# Metoder

*En klass beteende*

Tobias Andersson Gidlund

tobias.andersson.gidlund@lnu.se

# Agenda

Modularisering
- Allmänt om uppdelning
- Divide and Conquer

Metoder i Java
- Struktur
- Icke-värdereturnerande
- Inparametrar
- Värdereturnerande metoder
- Call-by-value och Call-by-reference
- Överlagring

Avslutning

# Modularisering med Metoder

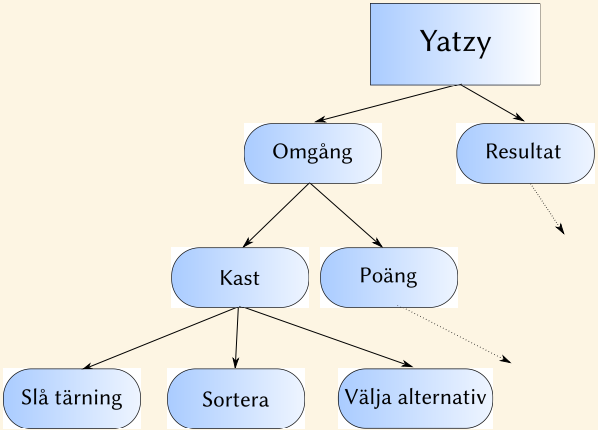
# Om uppdelning

- Det är möjligt att lösa nästan alla programmeringsproblem genom att skriva kod som bygger på sekvens, selektion och iteration.
- Ibland är det dock inte speciellt effektivt eller smidigt att göra på det sättet.
- Därför har man i de flesta språk en möjlighet att dela upp kod i olika moduler.
  - Det kan vara saker som logiskt hör samman, som klasser.
  - Dessutom kan det vara funktionalitet som man vill upprepa, till exempel medelvärdesberäkning.
- Java ger också möjlighet till fler sätt att dela upp koden, med de ovanstående är de viktigaste.

# Divide and Conquer

- Meningen med en modul är att gruppera en mängd med satser i ett program för att utföra en specifikt uppgift.
- Varje ”delprogram” utför alltså en mycket väl avgränsad uppgift – och bara den.
- När ett problem delas upp i flera mindre delar kallas strategin för *divide and conquer*.
  - Dela och övervinna.
- I stort innebär det att man delar upp *hela* problemet i mindre *delar*, där varje del är enkel att implementera.
  - Det gör det också enklare förstå vad som är lösningen på problemet.
- Det är sällan en god idé att försöka lösa hela problemet på en gång!

# Dela upp Yatzy (en början)



# Fördelar med att dela upp koden

- Enklare kod
  - Varje modul blir enklare att förstå då den innehåller färre satser med klarare uppgift.
- Återanvändning av kod
  - Det är enklare att återanvända kod eftersom koden kan anropas flera gånger.
- Enklare tester
  - Eftersom funktionaliteten är begränsad så är det enklare att isolera och hitta fel.
- Snabbare utveckling
  - Koden kan användas till fler projekt, till exempel kan en kod för användarhantering skapas och återanvändas.
- Enklare att jobba i lag
  - Eftersom uppgifterna är väl avgränsade i modulerna är det enklare att lägga ut dem på olika programmerare.

# Nackdelar med att dela upp koden

- Det viktigaste är att säga att i stort finns inga nackdelar!
- I början kan det dock vara besvärligt att veta hur och när kod ska delas upp.
- Det tar också lite tid att förstå konceptet med att skicka data till och från moduler.

# Operationer och metoder

- Java är ett objektorienterat programmeringsspråk där ett program består av ett antal klasser.
  - Senare kommer kursen gå igenom hur fler klasser skapas än den som håller `main`
- Den här föreläsningen kommer att handla om *beteendet* för en klass (och endast en klass så här långt).
- Inom objektorientering så kallar vi klassens beteende för dess *operationer*.
- När man i stället talar om objekt så kallas det *metoder*.
  - En metod är alltså instansieringen av en operation.
- Varje program som har skapats så här långt har varit en klass med ett namn och en enda metod – `main`.
- Föreläsningen kommer att visa att fler metoder kan skapas för en klass.

# Metoder I Java

# En metods delar

- En metod består av följande:
  - Ett **namn** som tydligt ska återspegla vad metoden gör.
  - En **metodsignatur** som visar vilken data som kommer in respektive återlämnas.
  - En **metodkropp** som är ett antal satser som utför metodens uppgift.
- Vissa metoder måste även visa vad som skickas tillbaka och det placeras i så fall i metodkroppen.
- För att en metod ska kunna exekveras så måste den *kallas* på någonstans i den övriga koden.
- När en metod kallas så skickar man med eventuella variabler till den och tar hand om returvärdet ifrån den (om något).

# Övergripande om en metod

- Generellt har en metod följande struktur i Java:

```
synlighet returtyp metodnamn(inparametrar)
```

- *Synlighet* kallas också *modifierare* och de kan vara flera.
- En metod kan vara antingen *värdereturnerande* eller *icke-värdereturnerande*.
- Det anger om metoden skickar tillbaka någon form av data eller inte till den punkt där den kallades.
- Ett exempel i Java är:

```
public void visaOK();
```

- Ovanstående är en publik metod som inte returnerar något och inte tar några parametrar.
  - Det reserverade ordet `void` betyder ungefär ”tomrum”.
- Notera också att metoder, enligt praxis, ska börja med liten bokstav.

# Synlighet/modifierare

- I programmeringsspråk som inte är objektorienterade så behövs ingen ”synlighet”.
- Inom objektorientering så talar synligheten om hur metoden kan nås:
  - `private` innebär att metoden enbart är möjlig att kalla inom ett objekt, från en annan metod.
  - `public` är när metoden kan nås av andra objekt i programmet.
  - `protected`-metoder kan nås av alla objekt i en hierarki eller i samma paket.
- En annan modifierare är `static` (statisk) vilket innebär ”på klassnivå”.
- För tillfället kan metoder sättas som publika och statiska.
- Synligheten och modifierare kommer att diskuteras närmare i föreläsningarna om objektorienterad programmering.

# Icke-värdereturnerande metoder

- En metod som inte returnerar något gör i stället något aktivt, till exempel skriva ut något.
- För att visa det används alltså returtypen `void`.
- När metoden kallas flyttas sekvensen till den plats där metodkroppen börjar.
  - Metodkroppen innesluts av klamrar eftersom det är ett block.
- När alla satser har utförts så avslutas metoden och kontrollen återgår till där anropet gjordes.
- Eftersom anropet ska göras direkt från `main`-metoden, så används också modifieraren `static`.
  - Som disktuerades tidigare och som kommer förklaras ytterligare senare.

# Exempel

- Metoden nedan kommer, när den senare anropas, att skriva ut en hälsning på skärmen.

```
public static void visaSalutation() {
    System.out.println(”May the Force be with You!”);
}
```

- Den är alltså *publik* och på *klassnivå*.
- Den returnerar ingenting vid anrop, visas med *void*.
- Metoden heter `visaSalutation`
- Den tar inga parametrar, dvs ingen information kommer *in* till metoden.

# Anropa en metod

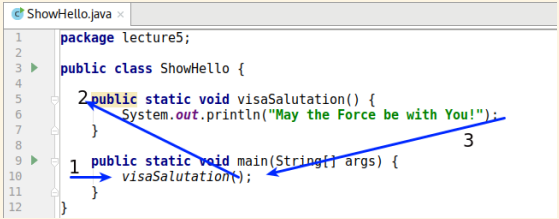
- En metod måste alltid kallas från någon annan plats i koden.
- Notera att det inte finns något som hindrar att en metod anropar en annan metod.
  - Eller ens sig själv – det kallas rekursion men tas inte närmare upp i den här kursen.
- Ett anrop till en icke-värdereturnerande metod görs genom att skriva namnet följt av parenteser.

```
visaSalutation();
```

- Den här metoden tar inte heller några parametrar, så inget behöver skrivas inom parenteserna.
  - Mer om det alldeles snart!

# Hur det fungerar

- I `main` görs anropet och sekvensen fortsätter i metoden.
- När metoden är slut, återvänder sekvensen till `main`.



# Ytterligare exempel

```
package lecture5;

public class ShowMoreHello {
    public static void visaSalutation() {
        System.out.println(”May the Force be with You!”);
        System.out.println(”Meaning of life, the universe and everything: 42”);
    }

    public static void main(String[] args) {
        visaSalutation();
        System.out.println(”Wish You Were Here”);
    }
}
```

Utskrift:

```
May the Force be with You!
Meaning of life, the universe and everything: 42
Wish You Were Here
```

# Parametrar *till* metoden

- Det är möjligt att skapa metoder som accepterar att data skickas till den när den anropas.
- Möjligheten att förse en metod med olika data vid olika anrop är en av styrkorna.
  - Det innebär att man återanvänder ett *beteende* men inte nödvändigtvis data.
- För att det ska vara möjligt att skicka in data, så måste metoden konstrueras så att den kan ta emot data.
- Det görs genom att ställa upp vilka datatyper som förväntas.
  - De ges även ett namn som är *lokalt* för metoden.
- När anropet görs så måste *exakt* de parametrarna skickas in.

# Metodsignatur för inparametrar

- Mellan parenteserna i metodsignaturen radas de variabelnamn och datatyper som ska användas upp med ett komma mellan dem.

```
public static void enMetod(int tal1, int tal2)
```

- När metoden ovan ska anropas *måste* två heltal skickas med.

```
enMetod(42, 64);
```

- Det måste inte vara heltalsliteraler som ovan (fasta värden), utan även variabler kan användas.

```
int ettTal = 42;
int annatTal = 64;

enMetod(ettTal, annatTal);
```

# Exempel

```
package lecture5;
import java.util.Scanner;

public class ManyHello {

    public static void showSalutation(int number, String salut) {
        for(int i = 0; i < number; i++) {
            System.out.println(salut);
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String hello;
        int numberOfHello;

        System.out.print("Vad vill du säga? ");
        hello = scan.nextLine();
        System.out.print("Hur många gånger vill du säga det? ");
        numberOfHello = scan.nextInt();

        showSalutation(numberOfHello, hello);
    }
}
```

# Utskrift

```
Vad vill du säga? Do what I do. Hold tight and pretend it’s a plan!
Hur många gånger vill du säga det? 3
Do what I do. Hold tight and pretend it’s a plan!
Do what I do. Hold tight and pretend it’s a plan!
Do what I do. Hold tight and pretend it’s a plan!
```

- Notera att eftersom variablerna är *lokala* i metoden, så behöver de *inte* ha samma namn.
- Det viktiga är att den indata som skickas har samma *datatyp*.
- De måste också komma i *samma* ordning som visas i metoden.

# Värdereturnerande metoder

- Många gånger vill man även ha något tillbaka från metoden.
- Observera att en metod endast kan returnera *ett* värde och det måste vara av *en* datatyp.
- När en metod ska vara värdereturnerande så måste `void` bytas mot en specifik datatyp.

```
public static int summa(int a, int b)
```

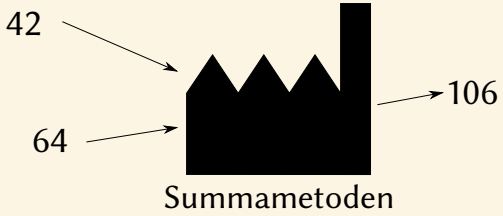
- Datatypen kan vara vilken som helst, men det måste finnas en variabel (eller liknande) som *tar emot* värdet vid anropet.

```
int svar = summa(42, 64);
```

- I exemplet ovan skickas talen 42 och 64 *in* till metoden och (förmodligen) skickas heltalet 106 tillbaka.
- Svaret läggs i variabeln `svar` och kan fortsätta att bearbetas.

# Som en fabrik

- En metod kan ses lite som en fabrik där vi skickar in något och får ut något som har bearbetats.
- Eftersom vi inte alltid har gjort metoderna själva utan använder andras metoder, så vet vi inte heller alltid vad som görs inne i ”fabriken”.
  - Det är som en svart låda för oss, vi vet vad som stoppas in och vi får något tillbaka.



# Metodkroppen för en värdereturnerande metod

- Till skillnad från de icke-värdereturnerande metoderna så måste man uttryckligen säga till när ett värde ska returneras.
- Det görs i metodkroppen med `return` följt av ett värde som motsvarar datatypen som ska returneras.

```
public static int summa(int a, int b)
{
    return a + b;
}
```

- Naturligtvis kan fler satser finns i metodkroppen.
- Det är även möjligt att ha flera `return`-satser, främst tillsammans med selektion.
  - Notera att det dock är viktigt att det *alltid* finns en retursats för *alla* möjliga alternativ.

# Exempel

```
package lecture5;
import java.util.Scanner;

public class Astronomy {

    public static double AUtoKM(double dist) {
        double AU = 149597871.0;
        return AU * dist;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Hur många astronomiska enheter? ");
        double avstand = scan.nextDouble();

        double svar = AUtoKM(avstand);

        System.out.printf("Det blir %1$.2f kilometer.", svar);
    }
```

# Exempel på körningar

```
Hur många astronomiska enheter? 1,0
Det blir 149597871,00 kilometer.

Hur många astronomiska enheter? 2,5
Det blir 373994677,50 kilometer.

Hur många astronomiska enheter? 4,2
Det blir 628311058,20 kilometer.
```

# Anrop direkt

- En värdereturnerande metod kan alltid användas på alla de ställen då datatypen är möjlig.
- Det innebär att den kan slås samman med en sträng eller användas direkt i ett uttryck.
  - Vilket vi tidigare har gjort med `Math.pow()`
- Utskriften i exemplet tidigare skulle alltså kunna skrivas om till:

```
System.out.printf(”Det blir %1$.2f kilometer.”, AUtoKM(avstand));
```

- Utskriften blir exakt samma som tidigare.
- Det är ett bra sätt att minska kodens omfång och till viss del minnesåtgång, men det kan också leda till svårläst kod.

# Exempel med flera retursatser

```java
package lecture5;
import java.util.Scanner;

public class Difference {

    public static int difference(int a, int b) {
        if(a > b)
            return a - b;
        else
            return b - a;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Ange första talet: ");
        int tal1 = scan.nextInt();
        System.out.print("Ange andra talet: ");
        int tal2 = scan.nextInt();

        System.out.println("Skillnaden är "
        + difference(tal1, tal2));
    }
}
```

## Exempel på körning:

```
Ange första talet: 2
Ange andra talet: 10
Skillnaden är 8

Ange första talet: 8
Ange andra talet: 2
Skillnaden är 6
```

# Ytterligare exempel

```
public static int randomNumber() {
    Random rnd = new Random();
    return rnd.nextInt(6);
}

public static String Pronomen() {
    int val = randomNumber();

    if (val == 0)
        return ”Jag”;
    else if (val == 1)
        return ”Du”;
    else if (val == 2)
        return ”Den”;
    else if (val == 3)
        return ”Vi”;
    else if (val == 4)
        return ”Ni”;
    else
        return ”De”;
}
```

```
public static String Verb() {
    int val = randomNumber();

    if(val==0)
        return ”äter”;
    else if (val == 1)
        return ”målar”;
    else if (val == 2)
        return ”smeker”;
    else if (val == 3)
        return ”stryper”;
    else if (val == 4)
        return ”ser”;
    else
        return ”drar”;
}
```

```
public static String Substantiv() {
    int val = randomNumber();

    if (val == 0)
        return ”en bil”;
    else if (val == 1)
        return ”en vän”;
    else if (val == 2)
        return ”ett hus”;
    else if (val == 3)
        return ”ett träd”;
    else if (val == 4)
        return ”en dator”;
    else
        return ”en bok”;
}
```

# Huvudprogrammet samt metod för mening

```
public static String mening() {
    return Pronomen() + " " + Verb() + " " + Substantiv() + ".";
}

public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);

    System.out.print("Hur många meningar vill du ha? ");
    int antal = scan.nextInt();

    for (int i = 0; i < antal; i++)
        System.out.println(mening());
}
```

## Utskrift:

```
Hur många meningar vill du ha? 5
Jag äter en bok.
Jag målar ett hus.
Ni stryper en bok.
Du ser en bil.
Den målar en bil.
```

# Mer om parametrar

- Viktigt att komma ihåg är att variabeldeklarationen i metodsignaturen är *lokal* för metoden.
  - Den är bara synlig i metodkroppen och inte utanför.
- Det är därför som namnet på den variabel som *skickas* till metoden *inte* behöver ha samma namn som den i metoden.
- Om en variabel behöver vara synlig för *alla* metoder deklareras den som ett attribut.
  - Det vill säga direkt under `public class ...` och dess startklammer.
- Undvik att använda för många attribut – de ska bara användas om de definierar klassen!
  - Mer om det i objektorienteringsföreläsningen senare.

# Metodkedjor

- Metodanrop kan kedjas, det vill säga läggas direkt på varandra.
- Det fungerar för att en metod kan anropas direkt på det som returneras.
- I Java 8 finns en klass för tid och datum som är smidigare än det som beskrivs i boken.
  - Klassen heter `LocalTime`.
- Följande betyder att ”på det som returneras från `LocalTime.now()` anropa `getHour()`“:

```
int timme = LocalTime.now().getHour();
```

- Det är exakt samma som att skriva:

```
LocalTime nutid = LocalTime.now();
int timme = nutid.getHour();
```

# Hela koden och exempel

```
package lecture5;
import java.time.LocalTime;

public class TimePlease {
    public static void main(String[] args) {
        int timme = LocalTime.now().getHour();

        if (timme > 6 && timme <= 9)
            System.out.println(”God morgon”);
        else if (timme > 9 && timme <= 11)
            System.out.println(”God förmiddag”);
        else if (timme > 11 && timme <= 13)
            System.out.println(”Lunch!”);
        else if (timme > 13 && timme <= 18)
            System.out.println(”God eftermiddag”);
        else
            System.out.println(”God natt”);
    }
}
```

Utskrift:

```
God förmiddag
```

# Call-by-value

- När en metod anropas med primitiva datatyper så *kopieras* värdet till metoden.

```
difference(tal1, tal2); // Värdet i tal1 kopieras till t1 och
                        // värdet i tal2 kopieras till t2 i metoden.
```

- Det innebär att *inga förändringar* görs av innehållet i variablerna som används vid anropet.
  - Det kopierade värdet kan dock förändras *inne i* metoden, men det påverkar alltså inte ursprungsanropets variabler.
- Det kallas för *call-by-value* eller ibland *pass-by-value*.
  - Det viktiga är att det handlar om ett värde som kopieras och inte en variabel som används.
- Exemplet på nästa sida visar varför det är viktigt att inte använda samma variabelnamn i anropet som i själva metoden, det kan förvirra.

# Exempel

```
package lecture5;
public class Value {

    public static void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;

        System.out.println(”Inne i metoden: a=” + a + ” och b=” + b);
    }

    public static void main(String[] args) {
        int a = 4;
        int b = 2;

        System.out.println(”Innan metoden:  a=” + a + ” och b=” + b);
        swap(a, b);
        System.out.println(”Efter metoden:  a=” + a + ” och b=” + b);
    }
}
```

## Utskrift:

```
Innan metoden:  a=4 och b=2
Inne i metoden: a=2 och b=4
Efter metoden:  a=4 och b=2
```

# Call-by-reference

- Om det är viktigt att värdena verkligen förändras används tekniken *call-by-reference*.
- Med det som har diskuterats hittills så går det inte att visa det.
- I korthet går det ut på att ett *objekt* måste skickas i stället för ett primitivt värde.
  - En ”referens” är i det här fallet är den minnesposition där värdet ligger.
  - Därför viktigt att förstå att Java *aldrig* är call-by-reference egentligen utan det är en *kopia* av minnesadressen som skickas.
- Call-by-reference kommer alltså att diskuteras mer i föreläsningen om objekt.

# Överlagring

- En metod kan förekomma med samma namn flera gånger i en klass, så länge den har *olika* parametrar.
  - Synlighet och modifierare måste vara samma.
  - Returtypen kan vara annan, men det får inte vara den *enda* skillnaden.
- Ett exempel som har används är `System.out.println()`:

```
// använder överlagrad metod med int som parameter
System.out.println(42);
// använder överlagrad metod med String som parameter
System.out.println(”42”);
```

- Vid anropet avgörs vilken av metoderna som ska användas.
- När överlagring används bör metoden ha samma beteende, `println()` ska till exempel skriva ut oavsett vad som skickas in till metoden.

# Exempel

```
package lecture5;
public class MoreDifference {
    public static int difference(int a, int b) {
        if(a > b)
            return a - b;
        else
            return b - a;
    }
    public static double difference(double a, double b) {
        if(a > b)
            return a - b;
        else
            return b - a;
    }
    public static int difference(String a, String b) {
        if(a.length() > b.length())
            return a.length() - b.length();
        else
            return b.length() - a.length();
    }
    public static void main(String[] args) {
        System.out.println("Skillnaden mellan 42 och 11: " + difference(42, 11));
        System.out.println("Skillnaden mellan 2,0 och 5,2: " + difference(2.0, 5.2));
        System.out.print("Skillnaden mellan \"Doctor Who\" och \"Star Wars\": ");
        System.out.println(difference("Doctor Who", "Star Wars"));
    }
}
```

# Utskrift

- Programmet ger följande utskrift:

```
Skillnaden mellan 42 och 11: 31
Skillnaden mellan 2,0 och 5,2: 3.2
Skillnaden mellan ”Doctor Who” och ”Star Wars”: 1
```

# Olika antal parametrar

- Antalet ingående parametrar kan också skilja mellan olika överlagrade metoder.
- Parametrarna kan vara av olika typ och ordning.
- Genom att använda olika antal parametrar så kan man göra olika nivåer av finkorninghet på metoderna.
- Det är också möjligt att ange en *dynamisk* lista av parametrar.
  - Boken går dessvärre inte in på det här.
  - Efter den sista datatypen kan man sätta tre punkter för att tala om att det är en lista av den typen.
  - Endast en dynamisk lista per metodsignatur.
- Din IDE är duktig på att säga till om något blir fel när man skapar många överlagrade metoder.

# Exempel

```
package lecture5;
public class UltimateHello {

    public static void sayHello() {
        System.out.println(”Well, hello there!”);
    }

    public static void sayHello(String message) {
        System.out.println(message);
    }

    public static void sayHello(int count) {
        for(int i = 0; i < count; i++)
            sayHello();
    }

    public static void sayHello(String message, int count) {
        for(int i = 0; i < count; i++)
            sayHello(message);
    }

    public static void main(String[] args) {
        sayHello();
        sayHello(”I'll see you on the dark side of the moon!”);
        sayHello(3);
        sayHello(”Hey you”, 2);
    }
```

# Utskrift

- Programmet ger följande utdata:

```
Well, hello there!
I'll see you on the dark side of the moon!
Well, hello there!
Well, hello there!
Well, hello there!
Hey you
Hey you
```

# Avslutning

- Anledningen till att använda metoder är att *dela upp* problemet i mindre delar.
- Det leder till enklare delar att förstå och implementera.
- Metoder är också en viktig del av objektorienteringen så därför är det viktigt att förstå.
  - Det kommer kursen att återvända till.
- Även om metoder inte är en *livsviktig* del i programmering, så är det något som i dag ses som absolut obligatorisk del.
  - Det är också en *livsviktig* del för att förstå objektorientering som kommer snart.