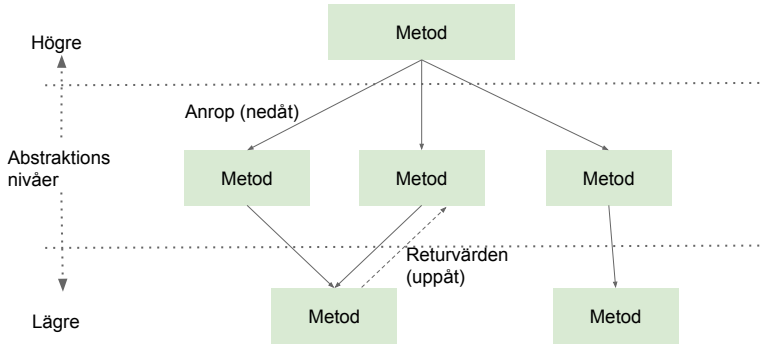
Metoder

TDA548/Joachim von Hacht

Konceptuell bild



2

En metod är en avgränsad del av ett program som utför en viss uppgift

- Ger ofta ett visst utvärde/resultat (motsvarar matematisk funktion) …
- … men inte alltid.

Att använda metoder i ett program ger många fördelar

- Programmet får en struktur, programmet blir greppbart
  - Om ett program överstiger ett par hundra rader börjar det bli ohanterligt ..
  - … genom att strukturera programmet m.h.a. metoder kan vi behärska komplexiteten.
- Vi kan bygga upp programmen bit för bit
  - Man skriver och testar en metod i taget.
- Metoder har namn!
  - Programmet blir lättare att förstå om vi inför begrepp på en högre nivå (säger mer vad det handlar om)
- Metoder kan återanvändas, innebär att vi undviker upprepad kod (redundans)!!
  - Vill vi köra samma kod på flera ställen, anropar vi samma metod
- Innehållet i en metod kan ändras utan att resten av programmet behöver ändras.

I bilden: Metoder arbetar på olika abstraktionsnivåer:

- Metoderna på låg abstraktionsnivå är mycket detaljerade (använder ofta primitiva typer)
- På nästa nivå är de mer övergripande och åstadkommer mer (arbetar med en större del av problemet).
- OBS! Metoder på högre nivå anropar de på lägre. Returvärden skickas från lägre nivå till högre.
- Kallas en skiktad design, mer i senare kurser.
- För att åstadkomma diagrammet, se: Bildserie Arbetssätt och Programkonstruktion

Metoddeklaration

```
// Declaration of method named add
// Parameters int a and int b
// Return type int
int add( int a, int b ){   // Head
  return a + b;            // Body
}
```

4

Man skapar en metod m.h.a. en **metoddeklaration**

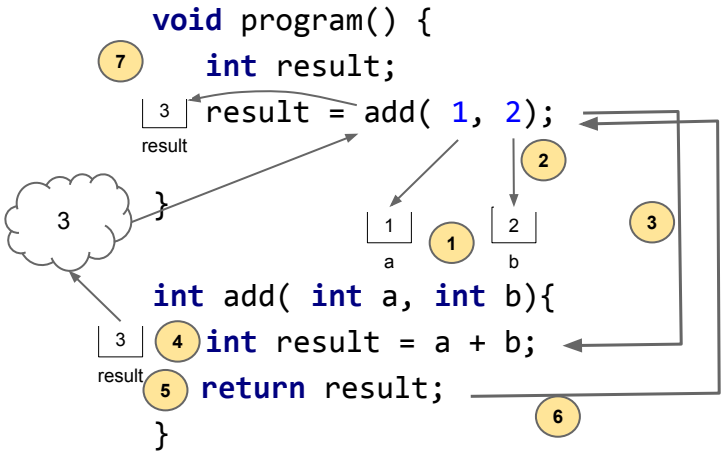
- Första raden i deklarationen kallas metoden **huvud** (head)
  - I huvudet anges (vänster -> höger): Returtyp, namn och en **parameterlista** inom parentes
    - Returtypen anger typen på värdet som metoden returnerar (om något, mer senare …)
    - Namnet väljer vi själva (efter de regler som finns för namn)
      - Namnet skall vara lagom långt och beskriva vad metoden skall göra, ofta är ett verb inblandat
      - Metodnamn inleds med liten bokstav därefter camelCase.
    - I parameterlistan deklareras ett 0-n variabler (parametrarna).
      - Dessa används för att ta emot inkommande data vid anropet (under körning).
      - För varje parameter anges typ och namn (även här väljer vi namn)
- Koden i blocket efter huvudet kallas metodens **kropp**. I kroppen skrivs den kod som skall köras då metoden anropas
- En **return**-sats i kroppen används för att avsluta metoden och skicka tillbaka värdet som står efter return.

- Om det står ett uttryck efter beräknas detta först
- Returtypen i huvudet och typen på uttrycket som returneras måste vara kompatibla, annars typfel
- Om vi anger en returtyp måste vi returnera ett värde annars kompileringsfel
    - Kompilatorn kontrollerar att det garanterat finns en return-sats som kommer att köras
    - Om man t.ex. har en if-sats i metoden måste man ev. ha flera return-satser
- En metod kan bara returnera ett värde!
    - Värdet kan dock var sammansatt t.ex. en array

Program innehåller normalt många metoddeklarationer

- I vilken ordning deklarationerna skrivs i programmet spelar ingen roll (de får inte vara nästlade)
- Vi lägger inledningsvis alla metoddeklarationer i slutet av programmet

Metodanrop



6

Att exekvera metoden, att **anropa** den, görs genom att skriva metodens namn och aktuell indata inom en parentes.

- Indatan kallas **argument** (fast i JLS kallas de formella parametrar)
- Argumenten måste matcha parametrarna (antal, position, typkompatibla) annars kompileringsfel.
- Om argumenten är uttryck som behöver beräknas gör detta först (ev. implicita typomvandlingar görs också). Beräknings sker v->h.

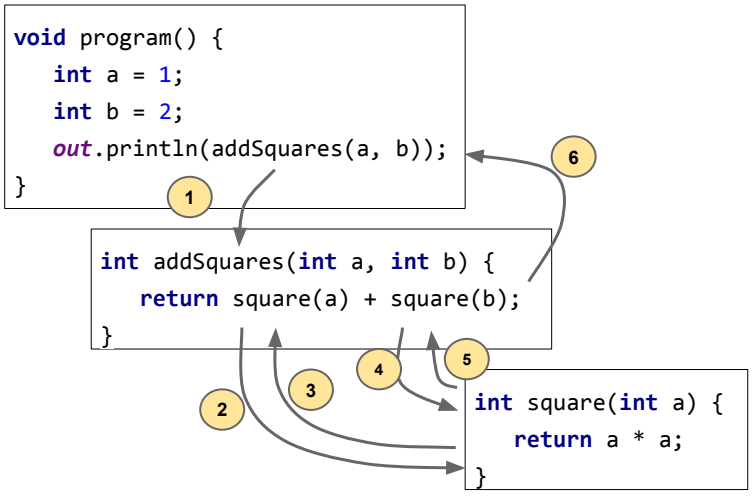
Följande sker vid anropet av metoden (detta skall ni kunna utantill!)

1. Variablerna i metoden (inkl. parametrar) skapas i en del av minnet kallat programmet **anropsstack (call stack).**
2. Argumentenvärden kopieras till metodens parametrar (utifrån position).
   - Kallas **värdeanrop (call by value).** Så sker alltid i Java
3. Därefter sker ett hopp, från aktuell (påbörjad) sats till den första satsen i metoden
4. Metoden exekveras sats för sats till en return-sats påträffas
5. Vid return-satsen kopieras returvärdet till en tillfällig lagringsplats, därefter frigörs minnet på anropsstacken, d.v.s. variablerna i metoden försvinner!!!
6. Programmet hoppar tillbaks till den påbörjade satsen (återhopp)
7. Om vi inte tilldelar returvärdet kommer det att försvinna då nästa

1. sats körs
2. I koden i bilden gör vi en tilldelning, vi sparar värdet. Typen på returnerat värde och variabeln måste vara kompatibla, annars typfel

OBS! Att metoder alltid jobbar med kopior av värden..

Metodanrop från Metod



En metod hoppar alltid tillbaks till satsen där den anropades

- Kan ske i flera steg, om en metod anropar en annan
- Anropsstacken används då till att hålla reda på i vilken metod programmet är och vart det skall hoppa då metoden är klar
- I IntelliJ kan man inspektera exakt vad som händer men anropsstacken (i debuggern)

Numeriska Parametrar och Returtyper

```
// Convert Fahrenheit to Celsius
double f2c(double fahrenheit){
   return (fahrenheit - 32) * 5 / 9;
}

// Absolute value (NOTE: 2 return statements)
int abs(int n) {
   if (n < 0) {
       return -n;
   }
   return n;
}
```


Används för olika typer av beräkningar.

Nästlade metodanrop

```
out.println(sqrt(pow(3, 2) + pow(4, 2)));
```


Det går att skicka returvärdet från en metod direkt som argument till en annan metod

- Kallas **nästlade anrop** (nested calls)
- Ibland användbart … slipper en del "onödiga" variabler för returvärden
  - Speciellt vid utskrifter
- … dock svårt att felsöka, allt sker i ett enda svep

Parametrar evalueras vänster till höger

- Men koda aldrig så att man blir beroende av detta

Boolesk Returtyp

```
// Boolean method (NOTE: No if statement needed)
boolean isGameOver(int score1, int score2){
    return score1 >= 10 || score2 >= 10
                        && score1 != score2;
}
```


Booleska metoder används för att svara på ja/nej frågor

- Ofta bara en rad med ett (komplext) boolesk uttryck
- Namnges som en ja/nej-fråga, hasWinner(), isEven(), isLeapYear()
- Användbara, höjer abstraktionsnivån, döljer rörig kod

Sträng som Returtyp

```
final Scanner sc = new Scanner(in);

// Get user name (NOTE: No variable used)
String getName() {
   out.print("Please enter your name > ");
   return sc.nextLine();  // Return result at once
}
```


Här i kombination med IO.

- Mer senare.

void-metoder

```
void roundMsg(int result, int computer, int statistic) {
   out.println("Computer choose: " + computer);
   if (result == draw) {
       out.println("A draw");
   } else if (result == humanWin) {
       out.println("You won");
   } else {
       out.println("Computer won");
   }
   out.println("Result " + statistic); // No return!
} // Jump back
```

10

void-metoder

- Man kan ange att en metod inte returnerar ett resultat …
- … görs genom att ange **void** istället för returtyp
- Typiskt funktioner som "bara gör något", skriver ut till skärmen t.ex.
- I övrigt fungerar void-metoder som andra metoder
  - Återhopp sker då sista krullparentesen nås.
- Metoden får inte innehålla en return-sats med ett värde efter
  - Däremot bara return går bra, innebär att man avslutar metoden
    - Man kan alltså avsluta "mitt i" en metod
    - Gäller även för icke-void metoder , undvik (men ok vid vissa tillfällen)!
- Eftersom void-metoder inte returnerar något, representerar de inte något värde
  - … de är inte uttryck (de är satser)
  - Kan inte stå t.ex. vid tilldelning

Array som Parameter

```
// Find max value in array
int max(int[] arr) {
   int m = arr[0];
   for (int i = 1; i < arr.length; i++) {
       if (arr[i] > m) {
           m = arr[i];
       }
   }
   return m;
}
```

Objekt som Parameter

```
// Print complete dog
void printDog(Dog dog) {
   out.print("Name: " + dog.name);
   out.println(" Age:" + dog.age);
}

// Class declaration
class Dog {
   String name;
   int age;
}
```

Array med Objekt som Parameter

```
Dog findOldest(Dog[] dogs){
   int index = 0;
   int maxAge = dogs[index].age;
   for( int i = 0 ; i < dogs.length ; i++){
       if( dogs[i].age > maxAge){
           index = i;
           maxAge = dogs[i].age;
       }
   }
   return dogs[index];
}
```


Kan skicka komplexa argument

- Här en parameter för en array med Dog-objekt

Array med Objekt som Returtyp

```
Dog[] getDogs(){
    Dog[] dogs = { new Dog(), new Dog() };
    dog[0].name = sc.nextLine();
    dog[1].name = sc.nextLine();
    return dogs;
}
```


Komplex returtyp

- Här en array med Dog-objekt som returvärde.

Synlighetsområde

```
{
 int i = 0;

}

//Ok, other scope
{
 int i = 2;

}
```

```
{

 int i = 0;

    {
        int i = 2; // Name!
        int j = 3;
    }


    j = 4; // Not visible
}
```

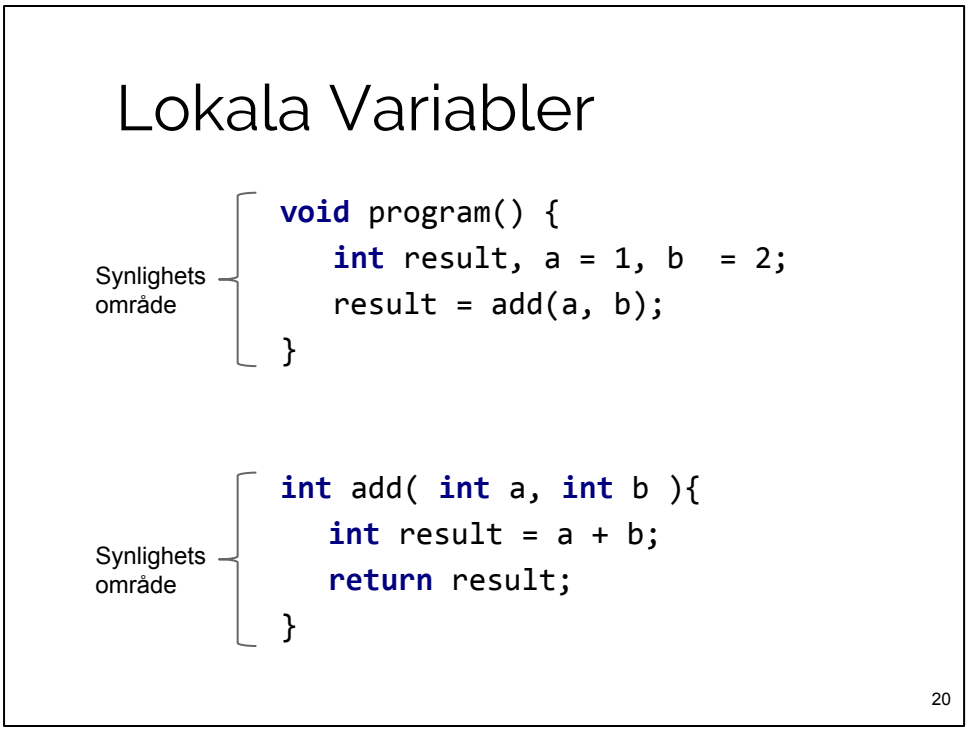
18

Synlighetsområdet (scope) anger var i programmet det är möjligt att använda en variabel

- I Java sammanfaller synlighetsområde och block (förenklat)

Följande gäller

- Variabler är bara synliga (kan bara användas) i det synlighetsområde de är deklarerade (fr.o.m. deklarationen och vidare), förutom …
  - … nästlade synlighetsområden
    - Ett inre block kommer åt variabler i ett yttre (som är deklarerade innan det inre blocket)
    - Yttre block kommer inte åt variabler i ett inre
- I Java gäller att variabler inte får ha samma namn inom samma synlighetsområde
- Inom olika synlighetsområden kan samma namn användas
  - Mycket praktiskt: Slipper hitta på nya namn hela tiden!



Variabler deklarerade i metoder kallas **lokala variabler**

- Vi räknar även parametrar som lokala variabler
- Synlighetsområdet är metodkroppen (ett block)
- Lokala variabler har en livslängd, de lever på anropsstacken och förstörs då programmet lämnar metoden.
- Lokala variabler måste ges ett värde (initieras eller tilldelas) innan de används, om ej kompileringsfel
- Inledningsvis tillåter vi bara lokala variabler i våra program! (d.v.s. inga variabeldeklarationer utanför metoder)

Bilden:

- "result" i koden ovan syftar på två olika variabler med samma namn, den ena i program() den andra i add()
- På samma sätt med a och b.
- Eftersom de finns i olika synlighetsområden kan vi använda samma namn!
- I exemplet finns totalt 6 variabler (2 st a, 2 st b och 2 st result)

Anm: I program() ovan deklarerar vi flera variabler på samma rad

- Möjligt att göra men inte så bra, bättre med en/rad (görs av utrymmesskäl här)

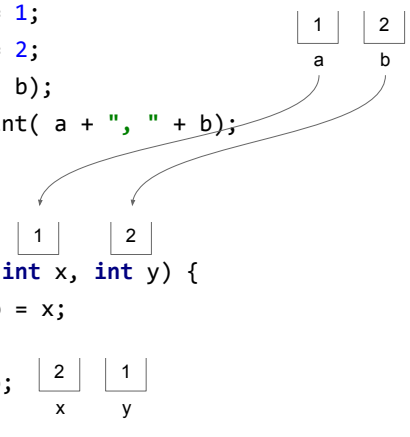
Summa:

- Samma variabelnamn inom olika synlighetsområden syftar på på olika variabler
- Olika variabelnamn (ev inom olika synlighetsområden) kan syfta på samma sak, om referenser är inblandade, se Referenser

Variabler utanför Metod

```
void program() {
   int a = 1;
   int b = 2;
   swap(a, b);
   out.print( a + ", " + b);
}

void swap(int x, int y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```



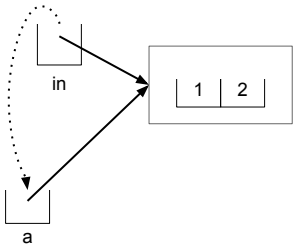
En metod kan aldrig komma åt lokala variabler i en annan metod.

- Antag att vi vill skriva en metod, swap, som byter plats på argumenten!
- När vi anropar metoden swap kopieras värdena i a och b till parametrarna (lokala variabler), därefter byter dessa värden
- … inget händer med a och b i program(),… swap kan aldrig komma åt a och b.
- Det går inte att skriva metoden swap med primitiva typer
  - Däremot kan metoder komma åt objekt “utanför” … om parametrarna är referenser.

Mer om Array-parametrar

```
void program() {
   int[] in = { 1,2 };
   zero(in);
    // in is now [ 0, 0 ]
}


void zero( int[] a ){
   for(int i = 0; i < a.length ; i++){
       a[i] = 0;
    }
}
```



Om en metod har en array-parametrar sker precis samma sak som vanligt men …

- … argumentet som kopieras är en referens!
- Parametern kommer därför att peka på samma objekt som argumentet!
- Metoden kan ändra på ett objekt utanför sig självt (variabeln “in” kan vi däremot aldrig ändra)

I koden ovan kommer array:en som variabeln in refererar att vara förändrad efter metodanropet

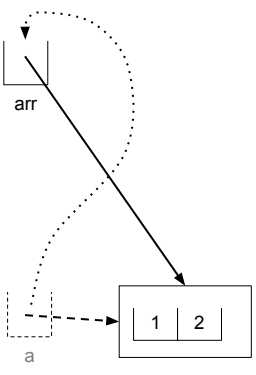
- Kallas **utparameter**
- Innebär en viss risk eftersom det kan vara svårt att inse att en metod har ändrat i objektet (eftersom den är void)

Mer om Array som Returtyp (1)

```
void program() {
   int[] arr = getArray();

}


int[] getArray(){
    int[] a = { 1, 2 };
    return a;
}  // a gone here
```



Om man returnerar en array sker samma sak som vid ett vanligt metodanrop

- Dock är <u>returvärdet en referens</u>

Vi kan skapa en array i en metod och skicka som returvärde.

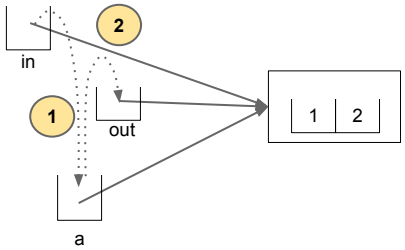
- Den lokala variabeln a försvinner! …
- … men inte array-objektet
- För att kunna komma åt array-objektet måste vi spara den returnerade referensen

Mer om Array som Returtyp (2)

```
void program() {
   int[] in = { 1,2 };
   int[] out = zero(in);

}

int[] zero( int[] a ){
   for(int i = 0; i < a.length ; i++){
       a[i] = 0;
    }
    return a;  // Return parameter!
}
```



Här returnerar vi samma referens som vi skickar in.

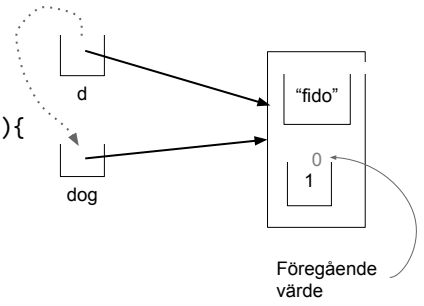
- Kan i vissa fall ge smidigare kod.

Mer om Objekt som Parameter

```
Dog d = new Dog();
incAge(d);

void incAge( Dog dog ){
   dog.age++;
}

class Dog {
   String name;
   int age;    // 0 default
}
```



Eftersom referensen till objektet kopiera till metodens parameter kommer den åt objektet (utanför metoden)

- På samma sätt som för en array
- Samma problem som för en array

Gäller även strängar men dessa kan inte förändras (icke muterbara) så ingen risk.

Överlagrade Metoder

```
// Overloaded (from Math API)
// Get max of two values
double max( double d1, double d2){...}
float max( float d1, float d2){...}
int max( int d1, int d2){...}
long max( long d1, long d2){...}
```

22

Metoder inom samma synlighetsområde får ha samma namn …

- … men då måste parameterlistorna skilja sig åt!
  - Returtypen spelar ingen roll, bara parametrarna räknas.
- Kallas att metoderna är **överlagrade** (overloaded)
  - Innebär att redan vid kompileringen väljs vilken metod som skall anropas vid körningen.
  - Vilken metoden som väljs beror på parametrarna(s) deklarerade (statiska) typer.
    - Implicita typomvandlingar kan förekomma för att hitta matchande metod
  - Jämför +-operatorn vars beteende beror på operanderna!
- Tanken med överlagrade metoder är att man skall slippa hitta på nya namn på metoder som gör "samma sak" men med olika antal eller parametertyper.
  - Metoder som gör olika saker skall inte ges samma namn, mycket viktigt med bra namn (verb)!

Generiska Metoder

```java
// T is the type variable NOTE: Method doesn't use the element objects
<T> int find(T[] arr, T value) {
   for (int i = 0; i < arr.length; i++) {
       if (arr[i] == value) {    // Reference identity!
           return i;
       }
   }
   return -1;
}

// Will convert elements to wrapper type
Integer[] ints = {1, 4, 6, 2, 7, 0, -1};
String[] strs = {"aaa", "xxx", "fff", "ccc", "ddd"};
out.println(find(strs, "aaa") == 0);
out.println(find(ints, 7) == 4);     // Parameters boxed to Integer
```

23

Ibland blir det klumpigt med överlagring, ett alternativ är Generiska metoder.

Generisk metoder kan ta vilken referenstyp som helst som parameter och returtyp

- Haken är att man inte kan göra något med själva objektet, man kan bara arbeta med referenserna t.ex. flytta runt dessa på olika sätt.
    - Finns lösningar, mer i andra kurser.
- Man anger att metoden är generisk m.h.a. av en typparameter inom vinkelparenteser, normalt kallad T, först av allt i metodhuvudet (före returtypen)
- Generiska metoder kan inte ta primitiva typer
    - För att lösa detta kan man använda omslagstyper, se Typer

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

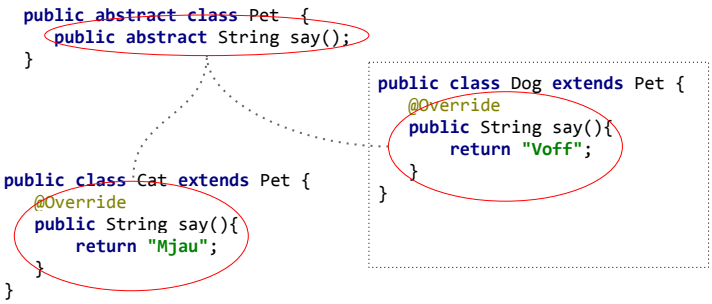
```
public abstract String say();
```

27

En abstrakt metod saknar metodkropp (ingen körbar kod).

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

Överskuggade Metoder




```
Pet p = ...;
out.println(p.say()); // Depends on object type!
```


En **överskuggad (overriden)** metod förekommer bara i samband med arv

- Innebär att man ersätter en ärvd metod från en superklass med en egen version av metoden (i subklassen).
- Vilken metod som körs bestäms under körning utifrån objektets typ (inte typen på referensvariabeln).
  - Kallas **sen bindning** (late binding)
  - Ordet **polymorfism** används också, syftar på att objekten beter sig olika utifrån sin typ
- För att det skall fungera måste metoden ha exakt samma metodhuvud som den ärvda, även returtyp (förutom namn på parametrar)
  - @Override gör så att kompilatorn kontrollerar att metodhuvudet matchar det ärvda …. annars blir det överlagring (vi vill inte, av misstag, åstadkomma överlagring).
- Flera olika metoder (i olika subklasser) kan överskugga metoden.

Varför Överskuggade Metoder?

```
List<Pet> pets = ...;

// Assume NO say()-method in classes
// Non overriding style, if new Pet added have to change code
for (Pet p : pets) {
   if (p instanceof Cat) {
       out.println("Mjau");
   } else if (p instanceof Dog) {
       out.println("Voff");
   }
   ...
}

// Assume overridden method say() in all classes
// Each Pet know what to say, if new pets no change to code!
for (Pet p : pets) {
   out.println(p.say());
}
```


Överskuggning gör att vi kan skriva mer abstrakt kod, ger flexibilitet.

- Vi kan lägga till nya objekt utan att ändra i koden (if-satsen i bilden).
- Objekten vet själva vad som skall göras då någon metod anropas.

Metoder med Sidoeffekter

```
// Intuitively 0 ... (?)
out.println( o.m(0) - o.m(0) );
```

35

Tidigare haft uttryck med sidoeffekter (x++ t.ex.)

- Samma fenomen kan uppträda för instansmetoder.
- Innebär i vårt fall att metoden förändrar en instansvariabel i objektet
- Dvs: Om en metod returnerar ett resultat och på samma gång ändrar en instansvariabel så har vi en metod med sidoeffekt
    - ** Undvik **!

Referentiell transparens (? svenska begrepp saknas, referential transparency)

- Enkelt sagt: Givet samma indata, får man alltid samma utdata från metoden?
    - Om så är det lättare att resonera om program (korrekthet)
- Metoder med sidoeffekter innebär att programmet inte blir referetiellt transparent
    - Kan inte undvikas i imperativ programmering, …
    - … men man kan försöka minimera
    - Metoder med returvärden undviker att ändra instansvariabler
    - void-metoder ändrar ofta instansvariabler, men metoderna är inga uttryck, vi får inget värde..
- Försök alltid att skriva metoder som bara tar indata och returnerar utdata …
    - …om tvunget, använd instansvariabler.

- Du behöver instansvariabler om objektet måste komma ihåg något mellan metodanropen.

Stack Overflow

```
void program() {
   program();  // Oh, ooh
}
```

```
/home/hajo/devel/jdk1.8.0/bin/java ...
Exception in thread "main" java.lang.StackOverflowError
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
```

Vid varje anrop skapas lokala variabler på anropsstacken

- Vad händer om en metod anropar sig självt?
- … om inget händer som stoppar metoden får vi StackOverflowError
- … allt minne för anropsstacken är fyllt.

Rekursiva Metoder

Matematisk definition

$F_n = F_{n-1} + F_{n-2}$ (1, 2, 2, 3, 5, 8, 13, ...)

$F_1 = 1, F_2 = 1$

```
// Recursive method
int fibonacci(int n) {
   if (n <= 1) {
       return n;
   } else {
       return fibonacci(n - 1) + fibonacci(n - 2);
   }
}
```

32

En rekursiv metod är en metod som anropar sig själv

- För att inte få StackOverflow måste argumenten förändras vid varje anrop.
- I bilden: n minskas hela tiden och blir förr eller senare 1 (då sker inga fler anrop)

Rekursiva metoder kan ibland ge eleganta lösningar till knepiga problem

- Ofta då man har mer komplicerade datastrukturer (grenande)
- Kan ibland direktöversättas från matematiska definitioner
- I Bilden: Rekursiv variant av fibonacci talen (1,1,2,3,5,8, …)
    - … tyvärr ineffektiv. Många metodanrop!