Högskolan Kristianstad
291 88 Kristianstad
044 250 30 00
www.hkr.se

Fakulteten för naturvetenskap
Andreas Nilsson

DA556, Programmering i Java

Lektion 6 - Klasser och Arv

Introduktion

Detta dokument syftar till att på ett, förhoppningsvis, enkelt sätt täcka delar av det som nämns under modulens läsanvisningar. Dokumentet riktar sig främst till de studenter som vill få en andra genomgång av materialet, på svenska, men alla studenter uppmanas att läsa igenom detta dokument och göra instuderingsuppgifterna sist i dokumentet.

I lektion 5 nämndes att när program blir större och mer komplexa måste programmeraren tänka över hur hen ska skriva sin kod och att hen måste dela in sin kod i mindre delar. Det samma gäller för klasser och objekt, fast på en något högre abstraktionsnivå.

Klasser används för att dela in ett program i logiska delar som ansvarar för en speciella funktion eller del av programmet. Klasser består av både metoder och av variabler. Ett exempel på en klass som redan använts är det ”dokument” som skapas när ett nytt projekt skapas i IntelliJ. Detta ”dokument” som innehåller bland annat main-metoden är en klass.

Efter lektionen är målet att studenten skall förstå hur begreppen klass och objekt förhåller sig till varandra samt hur dessa används rent praktiskt.

Varför behöver vi kunna detta?

En anledning till att lära sig om klasser och objekt är att de är grunden till en av de mest använda programmeringsteknikerna, objektorienterad programmering. I stort sett vart en programmerare än kommer så förväntas hen kunna programmera objektorienterat.

En annan fördel med att använda klasser och objekt är att koden blir mycket mer strukturerad, betydligt lättare att läsa och sätta sig in i. Det är inte alltid väl strukturerad kod är prestandamässigt snabbast om man jämför med andra tekniker för att programmera, men oftast överväger fördelarna och därav väljs oftast objektorienterad programmering.

Klasser möjliggör även återanvändning av kod på ett naturligt sätt. Detta underlättar programmerarens vardag då denne inte behöver uppfinna hjulet om och om igen utan kan återanvända redan framtagen och testad kod i nya projekt. Återanvändning av kod ger även fördelen att mer tid kan ägnas åt att lösa de svåra uppgifterna i det för tillfället aktuella projektet.

Strukturera kod

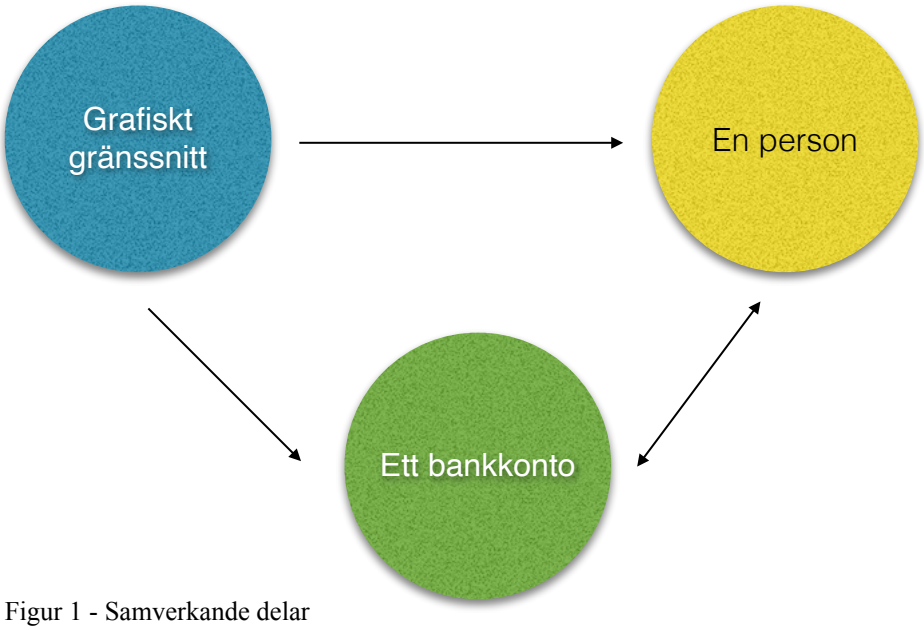
Att strukturera kod har inte alltid varit så lätt. I datorns barndom när maskinkod var det som användes för programmering fanns det exempelvis inte något lätt sätt att läsa och organisera kod. Just problemet med att läsa koden gjorde att koden blev svåröverskådlig.

När man senare gick över till programmeringsspråk som byggde på procedurer (tänk metoder) blev det lite lättare att organisera koden men för att kunna skriva och förstå kod på ett effektivt sätt krävdes att programmerarna var erfarna.

Nästa steg i utveckligen var att gå ifrån att fokusera på koden utan att istället fokusera på datan. Objektorienterad programmering (OOP) var född. Några av fördelarna man såg med OOP var att det blir lättare att återanvända kod genom att skapa bibliotek av klasser och att varje klass kan ses som en svart låda som ansvarar för att datan i den lådan är korrekt.

Att se på en klass som en svart låda som ansvarar för viss data är synnerligen tilltalande då programmeraren slipper att olika delar av koden i onödan påverkar varandra. Varje klass ansvarar för sin data och skulle klassen behöva ändras internt påverkar det i bästa fall inte andra delar av programmet.

I Figur 1 nedan kan tre fiktiva klasser ses. Varje klass ansvarar för respektive del i programmet och ifall något skulle behöva ändras i klassen som representerar en person påverkar det inte de övriga två klasserna ifall ”Personklassen” definierat ett tydligt och genomtänkt gränssnitt från början.



Figur 1 - Samverkande delar

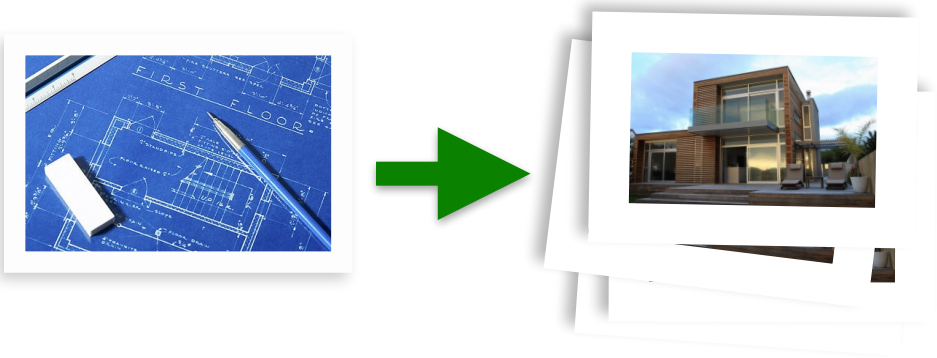
Klasser och objekt

Vad är då skillnaden mellan en klass och ett objekt? En klass kan ses som en mall för ett objekt. En klass beskriver vilka variabler och vilka metoder som kommer att finnas i ett objekt.

Även om dessa synonymer/beskrivningar inte används praktiskt kan de vara till hjälp för att förstå skillnaden mellan en klass och ett objekt.

> Klass = En ritning, en beskrivning, ett recept eller en mall
>
> Objekt = Ett faktiskt objekt skapat utifrån exempelvis en ritning. Ett hus, ett träd, en bakad kaka eller ett format tygstycke (utklippt efter en mall)

Figur 2 ger en grafisk representation av skillnaden mellan klasser och objekt.



Figur 2 - Klass (vänster) vs. Objekt (Höger)

En programmerare skriver klasser som sen används för att skapa (instansiera) ett eller flera objekt. Objekten används sedan i programmet för att utföra en eller annan aktivitet. Ett exempel skulle kunna vara att en programmerare skriver en klass som beskriver vilka egenskaper en bil har. Bilen kan ha en färg, en topphastighet och ett tillverkningsår. Klassen anger också funktionalitet som alla bilar har. Exempelvis kan en bil öka farten och bromsa. När klassen är klar kan programmeraren skapa instanser av sin ”bilklass”. Den första bilen kanske är en röd bil med topphastigheten 100km/h och tillverkningsår 2013 och en annan bil kanske är blå med en topphastighet av 200km/h och tillverkningsår 2015.

Åtkomstkontroll med public och private

Anledningen till att åtkomstkontroll används är att klassen skall kunna vara ansvarig för sin data. Ingen skall kunna ändra ett objekts data utan att gå via de vägar som definierats av objektets klass. Om programmeraren skapat en klass som hanterar bankkonton vill hen exempelvis inte att vem som helst skall kunna ändra information om hur mycket pengar som finns på bankkontot. Eventuella ändringar måste först verifieras osv.

För exempel på hur åtkomstkontroll används se kapitlet Exempel nedan.

Skulle en klass innehålla data som behöver göras tillgängliga för andra klasser görs detta genom att datan i sig (variablerna) sätts till att vara privata men att vad som kallas ”getter” och ”setter” metoder skapas. Dessa metoders uppgift är att antingen hämta eller ändra värdet på en speciell variabel. Fördelen med denna konstruktion är att metoderna kan utföra kontroller innan värdena ändras, exempelvis att vi inte kan sätta in en negativ summa på ett bankkonto eller att längden är ett nyfött barn inte kan vara negativ.

Metoderna i en klass är oftast en blandning av privata och publika metoder. Alla metoder som inte behöver användas av andra klasser skall vara privata. Detta för att i så stor utsträckning som möjligt dölja en klass inre för övriga klasser. Resultatet blir att en klass inre oftast kan modifieras och omarbetas utan att det påverkar andra klasser.

Det reserverade ordet static igen

Tidigare i kursen har statiskt kontext nämnts men det är värt att ta upp igen nu när kursen även introducerat klasser och objekt.

En statisk metod eller variabel kan sägas vara obunden, den tillhör inte ett specifikt objekt utan är data eller funktionalitet som delas av alla objekt av samma klass.

Några exempel på variabler som skulle kunna vara statiska är matematiska konstanter i stil med Pi, eller eventuellt en räknare som håller redan på hur många objekt som skapats av en klass (ökas med ett varje gång ett objekt skapas).

Metoder som skulle kunna vara statiska är exempelvis metoder som inte har någon anknytning till ett visst objekts data. Exempel på sådana metoder skulle kunna vara de metoder som finns i den inbyggda klassen Math (ex. Math.sqrt).

Arv

Inom objektorienterad programmering finns det ett koncept som kallas arv och som är väldigt starkt kopplat till klasser. För att sammanfatta vad arv är kan likheter dras till arv hos människor, klasser som ärver från en annan klass ärver dess variabler och funktionalitet.

Viktigt att notera är att även om en klass ärver en annan klass variabler och metoder genom arv är det inte säkert den ärvande klassen får direkt åtkomst till dem. Är ”förälderns” variabler eller metoder markerade som privata kommer ”barnet” trots att det ärver dem inte att direkt komma åt dem utan får exempelvis använda sig av getters och setters för att komma åt variablernas värde.

En klass som är ”förälder” i ett arv kallas superklass och en klass som är ”barn” kallas subklass.

Vitsen med att använda sig av arv är att ifall man har flera klasser som är snarlika kan man lägga all gemensam kod i en superklass och på så vis undvika att duplicera kod. Exempelvis har både bilar och lastbilar vissa gemensamma attribut som kan flyttas upp till en förälder som exempelvis döps till fordon. Nu kan klassen fordon specialisera sig på allt som är gemensamt för fordon i allmänhet och de respektive subklasserna på att vara specialister på just sin fordonstyp.

När man navigerar uppåt i ett träd av klasser som ärver av varandra säger man att man får mer generella klasser (Generalization) och när man går nedåt i trädet blir klasserna mer och mer specialiserade (Specialization).

Hur vet programmeraren om hen skall använda arv? Om man kan använda sig av följande mening och byta ut <sub class> och <super class> mot sina egna klasser är det lämpligt att fundera över ifall arv skall användas.

<sub class> **is a** <super class>

Exempelvis:
A Car **is a** Vehicle

Car	en subklass
Vehicle	en superklass

I Java är det bara tillåtet för en klass att ärva från en superklass. I vissa andra språk är det tillåtet för en klass att ha flera superklasser som man ärver i från, det kallas multipelt arv.

Mer än klasser och objekt

Java har alltid varit ett starkt objektorienterat språk, men på senare tid har det dykt upp ny funktionalitet i språket som till viss mån tar avsteg från det traditionella objektorienterade sättet att programmera, med klasser som skyddar sina

instansvariabler och där åtkomst sker genom getter och setter metoder. Istället kan man använda sig av vad som kallas records.

Records skiljer sig lite från klasser på några punkter, det första man ser när man tittar på dem är att de är mycket mer kompakta i sin natur, men något man troligen inte direkt ser genom att bara titta på dem är att det inte går att ge dem några nya värden när de väl skapats, de är *immutable*.

Då vi i denna kursen fokuserar på grunderna i Java kommer vi inte att titta närmare på, eller använda oss av records. Dock är det bra att känna till begreppet records och veta vad det är ifall man springer på det i något sammanhang.

Givetvis får den nyfikne studenten gärna på egen hand titta närmare på records, och skulle detta låta intressant kan det officiella förslaget på att lägga till records i språket vara en bra början[1], men vill man ha en lite mer beskrivande variant kan följande information om ändringen i Java kanske vara bättre[2] (egentligen säger det första exemplet det mesta om hur records används i praktiken).

Exempel

Låt oss studera några korta exempel för att se hur klasser och objekt kan användas rent praktiskt.

Exempel 1 - Klass med privat data och åtkomst via getter/setter

```
public class Example {
    public static void main(String[] args) {
        // Skapa ett nytt objekt
        DataStorage ds = new DataStorage();
        // Vi sätter att vi äger 5 kycklingar
        ds.setNbrOfChickens(5);

        System.out.println("We have this many chickens: " +
            ds.getNbrOfChickens());
    }
}

public class DataStorage {
    // Privat så att åtkomst endast kan ske via getter/setter. Detta
    // ger oss full kontroll över vilka värden som är tillåtna.
    private int nbrOfChickens = 0;

    // getters namnges "get" + variabelnamn
    public int getNbrOfChickens() {
        return nbrOfChickens;
    }

    // setters namnges "set" + variabelnamn
    public void setNbrOfChickens(int chickens) {
        // Vi kan inte äga mindre än noll kycklingar så tillåt bara
        // positiva värden.
        if(chickens >= 0) {
            nbrOfChickens = chickens;
        }
    }
}
```

Exempel 2 - En bilklass

```
public class Example {
    public static void main(String[] args) {
        // Skapa två bilar(objekt) från samma klass. De representerar
        // två helt olika bilar med olika egenskaper
        Car myCar = new Car();
        Car friendsCar = new Car();

        myCar.setTopSpeed(100);
        myCar.setProdYear(1990);
        myCar.setColor("Black");
        friendsCar.setTopSpeed(200);
        friendsCar.setProdYear(2015);
        friendsCar.setColor("Red");
```

```
        // Exempel på användande av metoder från de skapade objekten
        myCar.accelerate(50);
        myCar.brake(25);
        System.out.println(myCar.getColor());
        System.out.println(friendsCar.getColor());
    }
}

public class Car {
    private int topSpeed = 0;
    private int prodYear = 0;
    private String color = "";

    public int getTopSpeed() {
        return topSpeed;
    }

    // om parameternamn och instansvariabelns namn är samma kan man
    // referera till variabelnamnet genom att skriva this.XYZ
    public void setTopSpeed(int topSpeed) {
        this.topSpeed = topSpeed;
   }

    public int getProdYear() {
        return prodYear;
    }

    public void setProdYear(int prodYear) {
        this.prodYear = prodYear;
    }

    public int getColor() {
        return color;
    }

    public void setColor(int color) {
        this.color = color;
    }

    public void accelerate(int kmh) {
        // Här ska finnas kod för att accelerera bilen "kmh" km/h
    }

    public void brake(int newSpeed) {
        // Här skall finnas kod för add bromsa in till den nya farten
        // "newSpeed".
    }
}
```

Exempel 3 - Arv

```
public class Example {
    public static void main(String[] args) {
        Car c = new Car();
        // Då en bil är ett fordon kan vi skapa en variabel som är av
        // typen Vehicle men lagrar en Car. Det är en bil som lagras
        // men då Java bara ”vet” att det är en Vehicle kan vi inte
        // använda metoder deklarerade i subklassen utan bara de som
        // finns i superklassen.
        Vehicle c2 = new Car();
        c.setTopSpeed(100);
        c2.setColor(”Black”);

        // Då vi vet att c2 är en bil kan vi ”tvinga” Java att se det
        // som en bil genom att ”type casta”. Det innebär att vi talar
        // om att en variabel skall ses som om den vore av en annan
        // typ. ”Type cast” görs så här: ((ny typ)variabelnamn).
        // Genom att ”type casta” kan vi sätta c2:s topphastighet.
        ((Car)c2).setTopSpeed(250);
    }
}

public class Vehicle {
    // Alla egenskaper och metoder som är gemensamma för subklasserna
    // läggs i superklassen. I detta exempel nöjer vi oss med en
    // variabel för att illustrera hur det fungerar.
    private String color;

    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
}

// Genom att lägga till ”extends Vehicle” markerar vi att klassen
// skall ärva från klassen Vehicle.
public class Car extends Vehicle{
    // I subklasserna kan vi utöka en klass funktionalitet med sånt
    // som inte finns i superklassen. I detta exempel adderas förmågan
    // att ha en maxhastighet vilket i verkligheten borde legat i
    // Fordonsklassen, för alla fordon har en maxhastighet. Men för
    // att exemplifiera hur klasser fungerar läggs förmågan att ha en
    // maxhastighet i subklassen…
    private int topSpeed = 0;

    public String getTopSpeed() {
        return topSpeed;
    }

    public void setTopSpeed(String speed) {
        // Ifall parametern inte har samma namn som variabeln behöver
        // inte this.XYZ användas.
        topSpeed = speed;
   }
}
```

Referenser

[1] JEP 395: Records. (2022) Oracle Corporation. [Online] Tillgänglig från: https://openjdk.java.net/jeps/395. [Tillgänglig: 2022-02-18]

|2] Record Classes. (2021) Oracle Corporation. [Online] Tillgänglig från: https://docs.oracle.com/en/java/javase/15/language/records.html. [Tillgänglig: 2022-02-18]