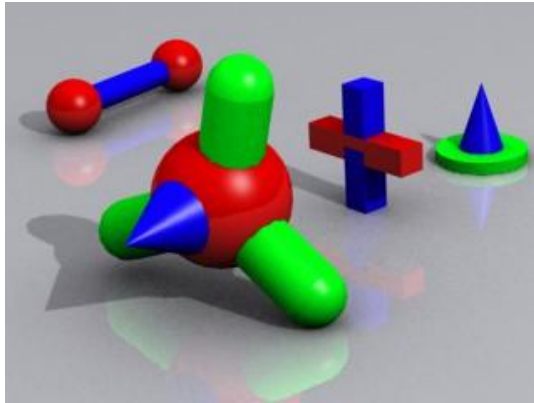


Klasser

TDA548/Joachim von Hacht

Flera men Olika



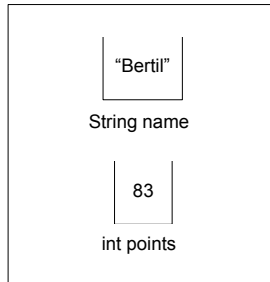
2

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 (mer senare)

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

Vi kan inte skapa objekt direkt (förutom vissa specialfall, en sträng är t.ex. ett objekt)

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

OBS Bilden! Att värdet "Bertil" egentligen inte ligger i variabeln (det är en referensvariabel, se Referenser).

Klassdeklaration

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

4

För att skapa en klass i Java gör man en klassdeklaration

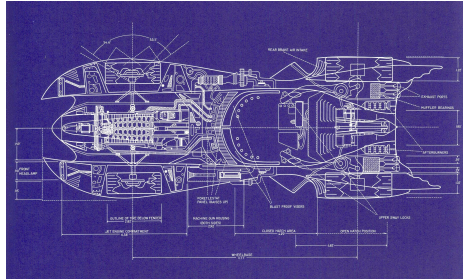
- Vi kan lägga klassdeklarationer var som helst i programmet.
- Vi lägger dem vanligen mot slutet i programmet (efter metoderna, inte i någon metod).
 - Eller i egna filer, se senare.

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**)
 - I vilken ordning instansvariabler deklarerar spelar ingen roll (lite förenklat).

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!

Variabelnamn

new-operator

Metod med samma namn som klass


```
Player p1; // Declare ...  
// ... Later instantiate and assign  
p1 = new Player();
```

6

En klass introducerar en ny referenstyp

- Typsystemet är utbyggbart för klasser, se Typer
- Om vi skapar en ny klass skapar vi en ny typ! Vi kan vi deklarerar variabler av denna typ!
- När vi deklarerat variabeln kan vi initiera den
 - För att initiera variabeln måste vi instansiera (skapa) ett objekt.
 - Instansieringen sker genom att använda **new**-operatoren tillsammans med en metod som heter som klassen.
 - 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 skapar ett objekt i minnet (inklusive de variabler som finns i objektet)
- Därefter körs metoden Player() för objektet, en sådan metod finns alltid men syns inte i koden, mer senare.
- 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
```

7

Ett namn på en array-variabel betecknade hela arrayen (lite förenklat, mer senare)

- För att komma åt enskilda variabler använde vi indexering

Ett namn på en objektvariabel (p1 i bilden) betecknar hela objektet (lite förenklat, mer senare)

- För att komma åt de enskilda variablerna används punktnotation d.v.s. objektvariabelnamn (p1), punkt, instansvariabelnamn (name)

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 ett helt objekt) ...
- ... därefter punktnotation för att komma åt variabeln i objektet.

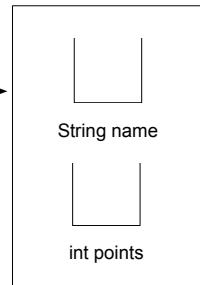
OBS! Bara `Player[] players = new Player[3];` skapar inte några player objekt, ... bara array-objektet skapas!

Mer om Variabler för Objekt

```
// Reference variable p1 and object
```

```
Player p1 = new Player();
```

Referensvariabel



Objekt

```
// Variable but no object!
```

```
Player p2;
```



Detta fungerar på samma sätt som för arrayer!

- Variabler för objekt är referensvariabler
 - En deklaration ger en referensvariabel, instansieringen (instansieringsuttrycket) ger en referens till ett namnlöst objekt
 - Värdet är alltså en referens ...
 - .. sidoeffekten är att ett (namnlöst) objekt skapas (i minnet).
- En deklaration av en variabel ger bara en referensvariabel, inget objekt!
- Punktnotation innebär att man avrefererar referensen och därefter väljer variabel utifrån namn m.h.a. punktnotation.
 - Punkten använd på objektet (inte på variabel eller referens)!

Tilldelning

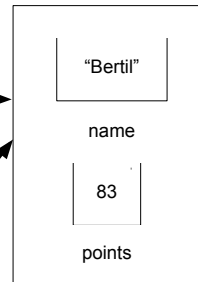
```
// Declaration and initialization  
Player p1 = new Player();
```

Referensvariabel
p1

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

Referensvariabel
p2

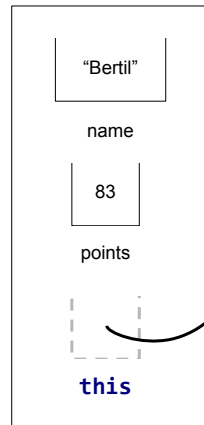
```
out.println(p1 == p2); // true
```



Fungerar på samma sätt som arrayer, det är referensen som kopieras!

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

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! can't change

```
this = ...
```

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 t.ex. vid namnkrockar, mer strax
- Den statiska typen för this är typen som klassen introducerar (Player i bilden).
 - Den dynamiska typen kan vara någon subtyp, se Typer.
- this är ett reserverat ord

Konstruktör

```
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  
    Player(String name, int points){  
        this.name = name;  
        this.points = points;  
    }  
}
```

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

12

För att initiera ett objekt på ett visst sätt kan vi skapa vi en speciell metod i klassdeklarationen, en **konstruktör**

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

En konstruktör

- Är en metod som automatiskt anropas i samband med instansiering
 - Måste ha samma namn som klassen.
 - Parameterlista som vanlig metod
 - 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 i så fall explicit **this**.

Det finns alltid en parameterlös konstruktör (den vi använt tidigare).

- Om man skapar en konstruktör med parametrar försvinner den parameterlösa konstruktorn.
- Vill man ha en sådan får man skriva dit den själv.

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
```

13

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 deklarerats 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

14

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.

Klassfiler

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

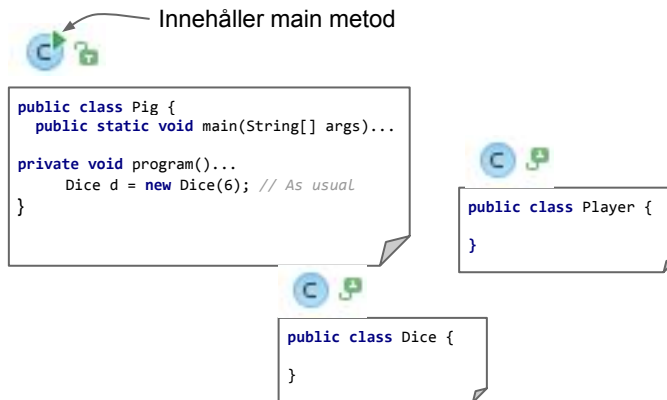
Player.java

15

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



16

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;
    }
}
```

17

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

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;  
    }  
  
}
```

19

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 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 $a \text{ op } b$ 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.

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;  
    }  
    ...  
}
```

20

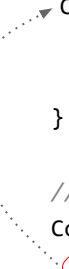
Konstruktorer kan överlagras på samma sätt som metoder.

- 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  
    }  
    ...  
}
```

A dotted arrow originates from the `this` parameter in the second constructor and points to the first constructor, indicating a recursive call to the same constructor.

21

Konstruktörer kan anropa andra konstruktörer.

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

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(){ ... }  
    ...  
}
```

22

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)
```

25

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;  
    }  
}
```

24

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

- Metoden ger referenslikhet.

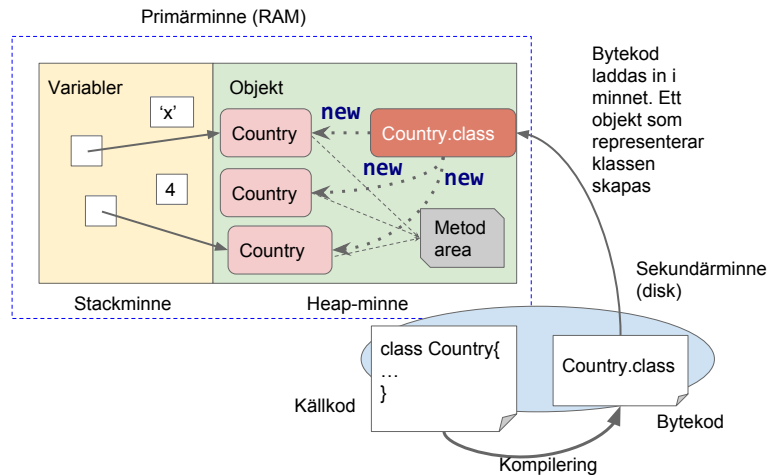
Vill vi ha värdelikhet för objekt måste vi själva definiera vad vi menar med likhet

- När vi bestämt oss skapar vi en egen 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 > ...

Klassladdning och Instansiering



28

Lite mer i detalj vad som händer

- När vi försöker skapa 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-operatoren
 - I samband med detta körs konstruktorn
 - Returvärdet, referensen till objektet, kan sparas i en variabel på stacken.
- 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

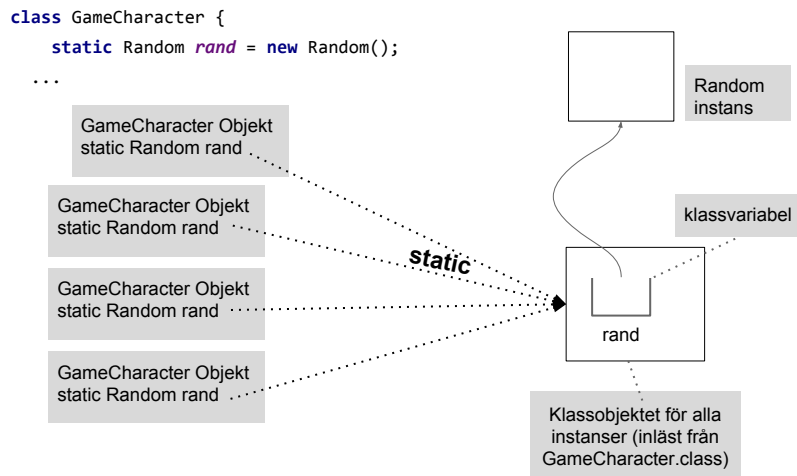
```
class GameCharacter {  
  
    // Share the Random object !!!  
    final static Random rand = new Random();  
    ...  
    void turnRandom() {  
        dx = 1 - rand.nextInt(3);  
        dy = 1 - rand.nextInt(3);  
    }  
}  
  
// No object needed, use class to access.  
GameCharacter.rand.nextInt(4);
```

26

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



27

Alla static variabler och metoden 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(..)
```

28

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!
}
```

29

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
```

30

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!
```

31

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.

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;
    ...
}
```

32

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.

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) { ... }
    ...
}
```

33

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.

Eget Arv: Extends

```
public class Pet {  
    private String name;  
    private int age;  
    public String getName() {return name;}  
    public void setName(String name) { this.name = name;}  
    public int getAge() { return age;}  
    public void setAge(int age) { this.age = age;}  
}  
  
public class Dog extends Pet {  
    ...  
}  
  
public class Cat extends Pet {  
    ...  
}  
  
Dog d = new Dog(...);  
Cat c = new Cat(...);  
out.println(d.getName()); // Call inherited methods  
out.println(c.getAge());
```

38

Alla klasser ärver vissa metoder implicit från klassen Object. Men vi kan även låta våra egna klasser ära från varann.

- Genom att ange extends vid klassdeklarationen kommer klassen **(subklassen)** att ära allt (icke-privat) från en annan angiven klass **(superklassen, basklassen)**
 - D.v.s. i klasserna Cat och Dog kan vi använda alla (ärvda) metoder från Pet.
 - Konstruktörer ärvs inte.
- För att komma åt de privata instansvariablerna måste vi använda set/get (vilka skall vara publika)
- Denna typ av arv kallas **implementationsarv** eftersom vi ärver körbar kod.
- Vi använder denna typ av arv för att undvika redundant kod!
 - Det som är gemensamt för flera klasser bryts ut och samlas i en basklass (metoder/instansvariabler).
 - Därefter får alla klasser som behöver det ära basklassen.
- Java kan bara använda implementationsarv från en superklass (man kan bara använda extends en gång!)
 - Java har inte multipelt implementationsarv.

Abstrakta Basklasser

```
public abstract class Pet {  
    private String name;  
    private int age;  
    public String getName() {return name;}  
    ...  
}  
  
public class Dog extends Pet {  
    ...  
}  
  
Dog d = new Dog(); // Ok  
Pet p = new Pet(); // No!
```

35

Ofta är det inte tänkt att man skall kunna skapa instanser av basklassen

- Vi skapar Dog och Cat-objekt men inte Pet-objekt (ett sällskapsdjur är inte ett djur, det är en klassifikation)!
- För att man inte skall kunna skapa objekt anges att klassen är **abstract**!
 - Kan inte använda new-operator på en abstrakt klass.

Ett annat fall då man måste deklarerera klassen som abstrakt är då den innehåller en abstrakt metod.

- Se Metoder.

super

```
public class FacingCreep extends
    AbstractFacingCreep {

    @Override
    public void move() {
        super.move(); // First run move in superclass
        facing = getDir(); // Then run this
    }

}
```

36

Super, i koden, förekommer bara i samband med arv och syftar på basklassobjektet.

- I detta fallet har vi en överskuggad metod men vill i denna köra basklassobjektets metod först.
- Super är inte en referens (vilket ju this är)

Konstruktörer och super(...)

```
public class Pet {  
    private String name;  
    private int age;  
    public Pet(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}  
  
public class Dog extends Pet {  
    public Dog(String name, int age) {  
        super(name, age);  
    }  
}  
  
Dog d = new Dog("Fido", 3);
```

37

Om superklassen har en konstruktor med parametrar måste subclassens konstruktor anropa denna.

- För att initiera superklassens instansvariabler.
 - Vi kan se det som att superklassobjektet är en del av subclassobjektet
- **super(..)** syftar på den direkta superklassens konstruktor (jämför this(...)).
 - Måste stå först i subclassens konstruktor.
- Detta innebär att i en arvshierarki så initieras klasserna uppifrån ner (alltid superklasser först)
 - Jämför också static och initiering!

Standard för Konstruktorer

```
// Evil game character
public class Creep extends AbstractCreep {

    // Constructor, setting all values (set some default in super)
    public Creep(Path path, double width, double height) {
        super(path, width, height, 1, 30, 10, 10);
    }

    // Overloaded constructor
    public Creep(Path path) { // Remove some params to simplify
        this(path, 5, 5);    // Call another constructor with
                             // default values
    }
}
```

38

Ett vanligt mönster för konstruktorer är att:

- Ha en baskonstruktor som sätter alla värden (och ev. anropar super)
- Ha en eller flera konstruktorer med förvalda värden (av bekvämlighetsskäl).
 - Dessa konstruktorer anropar alltid baskonstruktorn.

final Klasser

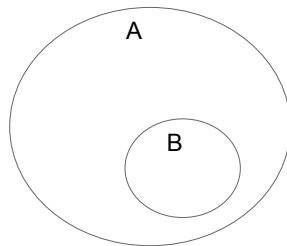
```
public final class Dog extends Pet {  
    ...  
}
```

Klasser som är final kan man inte ärva. En del av Java's klasser är final t.ex. String

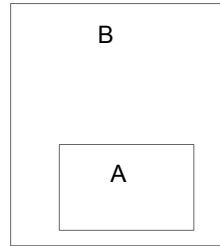
- enum är alltid final

Olika Syn på Arv

$B <: A$



Som typer



Som objekt

40

Antag B är subtyp till A ($B <: A$)

Utifrån typer är en subtyp en delmängd till supertypen.

- Det finns fler operationer för subtypen.

Utifrån objekt finns ett delobjekt av typ A i objektet B

- Man kan anropa alla metoder för A objektet på ett B objekt.

Gränssnitt

```
public interface MyList ... {  
    boolean isEmpty();  
    boolean add(int i);  
    int get(int index);  
}
```

Ett **gränssnitt** ([interface](#)) är en samling av publika abstrakta metoder (public abstract behöver inte anges)

- Kan inte instansiera ett gränssnitt (med new) eftersom metoderna är abstrakta.
- Ett gränssnitt introducerar en referenstyp
 - De operationer som är tillåten för typen är de metoder vi angivit.
 - Innebär att vi kan deklarera en referensvariabel med gränssnittstypen

I bilden deklareras ett gränssnitt MyList..

Implementera ett Gränssnitt

```
public class MyListImpl implements MyList {  
    @Override  
    boolean isEmpty(){ ... }    // Method bodies here  
    @Override  
    boolean add(int e) { ... }  
    @Override  
    int get(int index){ ... }  
}  
  
// Variable interface type, object as implementing type  
MyList list = new MyListImpl() ;  
list.add(4);    // Ok, method in interface type
```

För att få körbara kod för metoderna deklarerade i gränssnittet låter man någon klass implementera (implements) gränssnittet.

- Innebär att alla metoder från gränssnittet måste implementeras i klassen.
 - Kontrolleras av kompilatorn.
 - Vi anger @Override så att kompilatorn kontrollerar att vi använder samma metodhuvud som i gränssnittet.
- Att implementera ett gränssnitt kallas **gränssnittsarb** (vi ärver ingen körbar kod)
 - Klassen blir en subtyp till gränssnittet d.v.s. vi kan tilldela objekttypen till en variabel av gränssnittstypen.
 - Eftersom klassen kan minst lika mycket som gränssnittet anger.
- Vi har på detta sätt fått ett eller flera olika objekt som kan utföra operationerna givna i gränssnittet.
 - Vi kan alltså välja mellan olika implementationer, kan t.o.m. välja under körning!
 - Att använda parametrar och returtyper av gränssnittstyp ger oss flexibilitet.
 - Det kan finnas flera objekt som implementerar gränssnittet.
 - .. vi kan välja det som passar utan att behöva ändra

- typer för parametrarna.

I Java kan en klass implementera flera gränssnitt, för gränssnittsarb galler multipelt arv.

- Klasser kan dessutom ha både extends och implements i deklarationen.