Bilder Vecka 5

TDA548/Joachim von Hacht

Metoder

Kedjade metodanrop

```
Complex c = new Complex(1, 2);
// Chained calls (invisible objects)
c.add(c).add(c).equals(new Complex(3, 6));


// Returns an object
Complex add(Complex other) {
   return new Complex(re + other.re, img + other.img);
}
```


Om metoden returnerar en referens till ett objekt kan vi direkt anropa en metod på objektet (utan att spara referensen i en variabel).

- Sker detta i flera steg så finns det ett antal “osynliga” mellanliggande objekt.
- Denna teknik kan ge smidig kod i vissa fall.

En annan variant för att möjliggöra kedjade anrop är att returnera this

- Inga mellanliggande objekt kommer då att skapas
- Ingen bra idé för koden i bilden, där vill vi ha nya objekt, eftersom resultatobjektet skall vara helt skilt från operanderna.

Instans- och Klassmetoder

```
static int getPrecedence(String op) {
    if ("+-".contains(op)) {
        return 2;
    } else if ("*/".contains(op)) {
        return 3;
    } else if ("^".contains(op)) {
        return 4;
    } else {
        throw new RuntimeException(OP_NOT_FOUND);
    }
}
```


See Bildserie om klasser

Åtkomst för Metoder

public
private
(protected)

See bildserie om klasser

Abstrakta Metoder

```
public abstract String say();
```

6

En abstrakt metod saknar metodkropp (ingen körbar kod).

- En klass som innehåller någon abstract metod måste anges som en abstrakt klass
  - Se vidare Klasser.

Typer

(inget nytt)

Klasser

Instansmetoder

```
class Player {
   int points;  // Instance variable

    // Declare instance method
   void incPoints(){
      points++;
   }
}

// Later elsewhere, create object
Player p = new Player();
p.incPoints();  // Call instance metod
```

9

Klasser kan innehålla metoder!

Instansmetoder är metoder som är deklarerade i en klass.

- De kan anropas på alla objekt skapade utifrån klassen
- Det måste finnas ett objekt för att kunna anropa metoderna..
- I vilken ordning instansmetoder deklareras spelar ingen roll.
- Alla deklarationer ligger på samma nivå i klassdeklarationen (ej nästlade)

Mer om Instansvariabler

```
public class Person {
   private final String name;
   private int age;
   private double income;

   Person(String name, int age, double income) {
       this.name = name;
       this.age = age;
       this.income = income;
   }

   int getAge() {
       return age;
   }

   void setAge(int age) {
       this.age = age;
   }

      // Objects has the data to be able to answer questions!
   boolean isRetired(int retireAge) {
       return age >= retireAge;
   }
  ....
```

Synlighets område

10

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

- Synlighetsområdet för instansvariabler är hela klassen
  - I alla metoder!!
- Lokala variabler kan ha samma namn som en instansvariabel men isf döljs instansvariabeln av den lokala variabeln
  - Kan komma åt med this (som vi sett).
- Ett problem med instansvariabler är om något blir fel, vilken metod orsakade felet??
  - Undvik instansvariabler om det går, föredra lokala variabler.
  - Oftast behövs dock instansvariabler, så fort något skall "kommas ihåg" mellan metodanrop måste vi använda instansvariabler.
- Alla instansvariabler kommer att initieras med förvalda värden, t.ex. 0 för int.

Konstruktoröverlagring

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

    Complex(Complex other) {   // Another constructor
        re = other.re;  // No need for ‘this’, no name clash
        img = other.img;
    }
     …
}
```


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 …

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
    }
    ...
}
```


Konstruktorer kan anropa andra konstruktorer.

- Genom att skriva this(...) , d.v.s. parenteser efter this, så avses någon konstruktor (med matchande parameterlista).
- Kan ge lite enklare kod.
- Se vidare nedan.

Klassfiler

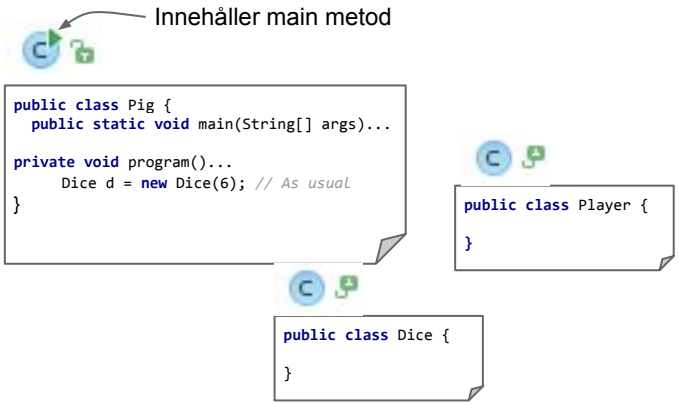
```
class Player {
   String name;
   int points;
}

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

Player.java

I riktiga Java-program deklareras klasser i helt egna filer vanligen med en klass per fil.

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

Exekvering med Klassfiler



14

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

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(nFaces) + 1;
    }
}
```

15

I Java kan man åstadkomma informationsgömning genom att dela upp programmet i klassfiler

- 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**, använder vi troligen inte (innebär att subklasser kan komma åt, se 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, … vilket det tyvärr ofta gör)
- 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 , som ovan.

- 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.

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

Konstanter

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

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

16

Konstanter är ett speciellt begrepp i Java.
En konstant:

- Deklareras som public static final (en klassvariabel)
- 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 = …
- Vi är lite mindre strikta, skriver de flesta "static" med stora bokstäver.

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;
   }

}
```

18

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
public class Pair {
    private final int x;
    private final int y;

    public 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 **Object**

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

    int hashCode() { … }

    Class<?> getClass() { … }

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

19

I Java finns en färdig klass, Object (och därmed typen 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()
- Se också Typer (typen Object)

Utskrifter av Objekt

```
public class Pair {
    private final int x;
    private 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)
```

22

Alla klasser ärver metoden toString från Object-klassen

- Använder man den ärvda toString-metoden får man en utskrift liknande "Pair@7f31245a" (i bilden)
  - 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)
  - För att informera kompilatorn om att vi gjort en egen version skriver vi @Override över metoden (kallas en **annotering**)
    - Kompilatorn kontrollerar då att vår metod och den ärvda metoden har exakt samma metodhuvud, så måste det vara!
- Eftersom vi gjort en egen version av metoden i klassen Pair kommer vår metod automatiskt att köras istället för den ärvda. Detta beteende är inbyggt i Java.
  - Kallas **override (överskuggning)**.
  - Vad som körs beror alltså på objektets typ.
- 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

```
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;
    }
}
```

21

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

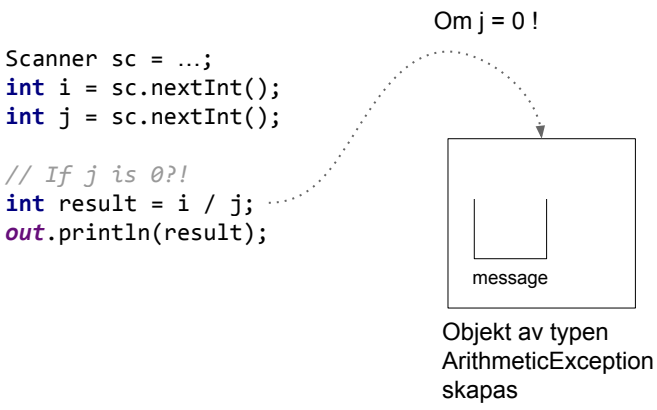
- När vi bestämt oss skapar vi en egen version av metod equals i klassen.
    - Som tidigare måste vår metod ha exakt samma metodhuvud som den ärvda (parametern måste vara av typen Object).
    - I bilden har vi bestämt att två spelarobjekt är lika då de har lika poäng (… kanske inte så bra?)
- Eftersom vi har skapat en egen equals() skriver @Override över
- 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 > …

Undantag

Undantag



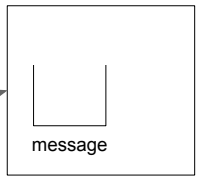
23

Programmet har under körning hamnat i en omöjlig situation (division med 0)

- Ett undantag (exception) uppstår
- I samband med undantaget:
  - Skapas automatisk ett objekt som bl.a. innehåller information om undantaget
  - Avbryts programmet …. om inte undantaget fångas, mer strax…
    - En felutskrift skrivs ut. Se Grunderna.

Fånga undantag

```
int i = ...;
int j = ...;
int result;

try {
    result = i / j; // If 0..
} catch (AritmeticException e ){
    // .. do this
    out.println("You divided by zero!");
}
out.println(result);
```



24

Att fånga ett undantag innebär att programmet inte avbryts

- Man fångar ett undantag genom att lägga ett anrop som kan kasta ett undantag i try-delen av en **try-catch-sats**
- Om inget undantag uppstår körs bara satserna i try-blocket, catch-blocket hoppas över.
- Om ett undantag uppstår, någonstans i try-delen, kommer satserna i blocket efter catch att utföras
    - Kvarvarande satser (efter undantaget) i try-blocket körs inte
    - Tanken är att man, i catch-delen, skall kunna åtgärda felet
    - Efter detta räknas undantaget som fångat och programmet fortsätter med första sats efter catch-blocket
- I catch-grenen initieras automatisk en parameter med en referens till undantagsobjektet som skapades
    - Typen på undantagsobjektet måste stämma med parametertypen
        - Annars sker ingen “fångning”

Kasta Undantag

```
public void add( String s ){
  if( s == null){
    // Create exception object and throw
    throw new IllegalArgumentException("nulls not allowed");
  }
  arr[i] = s;
  i++;
}


try {
    add( str );    // Call method
} catch (IllegalArgumentException e ){
    out.println( e.getMessage() ); // nulls not allowed
}
```

25

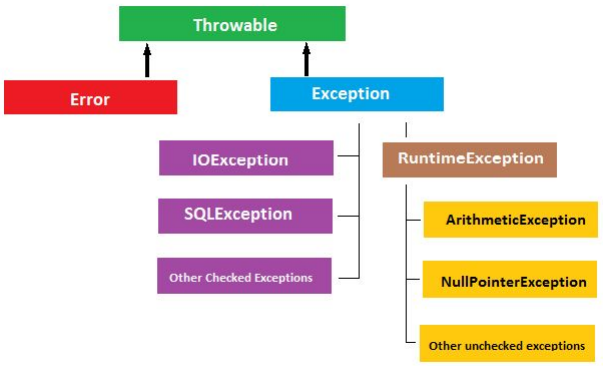
En metod kan “kasta” undantag.

- Antag t.ex. att vi har en metod som sparar referenser i en array och vi inte tillåter null-referenser i arrayen …
- … om någon annan programmerare gör fel och skickar in en referens, … vad göra???
- Jo, vi kan låta metoden kasta ett undantag om parametern är null … på så sätt upptäcks felet direkt (i stället för att null-värdet sparas och felet dyker upp långt senare).

Ett eget undantag kastas m.h.a. throw + att man skapar ett objekt av någon undantagsklass, mer strax.

- Meddelandet, argumentet till konstruktorn, kan avläsas med metoden e.getMessage() i catch-grenen
- Finns även e.printStackTrace(), skriver ut hela anropskedjan (stacken)

Undantagsklasser



26

Undantagsobjekten är instanser av olika undantagsklasser

- … klasserna är ordnade i en arvshierarki (d.v.s. det finns ett super/subtyp förhållande)

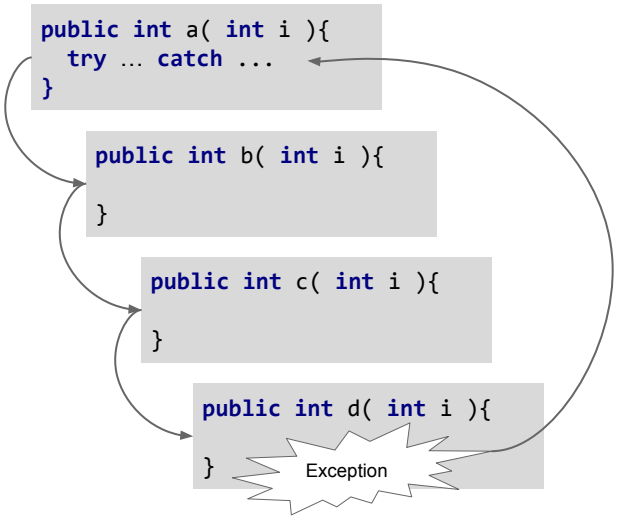
Undantagsklasserna direkt under Exception (de lila) representerar undantag som måste fångas (checked exceptions)

- D.v.s. situationer som kan kasta ett undantagsobjekt av dessa typer måste stå i en try..catch-sats
  - Kompilatorn kontrollerar, vi får ett kompileringsfel om vi inte fångar!
- Många färdiga Java-metoder kan kasta dessa typer av undantag, mer strax

Undantagsklasserna under Runtime är okontrollerade undantag (unchecked exception)

- Vi är inte tvungna att fånga dem.
- Normalt fångar man inte unchecked exceptions …
  - … man vill att undantaget skall krascha programmet (under utvecklingen) eftersom det finns ett programmeringsfel (vi har gjort fel)

Programflöde vid Undantag



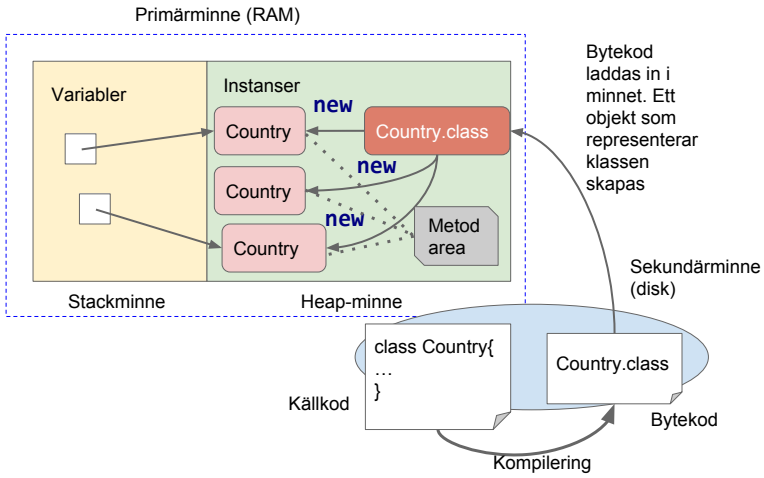
27

Ett undantag kan uppstå långt ner i en anropskejda

- I bilden: Metod a anropar b, som anropar c, som anropar d … ett undantag uppstår.
- Undantagshantering kommer att vandra genom anropen (anropsstacken) till dess den hittar en try..catch som kan fånga undantaget.
  - Vi får ett **icke-lokalt hopp** …
  - … programmet hoppar och fortsätter i en annan metod än den anropande.. (kanske väldigt långt bort, i någon helt annan klass…)
  - Finns ingen try-catch avbryts som sagt programmet.

Mer om Klasser

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)

Klassvariabler

```
class GameCharacter {

    // Share the Random object !!!
    final static Random rand = new Random();
    ...
    void turnRandom() {
        dx = 1 - rand.nextInt(3);
        dy = 1 - rand.nextInt(3);
    }
}

// No object needed, use class to access.
GameCharacter.rand.nextInt(4);
```

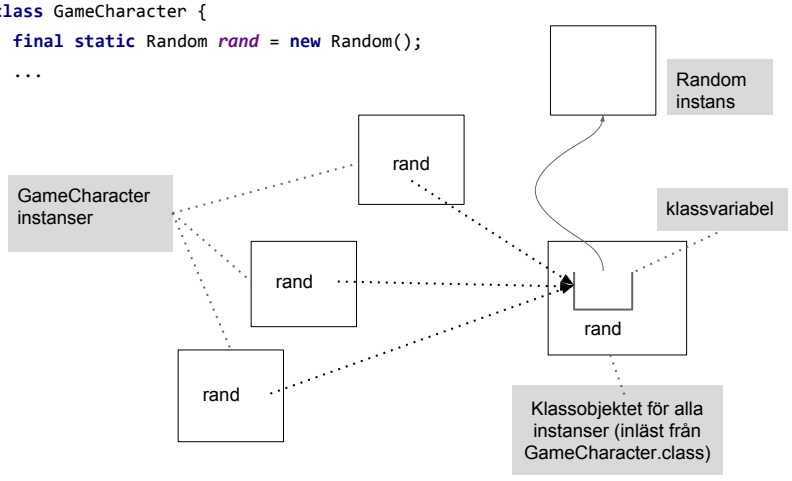
30

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

- Anges med static vid deklarationen
- Används sällan, ett exempel i bilden: Alla objekt behöver en slumpgenerator men de kan dela på den (onödigt att alla har en egen).
- För att komma åt variabeln använder man punktnotation direkt på klassnamnet.
- 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 i Minnet

```
class GameCharacter {
  final static Random rand = new Random();
  ...
```



31

Alla static variabler och metodet tillhör klassobjektet (bilden förenklad)

- Om instansen anger ett namn på en klassvariabel så syftar den implicit på variabeln i klassobjektet.

Alla instanser kan komma åt ett gemensamt icke muterbart klassobjekt (objektet som skapades utifrån class-filen.

- Instanserna kan få tag i klassobjektet med metoden getClass()
- Dock kan vi inte använda t.ex. o.getClass().rand (i bilden) för att komma åt klassvariabeln..

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 name
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 (samma som klassvariabler).
- Metoderna tillhör klassen (inte något objekt). På samma sätt som klassvariabler.
- För att komma åt metoderna använder man punktnotation direkt på klassnamnet!

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

- Så har vi gjort med t.ex. metoderna i Math.

Klass kontra Instans

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

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

static void doOther() {   // Class method
   out.println(i);
   out.println(j);  // Bad, which object?
   this.doIt();     // Bad, no this!
}
```

33

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.

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
```

34

Objekt initieras enligt (förenklat, se även Arv och Konstruktorer):

- Klassvariabler i skriven ordning
  - Innan någon instans har skapats, initieras då klassen laddas!
- 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.

- Se även Arv senare

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!
```

35

main-metoden måste finnas i alla Java-program (i någon klass)

- 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
    - 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)
- Parametern till metoden är en lista av strängar som operativsystemet skickar in då ett Java program startas.
  - Om man startar från kommandoraden skriver man strängarna efter klassnamnet.

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

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

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) { … }
    …

}
```

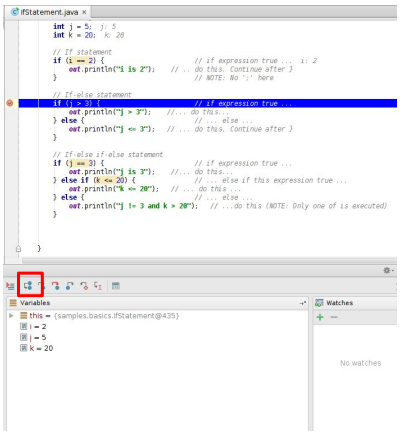

Vissa klasser är bara rena metodsamlingar t.ex. den färdig Java klassen Math.

- 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
- Alla metoder måste vara static.

Arbetssätt

Felsökning (debug)

Avlusare (debugger)



Utskrift av värde

```
out.print(someValue);
```

42

En stor del av programmeringsarbetet består av felsökning. Det finns flera alternativ

- Använda out.println() och skriv ut värden
  - Kan var en bra metod för upprepad inspektion
    - Utskriften sker varje gång programmet (loopen) körs
  - Nackdelar:
    - Blir det för många utskrifter kan man till slut inte hänga med
    - Utskrifterna måste tas bort
- Använda en avlusare (Debugger)
  - En avlusare är ett program som kan köra ett annat (ditt) program sats för sats
  - Finns inbyggd i IntelliJ

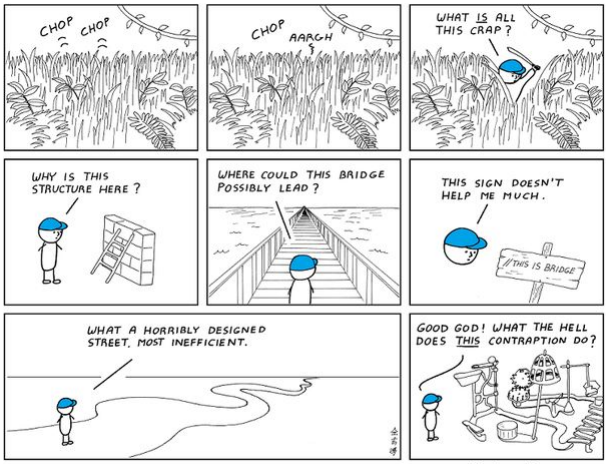
Avlusning (procedur)

- Klicka i vänstermarginalen för att få en brytpunkt (röd prick ovan).
  - En brytpunkt innebär att programmet kommer att stanna på markerad rad.
  - Klicka igen om du vill ta bort..
- Högerklicka i kodfönstret och välj Debug …
- Avlusaren startar och kör programmet fram till brytpunkten. Där stannar det.

- Därefter kan du köra sats för sats genom att klicka Step Over (röd fyrkant i bilden)
- Den blå raden skall hoppa en sats då du klickar
- Du kan hela tiden inspektera variabelvärden i det nedre fönstret
- Avsluta genom att klicka röd fyrkant ner till vänster (visas ej)
- Hakar något upp sig … börja om

OBS! Att felsöka/debuga i tester är mycket effektivt:
- Vi kan hårdkoda in data
- Vi arbetar med isolerade delar av programmet.

Programkonstruktion



I hate reading other people's code.

39

För att skapa högkvalitativ mjukvara finns ett stort antal regler eller rekommendationer för hur program skall konstrueras.

- I bilden: Om man upplever koden som i serien så har man definitivt problem med konstruktionen.
- Software construction.

Best Practices

```
if(!scene6TF1.getText().isEmpty() && scene6TF1.getText().length() < 21 && Pattern.matches("[\306\330\305\346\370\34
    if (!scene6TF2.getText().isEmpty() && scene6TF2.getText().length() < 51 && Pattern.matches("[\306\330\305\346\3
        if (!scene6TF3.getText().isEmpty() && scene6TF3.getText().length() < 81 && Pattern.matches("[\306\330\305\3
            if (!scene6TF4.getText().isEmpty() && scene6TF4.getText().length() == 8 && Pattern.matches("[0-9]+", sc
                if (!scene6TF5.getText().isEmpty() && scene6TF5.getText().length() == 4 && Pattern.matches("[0-9]+",
                    if (!scene6TF6.getText().isEmpty() && scene6TF6.getText().length() == 10 && Pattern.matches("[0
                        if(scene6CB1.getValue() != null) {
                            if(scene6CB2.getValue() != null)
                            {
                                System.out.println("Tillykke, alle felter er godkendt!");
                                // Opret medarbejder - det vil sige foretag et SQL statement der indsætter de forskı

                            } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Løntrin er ikke valgt i drop-down menu
                        } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Stilling er ikke valgt i drop-down menuen.'
                    } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Kontonummeret indtastet forkert. Feltet må kun
                } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Registreringsnummer indtastet forkert. Feltet må ku
            } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Telefonnumer indtastet forkert. Feltet må kun indeholdı
        } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Adresse indtastet forkert. Feltet må kun indeholde bogstave
    } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Efternavn indtastet forkert. Feltet må kun indeholde bogstaver,
} else { FejlMeddelelse.visFejlMeddelelse("Fejl: Fornavn indtastet forkert. Feltet må kun indeholde bogstaver, mel
});
```

Violating best practices!

45

Med best practices (beprövad erfarenhet) avses ett stort antal informella regler som ganska stor sannolikhet leder till bättre kodkvalitet. T.ex:

Namn:

- Beskrivande, lagom långa namn
- Standard förkortningar point = pts, m.fl.
- Singular/Pluralis
- Metoder inleds ofta med verb
- Booleska variabler/metoder namnges som ja/nej frågor

Arrayer:

- Arrayer används framförallt då man inte vill att strukturen skall ändras, vi kan ta bort element men index finns kvar.
- Finns inte det kravet föredra någon samling (t.ex. List)

Metoder

- En metod skall vara expert på en sak!
- Hellre många små specialiserade metoder än några stora
- “one-liners” metoder är helt ok.
- Skilj IO och logik
- Undvik utparametrar
- Undvik att returnera null (skicka t.ex. en tom array/List)

Klasser:

- En klass skall fånga ett koncept
- En klass skall ha ett ansvarsområde
- … på samma sätt som för metoder
- … annars blir de svåra att (åter)använda, felsöka, … det blir rörigt!
- Bättre att kombinera flera (små) tydliga klasser
- Använd informationsgömning!

Code Smells

```
// Bad redundant code!
if ( … ) {
    …
    dices = 1;
} else {
    …
    dices = 1;
}
```

```
// Non redundant
if ( …  ) {
   …
} else {
    …
}
dices = 1;
```

41

Code smells är tecken på att koden inte är bra (inversen till best practices)

Exemplet i bilden:
Redundant eller duplicerad kod är en code smell

- Mer (onödig) kod -> mer chanser till fel
- Kod måste hållas i synk, ändringar måste göras på flera ställen.
- Idealet är att allt finns/görs på exakt ett ställe i programmet.

Hur åtgärda:

- När ni fått till ett fungerande kodavsnitt gör en "**code review**"!
- Känns någon code smell … t.ex. görs något i onödan, eller görs samma sak på flera ställen?