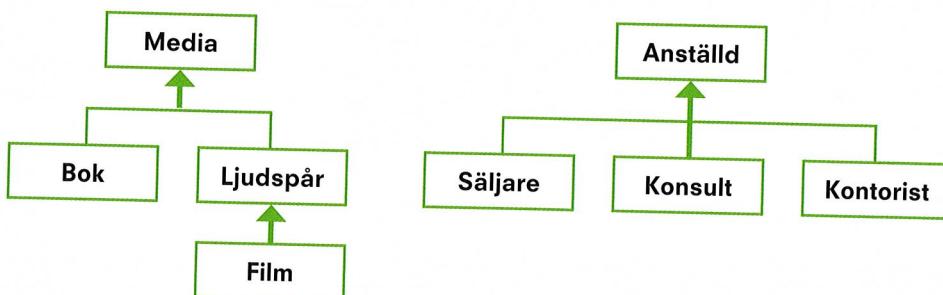


- ▶ Anställd.BeräknaLön() returnerar alltid 0.
- ▶ Säljare.BeräknaLön() returnerar försäljning \* provision / 100.
- ▶ Konsult.BeräknaLön() returnerar timLön \* arbetadeTimmar.
- ▶ Kontorist.BeräknaLön() returnerar bara månadsLön.

## 2.3 Abstrakta klasser

Betrakta klasshierarkierna från föregående avsnitt.



I inget av programmen användes basklassen för att skapa några objekt (förutom Ljudspär som är basklass till Film). Basklasserna Media och Anställd var bara till för att etablera en gemensam grundtyp. Om man som här inte avser att använda några instanser (objekt) av en klass, är det lämpligt att göra den abstrakt. En abstrakt klass går helt enkelt inte att göra ett objekt av. Om klassen Anställd ovan vore abstrakt, skulle det inte gå att skriva new Anställd(...) i källkoden.

### Exempel 2.3 Abstrakt basklass och metod

Klassen Anställd, till exempel, hade en metod BeräknaLön(), men den anropas aldrig. Vad skulle den beräkna lön efter? I en abstrakt klass kan man definiera en metod utan kropp, för den kommer aldrig att anropas. Man måste däremot alltid implementera den med override-metoder i alla basklasser. Nyckelordet abstract i en metoddefinition gör metoden till en sorts virtuell metod.

```

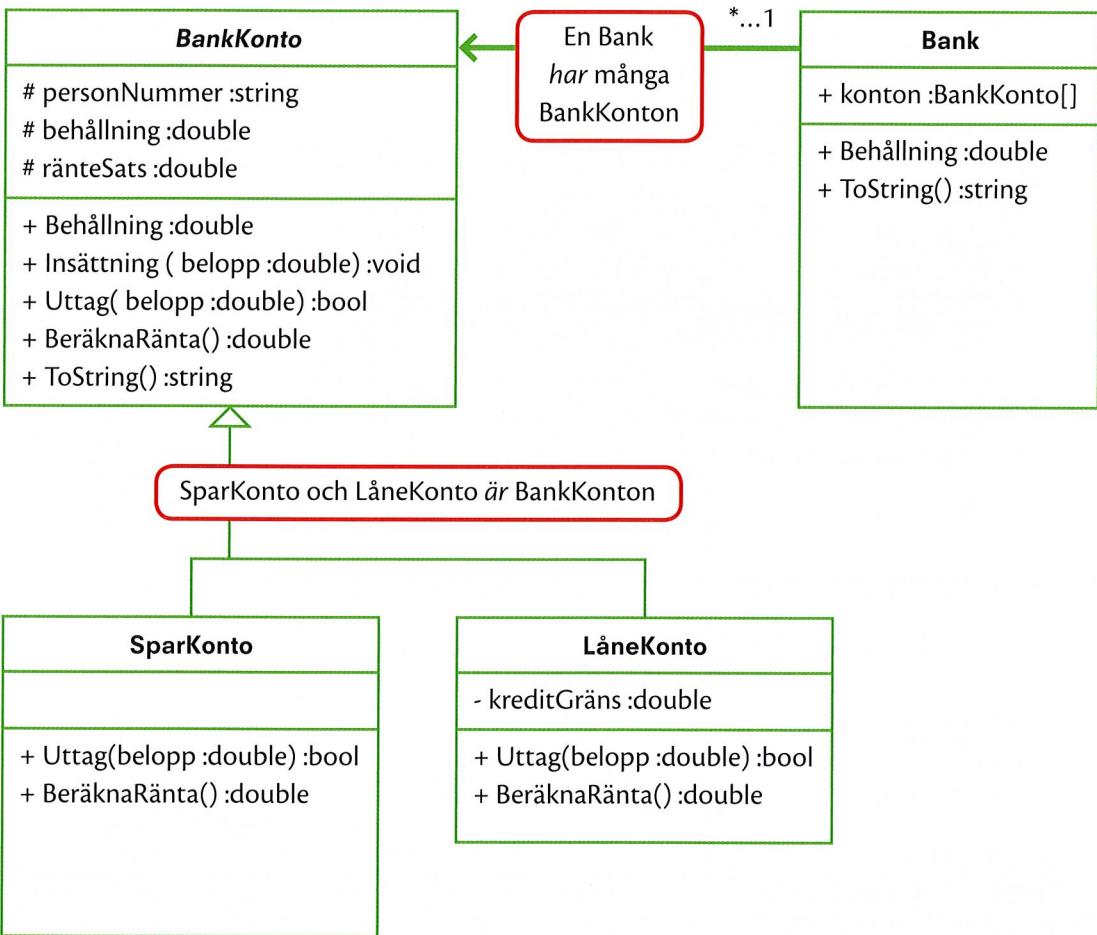
abstract class Anställd
{
    string namn = "";
    public Anställd( string n ) { namn = n; }
    public abstract double BeräknaLön();
}

class Säljare : Anställd
{
    double försäljning = 0;
    double provision = 0;
    public Säljare( string n, double f, double p ) : base(n)
    {
        försäljning=f; provision=p;
    }
    public override double BeräknaLön(){ return försäljning*provision/100; }
}
  
```

## Klassrelationer

Vi tänker oss nu ett program för en bank som erbjuder sina kunder två sorters konton. Den ena sorten är ett sparkonto, där man kan sätta in och ta ut pengar, men man kan aldrig ta ut mer än vad som finns. Den andra sorten är ett lånekonto. Man kan sätta in och ta ut pengar från det också, men man kan ta ut mer pengar än det finns, upp till en kreditgräns. Båda typerna av konton har en räntesats. Varje år drar banken ränta från lånekontona, och betalar ut ränta till sparkontona.

En klasshierarki skulle kunna se ut som nedan. Konstruktorerna är inte med av utrymmesskäl. Stakettecknet # anger att en medlem ska vara protected, alltså tillgänglig inom klasshierarkin, men gömd utanför den.



Det finns fyra klasser. Klassen Bank är en fristående klass, men den innehåller en samling av bankkonton. Precis som triangeln i diagrammet anger att ett SparKonto är ett BankKonto, anger pilen att Bank har BankKonto-objekt. \*..1 intill pilen anger att en bank har många bankkonton, och kallas för *multiplicitet*. Man skulle kunna ange multipliciteten 2..1 om man ville att en bank ska ha två bankkonton. Man kan läsa multipliciteten på båda hållen. I det här fallet blir det "en bank har många bankkonton", eller på andra håll "ett bankkonto tillhör en bank". Asterisken \* står för det mer obestämda antalet "många".

"Är" och "har" är två olika sorters relationer mellan klasser. "Är" indikerar arv. "Har" betyder bara att det i en klass finns en referens av en annan klass.

Relationen mellan SparKonto och BankKonto är av typen "är":

```
SparKonto : BankKonto
{
}
```

Relationen mellan Bank och BankKonto är av typen "har":

```
Bank
{
    BankKonto konto;
}
```

Klassen BankKonto är abstrakt, vilket anges med namnet i kursiv stil i diagrammet. Den innehåller två abstrakta metoder, Uttag och BeräknaRänta, som också är i kursiv stil. Underklasserna implementerar dessa metoder på sina egna sätt.

SparKonto.Uttag(...)	Minskar behållning med det angivna beloppet om beloppet är mindre än behållningen (som alltså inte får bli negativ). Returnerar true om uttaget godkändes, annars false.
LåneKonto.Uttag(...)	Minskar behållningen med det angivna beloppet, som kan bli negativt, så länge skulden inte blir större än kreditgränsen. Returnerar true om uttaget godkändes, annars false.
SparKonto.BeräknaRänta(...)	Ökar behållningen med den beräknade räntan.
LåneKonto.BeräknaRänta(...)	Minskar behållningen med den beräknade räntan.

### Övning 2.3 Sparbanken Banken

Implementera bankprogrammet ovan. Programmets fönster skulle kunna se ut som nedan. Om man registrerar ett konto med kredit blir det ett lånekonto, annars ett sparkonto.

## 2.4 Slutna klasser

Slutna klasser går inte att ärva. De definieras med nyckelordet sealed.

```
sealed class A
{
}

class B : A
{
}
```

Inte tillåtet. B kan inte ärva från den slutna klassen A.

String-klassen är ett exempel på en sluten klass. Man kan skriva egna klasser som ärver från andra .NET-klasser, men inte just från String.

## 2.5 Interface

Ett interface är ett kodblock som *bara* kan innehålla metoddefinitioner och egenskaper utan kroppar (och vissa typer, till exempel ett nästlat interface). På det sättet påminner det om en abstrakt klass, men en abstrakt klass *får* innehålla medlemsvariabler, fullständiga metoder och allt annat som en vanlig klass kan innehålla. Namnen börjar av konventionen med stora bokstaven I.

```
interface ISumma
{
    double Addera();
}
```

Som en klass, men utan variabler, konstruktör och metodkroppar.

Ett interface är en typ, och klasser kan ärva från den. Men de ärver inget fysiskt, utan underklasserna måste implementera interface-metoderna. Om man till exempel vill göra en klass som ärver från ISumma så måste klassen implementera Addera-metoden.

```
class Summa : ISumma
{
    double a, b;
    double Addera ( ) { return a + b; }
}
```

Klassen förbinder sig att implementera interfacets metoder

Kravet på att underklasser till ett interface måste implementera dess metoder gör att interface ofta beskrivs som ett kontrakt.



Det som gör interface användbart är att en klass kan implementera mer än ett interface.

## Exempel 2.4 En klass som implementerar flera interface

```

interface ISumma
{
    double Addera( );
}

interface IDifferens
{
    double Subtrahera( );
}

interface IProdukt
{
    double Multiplicera( );
}

interface IKvot
{
    double Dividera( );
}

class Beräkning : ISumma, IDifferens, IProdukt, IKvot
{
    double a, b;

    public Beräkning ( double a, double b ) { this.a = a; this.b = b; }

    double Addera( )      { return a + b; }
    double Subtrahera( ) { return a - b; }
    double Multiplicera( ) { return a * b; }
    double Dividera( )   { return a / b; }
}

```

Klassen beräkning implementerar fyra interface.

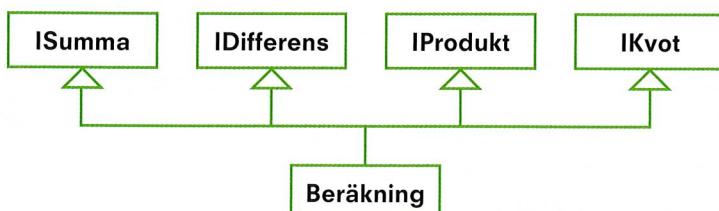
Klassen Beräkning ärver ingen kod från interfacen, och måste implementera alla metoder som de kräver, som så att säga ingår i kontraktet. Klassen som datatyp "är" en Beräkning, men "är" också en ISumma, en IDifferens och så vidare. Det betyder att en referens av typen ISumma, till exempel, kan peka på en Beräkning, och därmed delta i klasshierarkins polymorfism.

```

ISumma summa = new Beräkning( 3, 5 );
double resultat = summa.Addera();

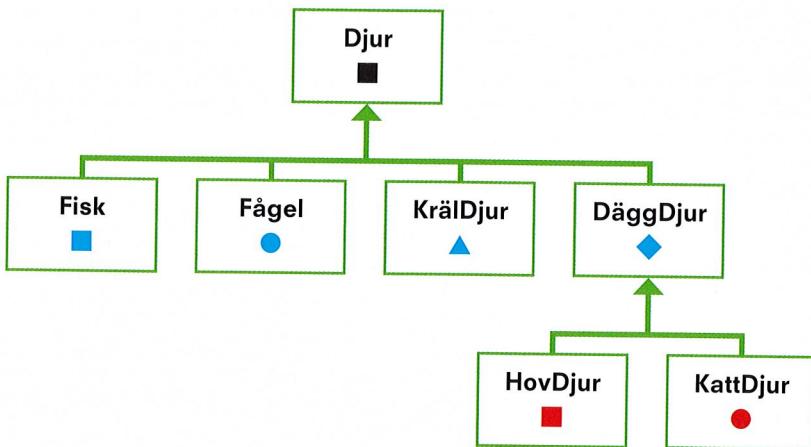
```

Ett diagram över hierarkin ovan skulle se ut så här:



## Att designa en klasshierarki

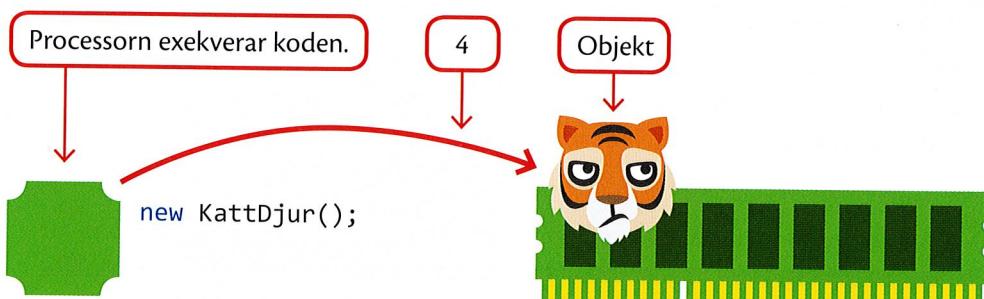
Klasser, interface och polymorfism är ganska avancerade koncept. Därför kommer här ett utförligt exempel, med lite repetition och resonemang kring en klasshierarki, och olika sätt att angripa ett problem. Antag att vi ska göra ett program som simulerar en djurpark. Det finns alla möjliga djur i parken med några gemensamma och många olika egenskaper. Vi bestämmer oss för en klasshierarki som nedan.



Låt oss säga att varje klass innehåller en metod. Basklassens metod heter `SvartFyrkant`. I underklasserna finns `BlåFyrkant`, `BlåCirkel`, `BlåTriangel` och `BlåRomb`. Underklasserna till `DäggDjur` har metoderna `RödFyrkant` och `RödCirkel`.

Antag också att klassen `DäggDjur` har ett heltal som heter `antalBen`, som ska ha värdet 4 som standard.

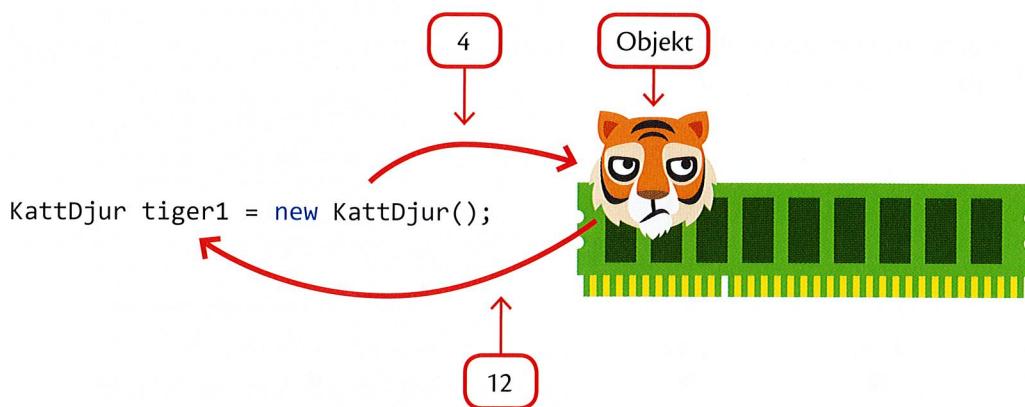
Fundera nu på vad det innebär att skapa (konstruera) ett objekt, till exempel ett `KattDjur`.



Tigerhuvudet på arbetsminnet står för ett `KattDjur`-objekt vars värde fylls av konstruktorns kod. Till exempel ska heltalet `antalBen` ha värdet 4. Minnesadressen till objektet är ett heltal, säg 12.

Konstruktorn är meningslös att anropa utan att ta hand om objektets adress. Den skapar ett objekt som inte går att hitta igen. Precis som det behövs husnummer för att hitta ett hus på en lång gata, behövs en minnesadress för att hitta ett objekt i arbetsminnet. Det behövs en variabel som lagrar adressen.

Om objektets address i minnet är 12, får referensen `tiger1` värdet 12. En variabel som innehåller en minnesadress kallas för en referensvariabel. Med den kan objektet hittas igen, så att man till exempel kan skriva `tiger1.AntalBen` för att få tillbaka 4:an till processorn.



Alla minnesadresser är positiva heltal. Så varför inte skriva  
uint tiger1 = new KattDjur()?

I maskinkod finns inte datatyper på samma sätt som i ett högnivåspråk, så vad spelar det för roll om man deklarerar tiger1 som KattDjur, när den ändå bara är en uint egentligen? Jo, kompilatorn har till uppgift inte bara att skapa maskinkod, utan att skapa maskinkod som ger förutsägbara resultat. Man måste bestämma en datatyp för referenser, bara för att kompilatorn ska veta vad de pekar på. Då kan kompilatorn se till att vi slarviga människor inte skriver sånt som tiger1.Parse(), eller tiger1.Sort(). Den kräver att det alltid står något efter punkten som faktiskt är körbart när programmet exekverar. Till exempel tiger1.RödCirke1().

Betrakta följande två rader:

```
KattDjur tiger1 = new KattDjur();
Djur      tiger2 = new KattDjur();
```

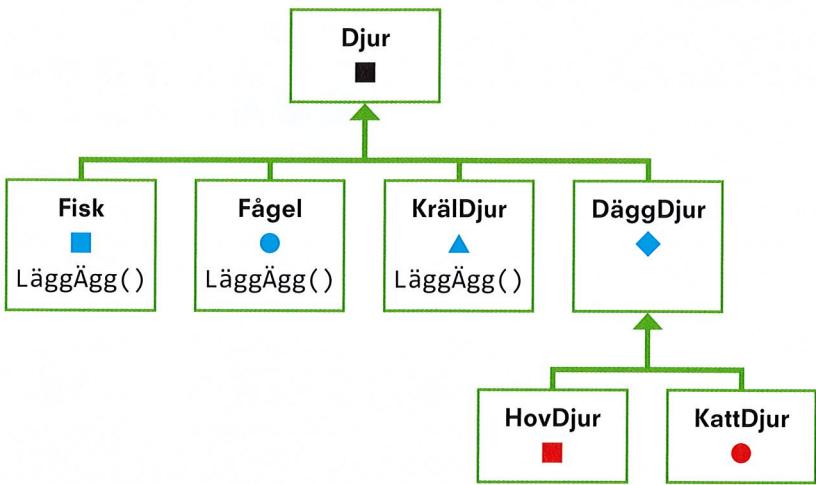


De båda satserna gör egentligen exakt samma sak. De skapar två objekt på två olika platser och med två olika adresser, men värdena i objekten är identiska. Båda objekten innehåller ett heltal som är 4. Skillnaden ligger i hur referenserna sedan kan användas:

```
tiger1.BlåRomb();
tiger2.BlåRomb();
```

Kompilatorn godkänner den första satsen, men inte den andra. Även om det för oss människor är solklart att båda referenserna pekar på identiska objekt, är kompilatorn benhård och tillåter inte att man rör BlåRomb med en Djur-referens. tiger2 kan bara anropa SvartFyrkant, för det är allt som finns i klassen Djur. Det är en bra säkerhetsåtgärd, för det är vanligt med kod där det inte är lika solklart vad en referens pekar på. Ta gärna en titt på exempel 2.1 igen.

Antag nu att vi vill lägga till metoden LäggÄgg, för att simulera förökningen bland djurparkens äggläggande djur.



Det är snyggast om bara Fisk, Fågel och Kräldjur kan lägga ägg. Men det finns ett litet programmeringstekniskt problem med att ha LäggÄgg bara i dessa klasser. Man kan nämligen bara anropa dem med referenser av typen Fisk, Fågel eller Kräldjur. Samtidigt vill vi ha alla referenser i en lista som vi kan loopa igenom, och den måste innehålla Djur-referenser.

```
List<Djur> parkensDjur = new List<Djur>();
```

Efterhand som vi skapar olika sorters objekt kommer referenserna i listan att peka på alla möjliga djur, men det går bara att anropa SvartFyrkant med dem.

```
parkensDjur[0].SvartFyrkant(); // OK.
parkensDjur[0].LäggÄgg(); // Aja baja, referensen är
                            en Djur-referens.
```

Så hur löser vi det? Ett alternativ är att kontrollera objektens typ vid exekvering, och bara välja att anropa LäggÄgg för objekt av rätt typ:

```
foreach ( Djur djur in parkensDjur )
{
    if      ( djur is Fisk )      (djur as Fisk).LäggÄgg();
    else if ( djur is Fågel )     (djur as Fågel).LäggÄgg();
    else if ( djur is Kräldjur )  (djur as Kräldjur).LäggÄgg();
}
```

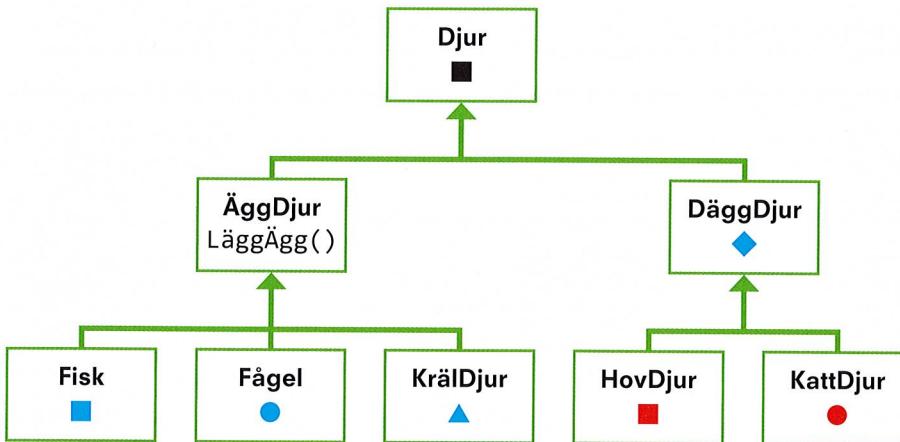
Det ser lite tråkigt ut. Det blir ännu tråkigare när vi ska uppgradera programmet till version 2, med stöd för insekter, som ju också lägger ägg. I en stor klasshierarki blir det många `is` och `as` som ökar röran i redan svår begriplig kod.

Man skulle kanske heller göra en virtuell LäggÄgg-metod i Djur-klassen, (som inte gör något), och overrides i de underklasser som behöver lägga ägg. Då skulle man kunna anropa LäggÄgg på alla objekt.

```
foreach ( Djur djur in parkensDjur )
{
    djur.LäggÄgg();
}
```

Problemet med denna lösning är att när vi lägger till fler metoder, (ÄtGräs, Jaga-Svansen, HoppaHögt) behövs fler virtuella metoder i basklassen. Förr eller senare blir det orimligt att ha en massa virtuella metoder som ärvs av alla underklasser, trots att de är irrelevanta för många av dem.

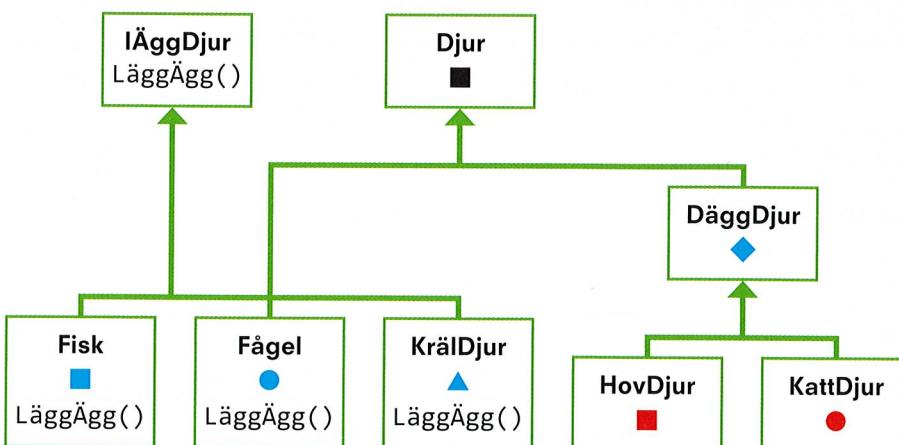
Ett tredje alternativ är att göra en basklass till alla äggläggande djur.



Då behövs bara en is-as-operation för att lägga ägg:

```
foreach ( Djur djur in parkensDjur )
{
    if ( djur is ÄggDjur ) (djur as ÄggDjur).LäggÄgg();
}
```

I det här fallet har äggdjuren egentligen inte mycket gemensamt mer än just att de lägger ägg. Om det enda som förenar några klasser är någon enstaka metod kan bas-typen lika gärna få vara ett interface.



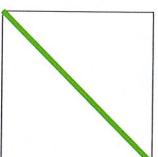
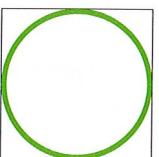
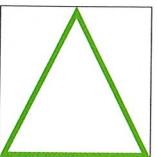
Koden för att anropa LäggÄgg blir nästan identisk som ovan. Det är bara att byta ut ordet ÄggDjur mot IÄggDjur.

Som du ser finns det en hel del överväganden att göra när man designar en klasshierarki. Ofta finns det ingen perfekt lösning. Det är okej att använda `is` och `as` här och var för att anropa metoder. Det är okej att ha en eller annan virtuell metod i basklassen även om den är irrelevant för en del underklasser.

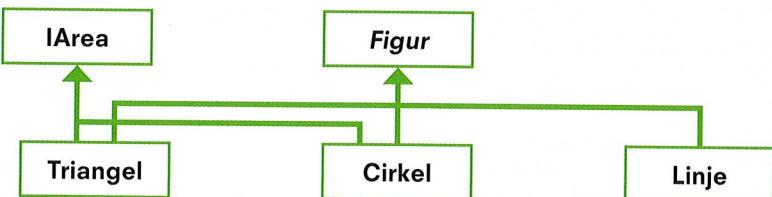
Sträva efter att göra dina klasser och hierarkier så lättfattliga, okomplicerade och oberoende som möjligt. Oberoende innebär att koden i en klass ska anropa koden i andra klasser så lite som möjligt. Det måste finnas en eller några klasser i en applikation, till exempel fönsterklassen, som så att säga binder samman allting, men `DäggDjur`-koden ska helst bara hantera sina egna variabler och inte anropa något i `Kräldjur`, för då bygger man in ett beroende mellan klasserna som gör det svårare att ändra och förbättra den ena utan att man måste ändra i den andra.

### Övning 2.4 Clipart

Tänk dig ett program som ska hantera några geometriska former, till exempel som clipart i Word. Formerna är likbent triangel, cirkel och rät linje. Vi tänker oss att varje form definieras av en tänkt omslutande rektangel med en viss bredd och höjd, (se figurerna). Cirkeln har alltid samma bredd och höjd. En linje kan ha  $\text{bredd}=0$  om den är lodrät eller  $\text{höjd}=0$  om den är vågrät. Det ska gå att beräkna arean för alla figurer som har en area (trianglar och cirklar).



Nedan finns ett förslag på klasshierarki..



Skapa ett program med den här klasshierarkin, så att man kan lägga till figurer med olika form i en lista. För enkelhets skull visar fönstret bara formerna i text. Programmet ska kunna beräkna den totala arean för alla formerna i listan. Detta går att lösa på flera olika sätt, precis som med `LäggÄgg` i förra exemplet.