verkar det onödigt att skapa objekt…

- .. detta löser man i Java genom att införa klassmetoder.
- Metoderna tillhör klassen (inte något objekt)
- För att komma åt metoderna använder man punktnotation direkt på klassnamnet!
- Exempel:
  - Alla metoder i Math är klassmetoder..
  - Omslagsklasserna har gott om klassmetoder.
    - Metoder för typomvandlingar användbara!
      - Särskilt mellan String och annan primitiv typ
- IntelliJ visar klassmetoder med kursiv stil

Character

```
Character.isWhitespace(' ');
Character.isDigit('1'));
Character.isLetter('X');
Character.isLetterOrDigit('2');
Character.isLowerCase('c');
Character.toString('Z').equals("Z");
```

Standardklassen Character innehåller en hel del användbara klassmetoder för tecken

- Vi skall jobba med texter denna vecka!

Egna Klassmetoder

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

Order o = new Order();    // Will have own order number
int i = Order.getLastOrderNumber();  // Call directly on class
```

Vi kan skapa egna klassmetoder

- Anges med static i metodhuvudet
- Klassmetoder kan överlagras!

static förekommer ju också i samband med import, lite annan betydelse.

- Innebär att vi kommer åt (alla) statiska metoder i en klass utan att behöva skriva klassnamnet före.

Klass och Instans

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

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

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

Instansemetoder kan använda klassvariabler och klassmetoder

Klassmetoder kan inte använda 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.

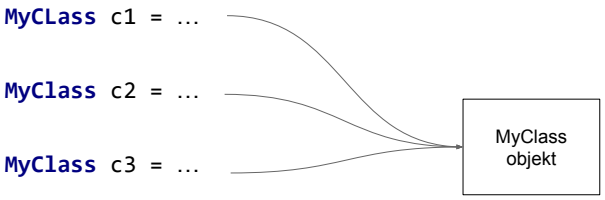
Rent Statiska Klasser

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

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


}
```

Vissa klasser är bara rena metodsamlingar .t.ex. Math.

- Man skall inte kunna skapa några Math-objekt …
- … därför görs konstruktorn private. D.v.s. ingen untanför klassen kommer åt kontruktorn

Programmering



Hur skapa en klass som man bara kan instansiera ett enda objekt av (en sådan klass kallas en Singleton)

- TIPS: Använd klassvariabler och metoder.
- Detta är kanske inte det bästa sättet att göra en Singleton på men lite intressant …

Återblick

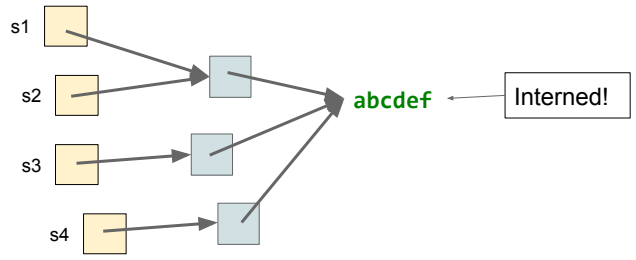
```
import static java.lang.System.*;
// If no import: System.out.println()
out.println(...);   // final class variable

import static java.lang.Math.*;
out.println(sin(...));     // If no import: Math.sin()

KeyEvent.VK_UP    // Constant

Arrays.toString( arr );    // Class method

Color.BLUE     // Constant
```

Analys

- out är en klassvariabel i klassen System (referens till ett objekt)
- sin är en klassmetod i klassen Math
- KeyEvent innehåller en mängd konstanter bl. a. för tangentbordskoder
- toString är en klassmetod i klassen Arrays
- BLUE är en konstant i klassen Color

Strängar

```
String s1 = "abcdef"; // Object created
String s2 = "abcdef"; // No new object
String s3 = s1 + "";  // Object created
String s4 = new String("abcdef");  // Avoid
```



Sträng är en standarklass för texter (följder av tecken) i Java (alla moderna språk har inbyggt stöd för att arbeta med texter)

- Alla strängar är instanser av referenstypen String
  - Strängar är alltså objekt
- Strängliteraler skrivs med omslutande “-tecken (som vi redan vet)
- Strängar är icke-muterbara
  - Operationer som innebär förändring av strängen medför att nya strängar skapas (jämför vår Complex-klass)
  - Gäller i synnerhet +-operatorn
- Undvik att använda String-konstruktorer, skapar onödiga objekt
- Alla strängliteraler delar samma object (gäller alltså s1 och s2 i bilden)
  - Själva teckenföljden sparas i en “pool”.
    - “All literal strings and string-valued constant expressions are interned.”
  - Om strängen redan finns i poolen sparas den inte igen
- En sträng är inte samma som en char[] .. dock …
- … enskilda tecken i strängen indexeras som en array (börjar på 0)

Strängar och Likhet

```
String s1 = "abcdef";
String s2 = new String(s1);

out.println(s1 == s2);  // False (by ref. semantics)

// Must use for value sematics
out.println(s1.equals(s2));  // True

// Also value sematics
out.println("olle".compareTo("fia") < 0);  // True
```

För strängar gäller referenssemantik för jämförelse (==)

- Vanligen vill vi ha värdesematik (d.v.s. jämföra tecknen inte pilarna)
  - Vi får detta genom att använda den färdiga metoden equals()
    …
  - … eller metoden compareTo()
    - Ger 0 vid likhet
    - < 0, om argumentet är mindre i lexikografisk ordning (tecken för tecken vänster till höger)
    - > 0, om argumentet är större i lexikografisk ordning
  - Alla strängobjekt har dessa metoder..

Strängmetoder

Inspektera

```
str.isEmpty();
str.length();
"abcdef".charAt(3);   // 'd'
"abcdef".indexOf('a');   // 0
```

Söka

```
str.contains("cd"));
str.startsWith("abc");
str.endsWith("def");
```

Manipulera (alla skapar nya objekt)

```
str.replace("failure", "icecream");
"abcdef".substring(0, 4);  // "abcd"
"abcdef".substring(4);    // "ef"
"abc:def".split(":");   // [ "abc", "def" ]   (array of strings)
```

Olika kategorier, se kodexempel

- Inspektera
- Söka (i texten)
- Manipulera (förändra texten)

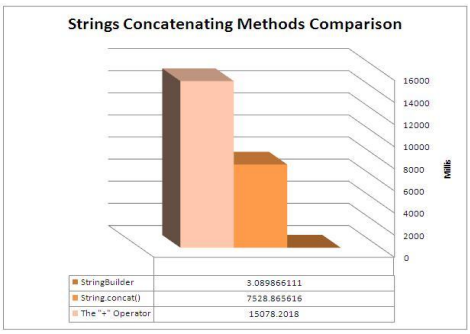
Ni behöver inte kunna metoderna utantill

- Skulle det behövs (dvs. tentan) så får ni en lista på användbara
- Har är dokumentationen av String

För att söka och manipulera används ofta reguljära uttryck (regular expressions, reg exp)

- Vi hinner inte gå in på det … (se kodexempel)

StringBuilder




```
StringBuilder sb = new StringBuilder();
out.println(sb.append("hello")
    .append(" ")
    .append("goodbye")
    .toString());   // Convert to String
```

Eftersom +-operatorn hela tiden skapar nya strängar och kopierar över tecken för tecken till dessa kan det bli
kostsamt (i tid mätt)

En snabbare variant är att använda en StringBuilder.

- StringBuilder fungerar som en muterbar sträng
- append-metoden lägger till sist i strängen (utan kopiering)
    - Smidigt med kedjade anrop
- toString omvandlar StringBuildern innehåll till sträng (icke-muterbar)

Hmm, skapar inte append() nya objekt (om vi använder kedjade anrop)?

- Nej, … (hur fungerar append?)

Övning

Rövarspråket

```
"Jag talar rövarspråket" →
        "JoJagog totalolaror rorövovarorsospoproråkoketot"
```

Skriv program som översätter till “Rövarspråket”

Generellt angreppssätt

- Sök färdiga metoder i String som gör “lämpliga” delar
    - Undvik att skriva egna metoder, det du söker finns oftast
- Kombinera dessa till en lösning