

3.1 Vad är objektorienterad programmering?

En given definition på programmering är problemlösning med hjälp av datorn. Om man då beskriver problemets lösning i form av en *algorithm* kan man säga *Program* = *algorithm* + *data*. Denna definition ställdes upp av Niklaus Wirth på 60-talet och återspeglar den procedurala synen på programmering. Fokuset ligger på *algoritmen* dvs att inte bara hitta utan även *beskriva* tillvägagångssättet (proceduren) för att lösa ett problem. Sedan återstår bara att koda denna beskrivning. En annan definition som kom upp på 80-talet och återspeglar den objektorienterade synen på programmering är:

Program = Modell av verkligheten

Om man i formeln *Program* = *algorithm* + *data* lägger betoningen på *data* istället för på *algoritmen* och inte längre betraktar *data* som ett slags bihang till *algoritmen* utan som *objekt* kommer man till *objektorienterad programmering*. Denna nya programmeringsfilosofi genomsyr alla våra program, eftersom C# med alla sina fördefinierade biblioteksprogram är i högsta grad objektorienterade.

Paradigmskifte

Det som i programmeringshistorien gjorde att man behövde objektorienterad programmering var den växande komplexiteten hos program under 70-talet. Programmens storlek var avgörande för den växande komplexiteten. Man insåg att det inte längre räckte till att skriva och testa program som fungerade just då. Det var nödvändigt att med rimliga kostnader kunna även *underhålla* stora program, *förnya* och *vidareutveckla* dem så att de fungerade även i flera år och att de framför allt kunde anpassas till nyuppkomna situationer utan oöverkomliga svårigheter. Det i sin tur krävde att man redan i designstadiet behövde ett annorlunda upplägg. Fokuset förskjöts från problemlösning till modellering av verkligheten. Objektorienterad design kom in i bilden. Allt detta var endast med procedural programmering inte längre möjligt. Ett s.k. *paradigmskifte* hade blivit nödvändigt, dvs en ändring av helhetssynen på programmering.

Objektorienterad programmering syftar åt att efterlikna verkligheten. Man vill avbilda den reala världen – åtminstone den del som tillåter datorisering – och konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten genom att testa modellen. För att undvika filosofiska diskussioner kan vi anta att den reala världen består kort sagt av *objekt*. Världen kring oss är full med sådana objekt: Människor, byggnader, bilar, tåg, flygplan, träd, möbler, böcker, butiker, skolor, bibliotek, kontor, anställda, kunder, varor, fakturor, order, bokningar, kurser osv. Objekten kan vara verkliga eller virtuella. Ett datorprogram försöker att beskriva dessa objekt. Låt oss precisera detta:

Objekt, klass och metod

Ett *objekt* har vissa egenskaper. Generellt kan man säga att ett objekt är summan av alla sina egenskaper. Ett annat ord för egenskap är *attribut*. Ett objekt består av alla sina attribut. Attributen tillhör objektet. T.ex. har objektet bil som attribut fabrikat, modell, färg, årsmodell, antal körda mil, antal hästkrafter, maximala hastigheten, antal och storlek på cylindrar i motorn osv. Alla dessa data ger svar på frågan ”Vad är det för bil?”. Men bilden vore ofullständig om vi nöjde oss med dessa intressanta, men statiska data. Vi vill också veta vad man kan *göra* med bilen. Ett objekt kan i regel även utföra vissa aktioner eller operationer. I den objektorienterade programmeringens terminologi kallas de för *metoder*. Typiska metoder för en bil är t.ex. att köra fram, att backa, att accelerera, att bromsa, att parkera, att byta olja osv. Den fullständiga definitionen på en bil som objekt vore alltså att ange *både* dess attribut *och* metoder. Bilfabrikanten måste förse bilen med alla dessa färdigheter för att kunna sälja den. Därför går man i bilfabriken efter en plan när man tillverkar bilen. I den objektorienterade programmeringens terminologi kallas denna plan för bilens *klass*. När vi skriver ett program måste vi först formulera *klassen Bil* för att sedan kunna skapa objekt av den. Klassen skrivs bara en gång, medan objekt kan skapas enligt klassens beskrivning i obegränsat antal. I klassen måste vi ta upp alla attribut och metoder som är relevanta eller av någon anledning önskvärda för en bil. Den praktiska användningen avgör från fall till fall vad som är relevant eller önskvärt.

Vad är skillnaden mellan *objekt* och *klass*? Om vi byter ut bilar mot pepparkakor kan man säga att pepparkaksformen är klassen och själva pepparkakorna är objekten. Klassen är alltså en slags mall, en föreskrift för produktion av objekt: En enda pepparkaksform kan producera tusentals pepparkaksgubbar. Gubbarna kan skiljas från varandra i vissa detaljer, t.ex. materialet, smaken osv. Man kan t.o.m. måla dem i olika färger eller modifiera på annat sätt efteråt. De förblir pepparkaksgubbar av den ursprungliga formen. I formen ingår det som är gemensamt hos alla pepparkaksgubbar. Man har, när man byggde formen, bortsett från oväsentliga skillnader och tagit hänsyn endast till det väsentliga, det gemensamma hos alla pepparkakor.

Att *bortse* från skillnader och att bibehålla det gemensamma hos olika verkliga objekt, är en *abstraktion* (*abstrahera* betyder på latin: att ta bort, att dra av). Man tar bort allt som skiljer saker och ting av samma kategori eller typ och kommer på det viset till själva kategorin. Abstraktion leder till *begrepps*bildning, till *klassificering* eller *kategorisering* av den reala världen. Ett växande barn går igenom samma abstraktionsprocess, ser först sina föräldrar (objekt), abstraherar sedan via erfarenhet så småningom till begreppet *människa* (klassen) och inser att sina föräldrar är två konkreta exemplar av den abstrakta klassen *människa*. Så gör barnet med alla saker och ting omkring sig och lär sig vuxenvärldens begreppsapparat. Det abstrakta begreppet *penna* (klassen) t.ex. bildas efter att man sett hundratals verkliga pennor (objekt). Objektorienterad programmering återspeglar denna naturliga tankeprocess från det konkreta till det abstrakta, från objekt till klass.

Metoder

En *metod* är en funktionalitet som definieras i en klass. Den talar om vad ett objekt av denna klass kan *göra*. Det finns två steg i hantering av metoder: Först definierar man dem dvs skapar man deras kod i en klass. Sedan *anropar* dvs aktiverar man dem i ett objekt av denna klass. Ofta är det första steget redan genomfört av andra, så vi behöver bara anropa en redan *fördefinierad* metod. I klassen *Bil* t.ex. är metoderna att köra fram, att backa, att accelerera, att bromsa osv. definierade i huvuden på bilkonstruktörerna och i deras konstruktionsritningar och dokumentationer. Sedan har man tillverkat massor med objekt av klassen *Bil* i fabriken och byggt in dessa metoder i alla bilar. Vi behöver bara anropa dem i den bil vi kör. Den bil vi kör är ett specifikt objekt av klassen *Bil*. Låt oss kalla det för **minVolvo**. Objektet **minVolvo** har ett antal attribut som t.ex. fabrikat, modell, färg, årsmodell osv., men också ett antal metoder, bl.a. metoden **Kör()**. Parenteserna i metodens namn brukar man skriva för att karakterisera **Kör()** som en *metod* och skilja den från klassens attribut. I C# skriver man ett anrop av metoden **Kör()** så här:

```
minVolvo.Kör();
```

Observera att *före* punkten står ett objekt, inte klassen. Det är ju den specifika bil som jag använder just nu som ska köras. Först *efter* punkten står själva anropet av metoden **kör()**. Det här sättet att skriva kallas *punktnotation*. Metoder måste alltid anropas med punktnotation, vilket har sin grund i att de endast är deklarerade i klasser, så att de endast existerar i objekt av en klass. Till skillnad från fristående *funktioner* kan metoder varken definieras utanför klasser eller anropas utanför objekt. I C# finns endast metoder, inga funktioner. Om vi bortser från bil exemplet kan det i andra sammanhang även förekomma en klass (istället för objekt) före punkten i anropet av en metod. I så fall är metoden definierad i klassen på ett speciellt sätt nämligen som en *statisk* metod, vilket tas upp senare när vi behandlar metoder i detalj.

En annan variant av metoden **Kör()** kan anropas på följande sätt:

```
minVolvo.Kör(40);
```

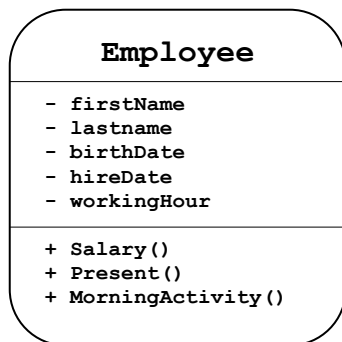
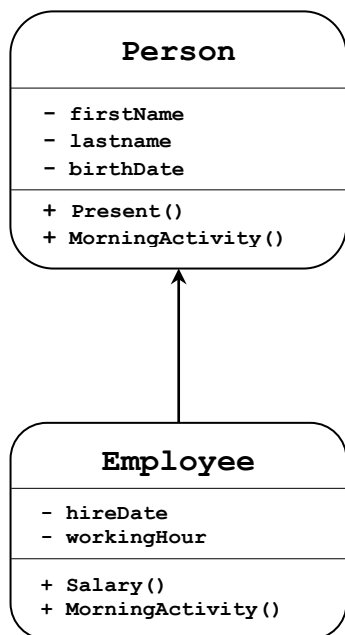
Det kan t.ex. betyda: Kör bilen med hastigheten 40 km/h. Värdet 40 kallas då en *parameter* som skickas till metoden när den anropas. I så fall måste även metoden **Kör()** vara definierad så att den har beredskapen att ta emot denna parameter. Så det kan inte vara samma metod som anropades *utan* parameter. Det måste vara en annan variant av den, exakt talat en annan metod med samma namn. Konceptet kallas *överlagring av metoder* och innebär två eller flera metoder med samma namn, men olika parametrar.

Klassdiagram

Låt oss ta som exempel en algoritm som beskriver hur man går upp, duschar, tar på sig kläderna och åker till jobbet (algoritmen *Morgonsyssla* i *Progr1+*, 1.4). Detta är ett typiskt fall av problemlösning: Det löser problemet *hur* man tar sig till jobbet. Tillvägagångssättet och framför allt hur vi beskriver det, är föremål för algoritmer. Men

vem eller vilka gör det, dvs vilka *objekt* som är involverade i algoritmen och hur man beskriver dessa objekt, är en annan aspekt på saken. Objektorienterad programmering prioriterar objektaspekten framför algoritmaspekten. Den primära frågan är inte längre vad man *gör* utan vem man *är* dvs *hur kan personen beskrivas*? Hur man gör för att ta sig till jobbet kommer att ingå som en del i denna beskrivning. Algoritmen Morgonsyssla blir en metod i *objektet* Person. Det är objektet som utför metodens instruktioner för att ta sig till jobbet.

Personen kan t.ex. vara en anställd vilket förresten skulle förklara varför han tar sig till jobbet. I så fall är personen ett objekt av kategorin eller klassen *Employee*. Därför definieras en klass som beskriver alla anställda. Personen i fråga görs till ett objekt, ett exemplar av denna klass. På så sätt kan koden återanvändas även för andra anställda. Återanvändning av kod gör utvecklingsarbetet av programvara effektivare och är en av den objektorienterade synens fördelar. I klassen *Employee* ingår all typ av information som är relevant för en anställd, det vi kallar för attribut, t.ex. för- och efternamn, födelse- och anställningsdatum, arbetstid osv. Dessutom tar vi upp allt som en anställd kan göra, det vi kallar för metoder, t.ex. att få lön, att presentera sig eller också att ta sig till jobbet. På så sätt blir algoritmen Morgonsyssla i den objektorienterade programmeringens terminologi en metod i klassen *Employee*. Ett verktyg speciellt för objektorienterade modelleringar är UML (*Unified Modeling Language*). Enligt det här modelleringsspråket skulle klassen *Employee* beskrivas med diagrammet till höger som kallas för *klassdiagram*. Där står tecknet – för attribut och + för metoder. Andra beteckningar för attribut är *datamedlem* eller *egenskap*. Dessa termer är synonymer. En klass består av datamedlemmar och metoder. Klassen **Employee** t.ex. har fem datamedlemmar och tre metoder.



Klassens konstruktör

Eftersom klassens datamedlemmar i regel är in-kapslade (privata) och inte åtkomliga utifrån klassen – detta gör man bl.a. ur datasäkerhets-synpunkt – måste programmeraren använda sig av ett verktyg för att på ett kodat sätt ändå kunna komma åt dem, läsa och ändra dem osv. Detta verktyg kallas klassens *konstruktör* och är en speciell metod vars namn är identiskt med klas-sens namn. Den initierar automatiskt klassens

privata datamedlemmar när ett objekt skapas. För enkelhetens skull har vi inte tagit upp den i klassdiagrammet ovan bland klassens metoder. Konstruktorn har ju endast programmeringsteknisk karaktär och behandlas i detalj senare.

Arv

I den reala världen som vi vill efterlikna, finns inga isolerade objekt. Alla objekt är mer eller mindre relaterade till andra objekt. En klok modellering måste dra nytta av de befintliga relationer mellan objekt för att effektivisera och optimera utvecklingsarbetet. En sådan relation är arvrelationen.

Man kan alltid etablera en arvrelation mellan två begrepp om de står i en ”är”-relation till varandra. I exemplet ovan kan vi konstatera ett en anställd *är* en person. Därför kan klassen **Employee** ära klassen **Person**, närmare bestämt ärver klassen **Employee** klassen **Person**s alla datamedlemmar och metoder. Klassen **Person** kallas *bas-* eller *superklass*. Klassen **Employee** kallas *härledd* eller *subklass*. Subklassen ärver superklassens alla datamedlemmar och metoder, vilket i praktiken innebär att klassen **Employee** tar över all kod som redan finns i klassen **Person** och lägger till ny kod som närmare specificerar en anställd. På så sätt slipper man skriva om kod som redan finns. T.ex. har en person ett för- och efternamn samt ett födelsedatum. Vid modellering av en anställd ärvs dessa attribut, och man lägger till de nya attributen `hireDate` och `workingHour` som är speciella för en anställd. Klassdiagrammet ovan (till vänster) visar modellen där arvrelationen ritats med en pil riktad mot superklassen. Följer man pilens riktning underifrån kan man avläsa att det är klassen **Employee** som ärver klassen **Person**.

Observera att klassen **Employee** inte har två utan fem attribut därför att den via arvrelationen även har **Person**-klassens tre attribut. Samma gäller för metoderna: **Employee**-klassen ärver metoden `Present()` från klassen **Person**. Modellen ovan går utifrån att personer presenterar sig på samma sätt som anställda. Sedan har anställda en löneberäkningsmetod som icke-anställda personer saknar. Men varför står metoden `MorningActivity()` i båda klasser? Närmare bestämt: Varför förekommer den i **Employee**-klassen fast den ärver den från superklassen? Svaret ges av ett annat koncept inom objektorienterad programmering:

Polymorfism

Modellen ovan går utifrån att icke-anställda personer har en annan form av morgonsyssla än anställda. De kanske inte tar sig till jobbet, i alla fall inte alla, utan har en annan morgonsyssla. Så vi har här att göra med två olika morgonsysslor tillhörande två olika klasser, men med samma namn. För objekt av typ **Person** kommer den ena och för objekt av typ **Employee** kommer den andra att gälla. Men varför har de samma namn? Vore det inte bättre, för att undvika namnkonflikt, att ge dem olika namn, när de ändå är olika metoder? Faktiskt inte!

Anledningen till att de har samma namn är följande: För det första blir det ingen namnkonflikt därför att de tillhör olika typer av objekt. De är inte fristående utan

inkapslade i var sitt objekt som skiljer åt dem. För det andra ska vi inte i onödan göra utvecklingsarbetet komplicerat genom att hitta på nya namn på metoder som skiljer sig från varandra endast i detaljer. Ingen människa skulle kunna komma ihåg så många namn. För det tredje vill vi efterlikna verkligheten där det bara kryllar av beteckningar som är identiska, men har olika innebörd i olika sammanhang. Inte heller det vanliga språket har olika namn på dem. Ta följande exempel: Att bromsa en lastbil görs på ett annat sätt än att bromsa en båt. Det finns ingen anledning att hitta på ett annat namn för funktionaliteten "att bromsa" hos olika typer av fordon. Tvärtom, det vore förvirrande att använda olika namn. Man vill ju helst slippa att tänka på de tekniska skillnaderna mellan olika typer av fordon när man pratar om bromsning. En och samma funktionalitet är realiserad på olika sätt. Med andra ord, man gör "samma sak", fast på annorlunda sätt. Objektorienterad programmering tar över detta koncept genom att välja ett och samma namn för olika metoder. När metoderna dessutom finns i klasser som ärver varandra kallas konceptet för *polymorfism*.

Polymorfism modifierar helt eller delvis funktionaliteten hos metoder med samma namn som förekommer i en arvhierarki.

”Poly” betyder *många* och ”morf” är *form* eller *gestalt* på latin och antik grekiska. Polymorfism handlar om en sak som har många olika gestalter, t.ex. ett ord som har många olika betydelser. En metod beskriver alltid någon funktionalitet. Polymorfism förändrar denna funktionalitet genom att definiera en metod i superklassen och definiera om innehållet, men behålla namnet i subklassen.

Objektorienterad programmering har kommit till för att förverkliga programmeringens gamla önskedrömmar om *modularisering*, *återanvändning av kod* och *strukturering av program*. Allt för att kunna underhålla stora program, förnya och vidareutveckla dem, så att de fungerar över längre tid och snabbt kan anpassas till nyuppkomna situationer.

Objektorienterad programmering bygger på tre hörnstenar:

- Inkapsling
- Arv
- Polymorfism

De sista två har vi försökt att introducera här utan att behöva skriva kod. För att förstå *inkapsling* behöver vi mer detaljerade kunskaper om objektorientering samt skriva lite kod, vilket vi gör i de kommande avsnitten. Sedan ska vi återkomma till *arv* och *polymorfism*, för att förse även dem med kod.