Programmering, Fördjupning

Contents

[Repetition utav grunder 2](#_Toc65653678)

[Variabler 2](#_Toc65653679)

[Metoder 3](#_Toc65653680)

[Villkor och loopar 4](#_Toc65653681)

[Debugging 5](#_Toc65653682)

[Felhantering 6](#_Toc65653683)

[Throw 7](#_Toc65653684)

[Finally 8](#_Toc65653685)

[Testning 8](#_Toc65653686)

[Arrange, act, Assert 9](#_Toc65653687)

[TDD (test driven development) 10](#_Toc65653688)

[Sortering. Bubblesort; quicksort 10](#_Toc65653689)

[C# Historia 11](#_Toc65653690)

[Dokumentation 11](#_Toc65653691)

[Objektorientering och klasser 12](#_Toc65653692)

[Inkapsling 12](#_Toc65653693)

[Klasser och objekt 13](#_Toc65653694)

[Value type och refrence type 13](#_Toc65653695)

[Static 14](#_Toc65653696)

[Structs 15](#_Toc65653697)

[Enums (enumerations) 15](#_Toc65653698)

[Klass-arv 15](#_Toc65653699)

[Virtual/overide. Modifiera/skriva över ärvda metoder i en klass. 17](#_Toc65653700)

[Hantera flera objekt utav olika typer. 17](#_Toc65653701)

[Null check 18](#_Toc65653702)

[Abstract 18](#_Toc65653703)

[Interfaces 18](#_Toc65653704)

[Generics 19](#_Toc65653705)

[Operator Overloading 21](#_Toc65653706)

[Collections 23](#_Toc65653707)

[<List> 24](#_Toc65653708)

[Queue 24](#_Toc65653709)

[HashSet 25](#_Toc65653710)

[Dictionary 25](#_Toc65653711)

[LINQ Language Intergrated Query 26](#_Toc65653712)

[Hämta vissa data och sortera data från en array med objekt. 26](#_Toc65653713)

[Join, slå ihop och filtrera data från två olika samlingar. 28](#_Toc65653714)

[Callbacks 28](#_Toc65653715)

[Delegates 29](#_Toc65653716)

[Anonyma metoder 31](#_Toc65653717)

[Asynkron kod 31](#_Toc65653718)

[Task<> 33](#_Toc65653719)

[PLINQ, Parallel Language Intergrated Query 34](#_Toc65653720)

[Parprogrammering 34](#_Toc65653721)

# Repetition utav grunder

### Variabler

Int = heltal = 2

Float = decimaltal = 3,3f

Double = decimaltal (större) = 3,3

Decimal = decimal tal (exakt) = 3,3m

String = text strängar = ”hej” (måste skrivas med ” ”, inte ’ ’)

Char = enstaka tecken = ’c’

Console.writeLine(”hej”) = en funktions om skriver ut ett värde angett i parenteserna.

**Console.writeLine(”hej”);**

**Hej**

Deklarera en variabel:

**int num = 12;** variabel typ, identifierare (namnet du anger), tilldelnings tecken (=), värde.

Konvertera en variabel

**String aNum = ”12”;**

**Int num = int.parse(aNum)**

Skriva ut flera värden.

**String a = ”Min ålder är: ”;**

**Int num = 24;**

**Console.WriteLine(a + num.ToString());**

Med ToString() metoden så kan man konvertera värden till string.

**Min ålder är: 24**  resultatet

Det finns ett bättre sätt att göra på. Ovan är ganska prestanda krävande.

**Console.WriteLine($”{a}{num}”);**

**Min ålder är: 24** resultat

Man kan skriva ganska mycket information så här.

**Console.WriteLine($”{a}{num}\nDin ålder är 25”);**

**Min ålder är: 24**

**Din ålder är 25** Med \n så kan du göra nya rader. Finns fler \ kommandon.

Vi kan också samla flera värden utav samma typ i **Arrays**.

**String[] namn = new string [] {Jacob, Gustav, Anders, Stefan}** Anger en array med 4 platser och anger värdena. Det går inte att ändra på storleken utav en array efter den är skapad.

**Int[] ålder = new int[4];**

Om man ska skriva ut ett vist värde i en array så måste man ange dess position i arrayen. Positioner i en array börjar från 0.

Metoden **”.Lenght”** anger hur många positioner en array har.

### Metoder

Man kan skriva stycken utav kod som man sedan kan anropa och köra dem flera gånger.

Detta kallas en metod eller en funktion.

Man skriver dem utanför mainmetoden.

**Static void main(string[] args)** main metoden

**{**

**Int i = 3**

**Int j = 34**

**Int k = Addition(I, j);**

**Console.WriteLine(k);**

**}**

**Static Int Addition(int tal1, inbt tal2)** en additions metod

**{**

**Int result = tal1 + tal2;**

**Return result;**

**}**

Ovan så har jag deklarerat: retur data typ (**Int**), identifierare(**addition**) och inputparametrar (**int tal1, int tal2**). Det går att skapa och anropa metoder utan inputparametrar.

Om metoden inte ska returnera något så använder man **void** som returvärde.

Resultatet utav ovan kod skulle bli en utskrift utav 37.

Man kan också skriva metoder med en pil. Additions koden ovan skulle kunna skrivas så här:

**Static Int addition(int tal1, int tal2) => tal1 + tal2;**

Man kan även ge inputparametrar ett standardvärde så att metoden kan användas även om man inte skickar in lika många inputparametrar som metoden deklarerades med.

**Static Int addition(tal1, tal2, tal 3 = 0;) => tal1 + tal2 + tal3;**

**Statci void main(string[] args)** main metoden

**{**

**Int i = 3;**

**Int j = 34;**

**Int k = Addition(I, j);**

**Console.WriteLine(k);**

**Int x = 3;**

**Int o = Addition(i, j, x);**

**Console.WriteLine(o);**

**}**

Viktigt att veta i C# är att variabler som skapas i en metod finns bara tillgänglig i den metoden.

### Villkor och loopar

Med villkor kan man diktera vad som ska göras beroende på olika förutsättningar. Med en **if**-sats så kan man göra några saker om ett villkor är sant.

**Int num = 2;**

**If (num<3)**

**{**

**Console.WriteLine(”hej”);** om 2 är mindre än 3 så skriv hej.

**}**

Med switch-satser kan man bestämma vad som ska göras beroende på ett värde.

**Switch (num)**

**{**

**Case 1:**

**Console.WriteLine(”blä”);**

**Break;** Du måste ha ett break med varje switch case.

**Case 2:**

**Console.WriteLine(”hej”);** Detta skulle skrivas då num = 2.

**Break;**

**Case 3:**

**Console.WriteLine(”fisk”);**

**Break;**

**}**

Man kanske vill göra något så länge ett villkor stämmer. Då kan man använda sig utav loopar.

**While (true)** medans sant är sant så gör x. Detta skulle forma en oändlig loop. Kommer aldrig att ta sig ut ur den om man inte anger ett villkor som resulterar i en **break;.**

**{**

**x**

**}**

**For (int i = 0; i<arr.Lenght; i++)** Gör x lika många gånger som det finns positioner i arrayen ”arr”.

**{**

**x**

**}**

**Foreach (int element in arr)** gör x för varje position i ”arr”. ”element” är en variabel som tar värdet på positionen i varje iteration

**{**

**x**

**}**

# Debugging

Om man vill kunna hänga med lite när man debuggar och undersöka vilka värden olika variabler får samt hur de hanteras så kan man debugga i så kallat **break mode**. Detta gör man genom att trycka på F10 eller F11. Du kan då se alla variabler som finns i det scoopet du är i (vart i programmet du är) samt deras värden i det nedre fältet. Man kan också skapa en så kallad **breakpoint** genom att trycka på den raden (till höger) som man vill köra i break mode från. Detta skapar en liten rödprick som programmet kör till och väl där så kan du välja att stega dig fram i koden eller låta den fortsätta själv.

Väl inne i break mode så kan du stega fram koden bit för bit med F10 och F11. F11 är kommandot för **step into** vilket hoppar till nästa steg i koden och visar allt. F10 är kommandot **step over** och när du använder det så hoppar den över de separata stegen i metoder och funktioner. De utförs fortfarande men du slipper att stega igenom dem om du vet att man kan lita på att det funkar.

# Felhantering

När man hanterar data och matar in det i programmet på olika viss så kan det hända att man får situationer som inte fungerar, tex delar med noll. Detta kallas **exception error**. Detta skulle krascha programmet om det inte hanteras.

Ett enkelt sätt att hantera dessa problem är att använda if-satser för att skapa värden som inte kan användas.

Tex om du vill undvika att det går att dela med noll.

**Int num1 =2**

**Int Num2 = 0;**

**Int result = =;**

**If (num2 != 0 )** om num inte är 0 så utför vi det vi ville göra.

**{**

**Result = num1 / num2**

**}**

Om du ska ta emot data från användaren så kan det också skapa problem om du inte ser till att konvertera det strängvärde du får tillbaka eller om användaren skriver fel.

Det vanligaste sättet att hantera fel är **Try** och **Catch**.

Med dessa två nyckelord så kan man skapa kodblock som försöker köras men om ett exception (ett fel) händer så hoppar man till Catch kodblocket där man kan skriva vad som ska hända om något går snett.

Ex:

**Try**

**{**

**Int num = int.parse(console.readline());** input från användaren med konvertering.

**Int num2 = int.parse(console.readline());**

**Int result = 0;**

**}**

**Catch (FormatException Ex)** Felhantering för fel utavanvändaren. Ex = variabel för felhantering

**{**

**Console.WriteLine(Du skrev något fel);**

**}**

**Try**

**{**

**Result = num / num 2**

**}**

**Catch (DivedeByZerroException Ex)** Hanterar beräkningsfel

**{**

**Console.writeLine(Ex.message)** Man kan skriva ut vilket fel meddelande man får. ex.message = felmeddelandet

**}**

Man kan också bara skriva;

**Catch**

**{**

Felhantering

**}**

För att hantera alla fel som kan uppstå men det är rekommenderat att vara specifik.

## Loga felmeddelanden

Om något i det problem krashar när användaren kör det så kan det vara bra att kunna läsa felmeddelandet efter att det sket. Ett sätt att göra detta är att logga felmeddelanden i en separat fil. Man kan sedan använda klassen ”**StreamWriter**” för att skriva i den externa filen. Med denna klass så kan man skapa ett objekt med ett antal olika konstruktorer. En utav dessa tar ett strängvärde som ska vara en filsökväg till den filen som man vill skriva felmedelandet. Klassen ”StreamWriter” finns i **System.IO** namespace/namndrymd som är en namndrymd som innehåller typer som låter skriva i externa filer.

**Using System.IO**

Om vi nu vill logga vårt felmeddelande så gör vi det i ett catch-block.

**Catch (exception ex)**

**{**

**StreamWriter streamWriter = new StreamWriter(@"C:\Users\Gustav\source\repos\Test\FelMeddelanden\log.txt");**

Notera @. Då \ har en specifik funktion inom strängar så måste man använda ett @ framför strängen så kompilatorn vet om att det inte är en funktionell backslash. @ tar bort alla funktionella tecken i en sträng.

Om filen man skriver inte finns så skapas en ny fil med namnet du anger.

För att sedan använda det StreamWriter-objektet du skapat och skriva något i filen så använder man metoden ”**.Write()**” och stoppar in den egenskapen utav felet som du vill se.

**streamWriter.Write(ex.GetType().Name);**

Om du vill skriva något mer så är det bara att lägga till fler .write metoder. Du kan också använda ”**.WriteLine**” om du vill att det som skrivs **efter** ska hamna på en ny rad. WriteLine skapar i detta fal en ny rad efter texten som ska skrivas.

**streamWriter.WriteLine(ex.message);**

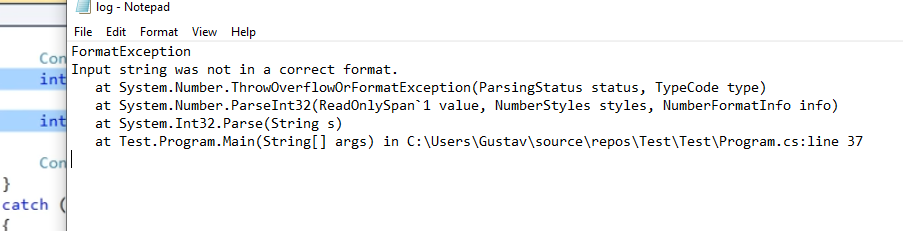
**streamWriter.WriteLine(ex.StackTrace);**

När du skrivit det du vill skriva så måste du avsluta streamWriter med metoden ”.Close()”

**streamWriter.Close();**

Om det nu blir något fel så kommer det att skrivas vad namnet på den typen utav fel är i filen du angav.

Ex:



## Throw

Ibland så kan man se att koden kan hamna i ett läge där man inte vill ha den. Tänk att du skriver en metod som tar ett heltal(int) och returnerar namnet på den månaden som talet representerar. Så om du skickar in 1 så får du tillbaka januari. Men vad ska returneras om man skicar in 13, 0, -2 eller 1002. Helst så ska inget returneras, utan man skulle hellre vilja ange att ett fel har inträfat och något fel meddelande ska skickas. Med ”**throw**” så kan man ange ett fal

Om detta händer så vill man kunna skriva ut någon form utav fel kod. Man vill göra ett eget exception. Detta kan man göra med nyckelordet ”**throw**”. För att kasta ett fel så behöver man skapa ett exception-objekt och man kan också skriva en text som följer med detta objekt.

Säg att vi vill bygga metoden nämnd ovan:

**Static string Month(int monthNr)**

**{**

**Switch (monthNr)**

**{**

**Case 1:**

**Return ”Januari”;**

**Case 2:**

**Return ”Feburary”**

Etc…

**Default:**

**Throw new ArgumentOutOfRangeException(”Aj då, fel nummer.”);**

**}**

Notera ”ArgumentOutOfRangeException”. Detta är en specifik typ utav exception objekt som anger att argumentet som getts är utanför de gränser som är satta. Man kan bara skriva ”**throw new exception()”** med som jag skrev tidigare så är det bra praxis att vara noggrann. Dessa är egentligen bara olika typer utav klasser som ärver från en klass kallad ”Exception”. Mer nedan om hur du skapar en egen exception-klass att kasta.

### Finally

**Static string Month(int monthNr)**

**{**

**Switch (monthNr)**

**{**

**Case 1:**

**Return ”Januari”;**

**Case 2:**

**Return ”Feburary”**

Etc…

**Default:**

**Throw new ArgumentOutOfRangeException(”Aj då, fel nummer.”);**

**Finally** 🡨------Obs notera!!

**{**

**NågotNummer++;**

**}**

**}**

Säg att vår kod ser ut som ovan. Notera ”**finally**”blocket. När ett exception sker eller kastas så kan det komma att kod som kommer efter inte körs. Om man har något viktigt som måste köras så kan man lägga det i ett finally block. Kod som ligger där kommer alltid att köras även om ett exception kastas.

## Skapa egna exceptions.

Som tidigare nämnt så går det att skapa egna exception-klaser att kasta. Man gör det genom att skriva en ny klass som ärver från en klass kallad ”Exception”. Om du kollar igenom alla olika exception-klaser så kan du se att deras namn alltid slutar med ”Exception”. Det är bra braxis att om du ska skapa egna exception klasser så ska du också namnge dem med Exception i slutet så att det är tydligt vad det är för sorts klass du har skapat.

**Class UserAlreadyLogedInException: Exception**

**{**

**}**

Om du skapar en egen exception-klass så kan du inte skicka in ett felmeddelande, som ska visas när undantaget kastas, per automatik.

**Throw new UserAlreadyLogedInException(”Du är redan inlogad”)** Funkar ej.

För att kunna göra detta så krävs det att du har en konstruktor som kan ta emot ett argument när undantags-objektet initieras.

Det smidiga med att ärva från bas klassen ”Exception” är att den har en konstruktor som tar ett string argument som vi kan implementera via arv i vår egen klass.

**Class UserAlreadyLogedInException: Exception**

**{**

**Public UserAlreadyLogedInException(string message) : base(message)**

**{**

**}**

**}**

När vi har skapat en konstruktor som tar ett argument så skapas dock ett annat problem. Och det är att den automatiska konstruktorn med inga parameter som skapas när ingen annan konstruktor finns, försvinner när vi skapar en egen. Om vi ska skapa ett objekt utav denna klass så måste vi alltså skicka in ett meddelande men vi kanske vill kunna kasta ett undantag utan att göra det.

Vi måste alltså ange ytterligare en konstruktor utan parameter. Smidigt nog så finns det även en sådan att ärva från bas klassen.

**Class UserAlreadyLogedInException: Exception**

**{**

**Public UserAlreadyLogedInException() : base()**

**Public UserAlreadyLogedInException(string message) : base(message)**

**{**

**}**

**}**

## Testning

Det är ganska vanligt att man använder sig utav separat kod för att testa den koden man skriver.

Man gör detta för att verifiera att alla krav uppfylls och för att säkerställa att inget är trasigt.

När man testar fortgående så hjälper det även att se till att man tänker på prestanda och användbarhet.

För att skapa ett test för en fil så högerklickar man varstans i filen och väljer **create unit test**.

Du kan här namnge testfilen och ändra vilket ramverk det ska använda och mer. Klicka OK.

Det skapar då ett nytt projekt som du kan hitta till höger i **Solution explorer**.

I detta projekt så skapas en test-metod per metod som du skrivit i det originella projektet.

Du har ej någon main-metod i testprogram.

Istället så har du:

**[TestMethod()]**

Och sedan dina testmetoder.

När man är i sitt testprogram så skriver man **filnamn(klassnamn).metod(metoden du vill testa)**:

I koden nedan så skickar vi två heltal och en operator till en metod som vi skrev tidigare och nu ska testa så att den fungerar. Metoden gör beräkningar med de två nummer och operatorn.

När man ska testa kod med annan kod som ligger i en separat fil så är det viktigt att man gör metoderna som ska testas **public** så att de kan kommas åt.

I testprogrammet:

**Int num1 = 2;** Skapar våra variabler som vi ska skicka till metoden vi testar.

**Int num2 = 3;**

**String op = +;**

**Int result = Program.compute(num1, num2, op);** Vi anropar metoden som ska testas.

Sedan använder man nyckelordet **CollectionAssert** eller **Assert** för att beskriva vad resultatet ska bli. Med **collectionAssert** så kan du jämföra flera värden (alltså en hel kollektion) på en gång. Till exempel alla positioner i en array.

**Assert.AreEqual(result, 5);** Vi skriver att det svaret vi får tillbaka från result ska vara 5.

För att köra tester efter att man skrivit dem så högerklickar man på projektet i solution explorer och väljer **run tests**.

### Arrange, act, Assert

När man skapar en testmetod så ska man försöka bygga en struktur med orden Arrange, act och assert.

Arrange innebär är att man ska först göra en sektion där man bestämmer vilken data man ska skicka in i metoden genom att skapa variabler för all indata samt en variabel för det väntade svaret.

Act innebär att man agerar med den data man skapade. Dvs att man anropar metoden med in variablerna.

Assert innebär att man jämför return-värdet (output från metoden man testar) med den variabeln du skapade som motsvarar det väntade svaret.

Ex

Jag har en metod som adderar två tal.

I min arrange del utav testmetoden så skapar jag två variabler som ska skickas in i metoden samt en variabel som är de andra två adderade.

I act anropar jag metoden.

I assert så ser jag till att metoden returnerar samma värde som jag väntade mig.

**[TestMethod]**

**Public void AdditionTest** metoden som ska testas heter ”Addition”

**{**

**//assert**

**Int num1 = 2;**

**Int num2 = 3;**

**Int expected = 5;** skapar variabler att skicka till metoden samt en variabel som representerar mitt förväntade svar. 2+3=5

**//act**

**Int act = Program.Addition(num1, num2);** anropar metoden med data. Sparar svaret

som en variabel.

**//assert**

**Assert.AreEqual(act, expected);** ser till att svaret och det väntade

svaret är detsamma.

Jag kan också göra så här.

**Assert.AreEqual(Program.Addition(num1, num2), expected);**

## TDD (test driven development)

TDD innebär att man skriver testet redan innan man skriver den koden man har som uppdrag att skriva. Beställde

# Sortering. Bubblesort; quicksort

Bubblesort är en sorterings algoritm som kan sortera kollektioner med data. Den är ganska enkel men värt att veta är att den är väldigt oeffektiv och prestanda krävande så använd den ej med större mängder data.

Bubblesort algoritm tagen från stack overflow:

for (int write = 0; write < arr.Length; write++) {

for (int sort = 0; sort < arr.Length - 1; sort++) {

if (arr[sort] > arr[sort + 1]) {

temp = arr[sort + 1];

arr[sort + 1] = arr[sort];

arr[sort] = temp;

}

}

}

En bättre sorteringsalgoritm är **quicksort**. Den är betydligt effektivare.

Algoritmen ser ut något så här:

static int Partition(int[] array, int low,

int high)

{

//1. Select a pivot point.

int pivot = array[high];

int lowIndex = (low - 1);

//2. Reorder the collection.

for (int j = low; j < high; j++)

{

if (array[j] <= pivot)

{

lowIndex++;

int temp = array[lowIndex];

array[lowIndex] = array[j];

array[j] = temp;

}

}

int temp1 = array[lowIndex + 1];

array[lowIndex + 1] = array[high];

array[high] = temp1;

return lowIndex + 1;

}

Värt att anmärka är att det fins inbyggda sorteringsfunktioner I VS så man behöver inte bygga dem manuellt.

# C# Historia

Började utvecklas 99 av Microsoft, lett utav Anders Hejlsberg. Skapades i samband med .net ramverket.

Skapat för att vara liknande C, men simpelt och objektorienterat.

Senaste varianten är C# 9.0 men det utvecklas pågående.

# Dokumentation

När man kodar så är det viktigt och vanligt att man skriver ett separat textdokument där man beskriver funktionaliteten och andra viktiga saker att veta om koden. Det är väldigt viktigt att göra när man börjar jobba med andra personer så att alla kan använda sig utav all kod utan att konstant behöva fråga varandra hur saker fungerar.

Ett sätt att skriva dokumentation är med kommentarer.

En standard är att man skriver något om varje funktion. Man ska beskriva:

* en summering utav vad funktionen gör.
* Beskriver varje inputparameter.
* Vad den returnerar.
* Hur den hanterar fel (exceptions).

Ett bra sätt att kommentera metoder är att använda 3 snedstreck (///) och sedan htlm liknande taggar. Om du skriver tre /// så kommer visual studio att automatiskt generera en summering, parameternamn och return så behöver du enbart fylla i dessa.

Exempel:

**/// <summary>**

**/// A method that makes computations.**

**///</summary>**

**///<param name =”num1”>The first interger in the computation.</param>**

**///<param name =”num2”>The second interger in the computation.</param>**

**///<param name =”op”>The operator that decides which type of computation that is executed. Available operators are /, \*, - and +.</param>**

**///<returns>Returns an interger that is the result off the computation.</returns>**

När man sedan anropar metoden och håller muspekaren över den, samt när man skriver in parameter så kommer ett litet fönster att visas där den beskrivning du skrev visar sig.

# Objektorientering och klasser

## Inkapsling

Inkapsling innebär att man kapslar in funktionalitet i olika metoder och liknande. Detta innebär att vi behöver inte nödvändigtvis vetta exakt vad som händer. Som exempel när vi skriver **Console.WriteLine()** så vet vi egentligen inte vad som händer i metoden men vi behöver ju inte det heller.

Istället för att förklara hur något funkar från grunden så nöjer vi oss med det resultat det ger. Vi jobbar med olika kapslar som gör olika saker. Vi vet hur man kan använda dem men inte nödvändigtvis vad som är i dem.

**Procedurell programmering** = att man delar upp det i olika metoder.

Inkapsling innebär också att man skyddar vad som är i kapseln (tex vad en funktion som Console.WriteLine() gör så att användaren (programmeraren i detta fal) inte kan ta sönder något utav misstag.

## Klasser och objekt

Objektorientering är något som förekommer i många olika språk men de kan skilja sig åt ganska så drastiskt.

Vi skapar objekt och klasser för att kunna gruppera olika data och metoder samt annan funktionalitet.

Ett objekt är ett sätt att samla olika former utav data i en samling. Om man har en person vars namn och ålder man ska hålla koll på så kan det bli svårt om man ska text använda Array:er. Då kan det vara enklare att skapa objekt.

En klass är som en mall för ett objekt. Det beskriver vilka egenskaper ett objekt har samt vad man kan göra. Man skapar sedan olika instanser utav klassen och dessa instanser är objekt. Vad data är i dem kan skilja åt men de har alla samma sorters data. Ex vi har olika namn, ålder och funktionalitet.

För att skapa en klass så högerklickar man på solution Explorer och väljer **add**.

Sedan väljer man klass och klickar OK. Då skapas en ny fil som du kan öppna i en ny flik.

I klasser kan man skapa globala variabler som kallas **fält/fields**. Dessa brukar man skapa en så kallad **kostruktor** tillvilket är en metod som till delar värden till fälten.

För att skapa en konstruktör så skriver man:

**Public int Ålder {get; set;}** variabeln är ”ålder”. {get; set;} är konstruktorn.

Man kan också ange om värden ska gå att ses utifrån klasser men inte ändras eller tvärt om. Ex:

**Public int ålder {get; private set;}** get är public men set är private.

När man gör så måste man använda konstruktorn för att ange ett värde på en klass variabel.

**Public** nyckelordet anger att det går att komma åt den variabeln från en annan klass/fil.

Variabler som skapas som public ska generellt sett namnges med storbokstav.

För att göra jämförelser mellan olika objekts variabler med en funktion i samma klass så kan man skriva:

**Variabel.other**

för att förklara att man syftar på ett annat objekts variabel.

Ex:

**Int differenceOfX = X – X.other;** samma variabel men olika objekt.

För att använda sig utav variabler och metoder i en klass från main-metoden så skriver man:

**Klassnamn.variabel**

**Ex**

**Console.WriteLine(Class1.Variabel1);**

### Value type och refrence type

Det finns två olika vis att spara data i en dator.

Value type innebär att man dedicerar en liten bit minne för att spara ett unikt värde. Detta är vad som använder med minder variabler som int, string etc. Detta kallas **value type**.

Större datastruktur som arrays och objekt fungerar dock inte så. Identifieraren till en variabel/datastruktur utav en större datatyp sparas enbart som en referens till vart i minnet alla värden sparas. Så när man använder den identifieraren så pekar man på en specifik plats i minnet. Detta kallas en **reference type.**

Detta skapar en sak att tänka på. Om du ska skapa två olika data utav reference type som ska vara lika, kanske två lika stora arry:er, så kan du inte använda den första som en mall för den andra.

**Int[] array1 = new int[]{1, 2, 3}** jag vill skapa en till, likadan, array.

**Int[] array2 = array1**

Ovan funkar inte. Vad jag gjort är att skapa två referenser till samma position i minnet. Alltså två adresser till samma data.

För att göra en referens till en value type variabel likt referende type så kan man använda nyckelordet **ref**. Om du till exempel ska skicka med en variabel som en in-parameter till en metod så kan du skriva ref framför variabeln när du anropar metoden.

Man brukar kalla den delen utav minnet där value-type data sparas **stack**.

Den delen utav minnet där reference type data sparas kallas **heap**.

När du skapar en array eller ett objekt så sparas ”adressen” (vart i minnet) till objektet i stack:en men själva data sparas i heap:en.

### Static

Metoder som är skrivna som statisk tillhör klassen det är skrivet i, och inte några objekt som är skapat efter klassen.

Det går inte att använda fält(klassvariabler) från en klass i en statisk metod skriven i samma klass då de fälten tillhör objekten som skapas och de får enbart värden då en objektinstans skapas. Det går dock att använda en statisk metod i en metod som konstruktorn eller liknande i ett objekt.

Vad detta är bra för är att för att anropa en metod som är skriven som statisk så behöver man inte skapa en instans utav objektet för att den ska vara tillgänglig utan man behöver bara ange klassen och anropa den.

Ett exempel vore **Console.WriteLine()**

Console är i sig en klass men vi har ju aldrig behövt skapa ett console objekt för att använda den.

Man kan se metoder som skrivs som statiska som metoder som är till ”allmänbruk” och kan komma att behöva användas utav flera olika klasser/instanser.

Det kan vara bra att försöka samla statiska metoder i en klass som är relevant för deras användningsområde.

Klasser som är skrivna som statiska kan ej användas för klass arv.

Om static inte är skrivet framför en metod så är de inte statiska som standard.

### Structs

**Struct** är ett nyckelord som man kan använda istället för class. Det är en annan typ utav klass där allt skapas på stacken istället för heapen som det vanligtvis gör med en klass. Den största skillnaden mellan de två är att alla värden och instanser blir value type i en struct istället för reference type.

En struct måste ha en konstruktor.

Struct kan inte ärva från andra klasser eller andra structs.

### Enums (enumerations)

Enums är variabler som kan skapas för att bara ha några specifika värden. Inga andra än vad som anges när de skapas godtas.

Ett exempel vore en operator enum. I vår beräkningsmetod så använde vi ett string värde som operator i en beräkning. Om vi vill undvika att det blir fel så kan vi istället använda en enum som bara har +, -, \* och / som alternativ.

Enums skapas som klasser eller structs utanför program klassen.

När man ska använda ett enum värde så skriver man **identifierare.värde.** Likt ett objekt.

**Enum Operator {+, -, \*, /};**

**Enum Directrion {Up, Down, Left, Right};**

**Class Program**

**{**

**Static void Main {string[] args}**

**{**

**Compute(2, 3, Operator.+)**

**}**

Enum värden är egentligen Int-värden med ett namn för varje heltal. Ovan är + = 1.

## Klass-arv

Om man har flera olika objekt som delar egenskaper (fält/klass variabler) och metoder så kan man skapa en gemensam klass som ärver dessa saker.

Man gör det genom att skapa en klass med de egenskaperna och metoder som ska delas (detta kallas **basklass** eller **base**) och sedan så lägger man till: ”**:klassnamn”** efter den nya klassen som ska ärva egenskaperna.

Ex,

ska skapa en klass som kan skapa olika personer i en skola. Jag ska skapa lärare och studenter. Både lärare och studenter har ett namn och en ålder som ska anges men lärare ska också ha en lön medan studenter ska ha ett betyg. Jag kan skapa en klass som ger ålder och namn.

**Public Class Person**

**{**

**Public string Name;** egenskaper

**Public int Age;**

**Public Person (string n, int a)** konstruktor

**{**

**Name = n;**

**Age = a;**

**}**

**Public string NameAndAge (string name, int age)**

**{**

**Console.WriteLine($”My name is {name} and i am {age} years old.”);**

**}**

**}**

**Public Class Student: Person** notera ”**: Person”**. Indikerar att Student ärver från Person och då har allt som Person har dvs Name och Age.

**{**

**Public string Grade;**

**Public Student(string g): base(string n, int a )** base = referens

**{** till bas-klassen. Använder

Grade = g; bas-klassens konstruktor.

**}**

**Public overide NameAndAge (string name, int age)**

**{**

**Console.WriteLine($”My name is {name} and i am {age} years old.” +**

**”\nAnd i am a student.”);**

**}**

Notera metoden ovan med nyckelordet **overide**. Vi har modifierat metoden från bas-klassen så att när den anropas med en student så gör den något annat.

**}**

**Public class Teacher: Person** teacher ärver också ifrån person.

**{**

**Public int Salary**

**Public Teacher(int s): base(string n, int a)**

**{**

**Salary = s**

**}**

**}**

Klasser kan enbart ärva från en klass.

### Virtual/overide. Modifiera/skriva över ärvda metoder i en klass.

Om man vill använda en modifierad variant utav en metod från basklasen så kallar man det en **virtual, overide**. Man lägger till nyckelordet ”**Virtual**” till metoden i klassen so de andra klasserna ska ärva ifrån.

Seden skriver man en ny metod med samma namn/identifierare med nyckelordet ”**overide**”. Det skriver då över den ärvda metoden och den nya/modifierade metoden gäller för den klassen och de objekten som skapas efter den.

Alla klasser och objekt som skapas ärver alltid från en inbyggd klass kallat ”objekt”. Härifrån så ärver alla klasser 4 metoder. Dessa kallas: ”Equals”, ”GetHasCode”, ”GetType” och ”ToString”. Man kan använda virtual/overide för att modifiera dessa.

### Hantera flera objekt utav olika typer.

Om man vill loopa igenom en samling utav objekt och göra något när man har en viss typ utav objekt så kan man använda nyckelordet ”**is**” i en if-sats. Vi vill kanske skriva ut något om ett utav tidigare nämda Person-objekt är en Student).

Vi antar att vi har en array med objekt utav typen Person ovan som vi skapat. Array:en är kallad people.

**Foreach (var human in people)**

**{**

**If (human is Student)** kollar om ”Person” är ett object utav typen person.

**{**

**Consloe.Writeline(”Denna person är en student”)**

**}**

**}**

Om man vill hantera ett objekt som en viss typ utav objekt så kan man använda nyckelordet ”**as**”. Detta innebär att man då kan använda olika fält(klass-variabler) som den typen utav objekt har tillgång till. Nedan kan vi komma åt en Teacher.Salary genom att säga att vi vill behandla objektet som en Teacher.

Att hantera ett okänt objekt som en vis typ kalas polymorfism.

**Foreach (var human in people)**

**{**

**If (human is Teacher)**

**{**

**Consloe.Writeline((human as Teacher).Salary);**

**}**

**}**

När man gör på ett sådant sätt så måste man se till att objektet i fråga är utav typen som man säger att man vill behandla det som. Annars returneras null och så kraschar det.

### Null check

Med operatorn ”**?.**” så kan man kontrollera om ett uttryck returnerar null. Om det gör det så skippas resten utav uttrycket. Med detta så skulle vi kunna förenkla koden ovan och ta bort if-satsen som kontrollerar att objektet är utav typen Teacher.

**Foreach (var human in people)**

**{**

**Consloe.Writeline((human as Teacher)?.Salary);**

**}**

### Abstract

Ibland så har man en klass vars enda syfte är att ha vissa egenskaper och funktioner som ska ärvas utav andra klasser. Det är alltså inte tänkt att man ska kunna skapa några instanser (några objekt) utav den men som standard så går det fortfarande att göra det. I koden ovan så kanske vi enbart vill kunna skapa lärare och studenter men det går ju fortfarande att skapa personer. För att förhindra detta så finns nyckelordet ”**abstract**”. Det skapar en klass som det inte går att skapa några instanser utav.

Man kan sedan också skapa tomma metoder som är abstrakta. Man gör detta då man vill att alla klasser som ärver efter den abstrakta klassen ska ha en specifik metod som gör något, men vad kanske skiljer sig åt per klass. Om man då skriver en tom abstrakt metod så måste man sedan implementera en overide-metod för varje klass som har ärvt efter den klassen. Annars klagar kompilatorn.

### Interfaces

I C# så kan man inte ärva ifrån flera klasser. Man kan dock vilja göra det och för att komma runt den begränsningen så kan man använda sig utav ett interface.

Ett interface skapas likadant som en klass. Antagligen så använder du Solution explorer och skapar en ny fil eller så går du utanför programklassen och skriver det själv i samma fil. När du namnger ett interface så är det värt att veta att de ska enligt standard börja med ett stort ”I”.

Vad interfaces gör och är till för är att de fungerar som en extra klass att kunna ärva abstrakta metoder från. Till skillnad från abstrakta metoder så kan interfaces inte innehålla någon kod eller data dock. Om till exempel ska skapa några personer som tidigare. Då kan du först skapa en abstrakt metod som du ärver grunderna från. Om du sedan har ytterligare några metoder som styr hur objekten interagerar med varandra så kan du lägga dem i ett interface. Ofta just sådana metoder som man lägger i interface. Det finns ett inbyggt interface kallad ”**IComparable**” som gör just detta. Den innehåller en metod som kallas ”**CompareTo**” som man måste implementera i varje klass som har IComparable som ett interface. För att ange att en klass har ett viss interface så gör man likadant som med klass-arv:

**Abstract class Persson: IComparable<Person>**

Dett som står mellan <> kallas generics. Läs om detta senare.

Om en klass ärver från en basklass och ska implementera ett interface också så skriver man bas klassen först följt utav ett komma och sedan namnet på interfacet:

**Class Teacher: Person , IComparable**

**{**

**Public int CompareTo(person other)** så här ska implementeringen utav CompareTo se ut.

**{**

**}**

**}**

Du kan enbart ärva från en basklass men du kan implementera hur många interfaces du vill.

### Generics

Generics är en funktion i c# som låter dig skriva kod som kan funka i många olika sammanhang. Vad de gör är att de låter dig skriva dina klasser eller metoder utan att definiera en specifik typ. När du sedan anropar metoden eller klassen och skickar in ett värde så definierar du vilken typ som ska användas.

Exempel, vi ska skapa en metod som tar emot två värden och ska returnera en array med de två värden i sig. Säg att vi först vill skapa en array med två int-värden i sig. Alltså en int-array/int[].

Vi måste då bygga en metod som returnerar en int-array och tar emot två int-värden som argument (**argument** = de värden man skickar in i en metod, **parameter** = vad metoden kräver för att anropas).

Det skulle se något ut så här:

**static int[] array(int num1, int num2)**

**{**

**return new int[] { num1, num2 };**

**}**

Sedan så vill vi skapa en array med två strring-värden. Då måste vi bygga en ny metod för att kunna göra det:

**static string[] ArrayString(string a, string b)**

**{**

**return new string[] { a, b};**

**}**

Eller:

**static string[] ArrayString(string a, string b) => new string[] { a, b };**

Vi behöver alltså skriva en ny metod för varje data-typ vi vill använda.

Vi testar att i stället använda generics. För att använda generics så skriver man **”<T>”** efter identifieraren/namnet på metoden/klassen. Sedan ersätter man varje data typ med T. T representerar alltså den typen som värdena har.

**static T[] Array<T>(T a, T b)**

**{**

**return new T[] { a, b};**

**}**

Nu kan vi anropa metoden med vilka värden vi vill och ange vilken typ det är explicit.

(Jag fick det att funka ett par gånger även när jag inte angav vilken typ det var jag skickade in utan bara stopade in ett heltal eller en string).

**Int[] arr1 = Array<int>(2, 4);**

**String[] arr2 = Array<”string”>(”Gustav, ”Höglund”);** Samma metod skapar två array:er utav olika typer.

Man kan också ange att en metod eller klass kan ha flera generiska typer genom att skriva flera mellan <>. PÅ så vis kan man skicka in flera olika typer utav data i samma metod och skilja på dem.

Ex: <T1, T2>

Värt att notera att din generiska typ inte måste heta ”T”. Det är bara vad de brukar heta.

#### Begränsa generiska typer

Om man ändå vill begränsa att T enbart får anta vissa typer så kan man använda nyckelordet ”**where**”. Ett sätt att begränsa vilka typer som det får anta är att implementera ett interface so heter ”**IComparable**” vilket dikterar att T får enbart anta typer som är jämförelsebara. Då kan man göra matematiska beräkningar. EX:

En metod med två generiska parameter och returnerar det större. Implementerar IComparable så att det går.

**Static T Max<T>(T a, T b) where T: IComparable**

**{**

**If ( a < b)**

**Return b**

**If (b < a)**

**Return a**

**}**

#### Generiska klasser

Man kan som tidigare nämnt också applicera detta på klasser.

**Class Något<T>**

**{**

**Public T X;**

**Public T Y;**

**Public Något(T x, T y)**

**{**

**X = x;**

**Y =y;**

**}**

**}**

När man sedan skapar ett objekt per denna klass så måste man vara tydlig med vilken typ som man anger.

**Något something = new Något<int>(2, 4);**

**Något somethingMore = new Något<string>(”Hej”, ”Hej”);**

## Operator Overloading

Operatorer som +,\* och = används ganska flitigt för att hantera olika data. Men när det kommer till mer komplicerade datatyper som objekt så kan det vara lite klurigt att använda dessa operatorer. Om man har två objekt utav samma typ med två fält varav som man vill gämföra i ett if-sats villkor så kan man inte bara skriva: objekt == objetk. Kompilatorn förstår inte vad man menar med detta.

För att komma runt detta så kan man skriva om operatorerna inom en klass när de hanterar en viss typ utav parametrar/data. Detta kallas operator overloading.

För att åstadkomma detta så skriver man något som liknar publika och statiska metoder med nyckelordet **operator**. Det är viktigt att de är publika och statiska, annars funkar de inte. Man kan ändra retur typen också men i fallet med == och != så är det inte rekommenderat. Behåll dem som bool. Det är också värt att veta att vissa operatorer måste överladas parvis. Till exempel == och !=. Dvs om du skapar en variant utav == så måste du göra likadant för !=. Detta gäller också för < och > samt <= och >=.

Exempel:

Vi har en datatyp kallad ”Time” som har två värden; timar (Hour) och minuter (minutes). Vi vill kunna jämföra om två Time objekt är lika, dvs de har lika många timmar och minuter.

**Class Time**

**{**

**Public int Hour {get; set;}**

**Public int Minutes {get; set;}**

**Public Time(inte hour, int minutes)**

**{**

**Hour = hour;**

**Minutes = minutes;**

**}**

Vi har våra variabler, nu behöver vi skapa våra överladdningar:

**Public static bool operator == (Time object1, Time object2)**

**{**

**If (object1.Hour == object2.Hour && object1.Minutes == object2.Minutes)**

**{**

**Return true;**

**}**

**Else**

**{**

**Return false;**

**}**

**}**

Ovan har vi alltså deklarerat att om man använder operatorn == med två Time objekt så kommer deras Hour och Minute fält att jämföras och om de är lika så returneras true. Notera att dett gäller enbart när man använder just två time objekt. Om vi i samma klass skulle använda == för att gämföra två int-värden så skulle det fungera som vanligt. Operator överladdning är alltså typ-känsligt.

Då vi har gjort en operator för == så måste vi också göra en för !=.

**Public static bool operator != (Time object1, Time object2) => object1.Hour != object2.Hour && object1.Minutes != object2.Minutes;**

Vi kanske också vill kunna använda + för att lägga till tid till våra Time objekt.

Vi ska kunna använda ett int värde för att lägga till tid i minuter till våra objekt. Vi skapar en överladdning för + operatorn med en Time parameter och en int parameter

**Public static Time operator +(Time object, int minutes)**

vi ändrar retur typen till Time då vi ska returnera ett nytt Time-värde.

**{**

**Return new Time(Object.Hour, object.Minutes + minutes)**

**}**

Det första att notera här är att vi måste skapa ett nytt Time objekt att returnera. Då vi hanterar objekt som är referens-typer så kan vi inte använda samma objekt som vi skickade in. Om Time var en struct så hade man inte behövt skapa en ny instans då den skulle vara en value type då.

Det andra att notera att detta möjliggör enbart denna typ utav uttryck:

**Int minutes = 22;**

**Time time = new Time(2, 4);**

**time = time + minutes;**

Vi kan inte sätta minuterna som ska adderas först framför objektet. Alltså inte så här:

**time = 22 + time;**

Om vi vill göra så måste vi skapa en ny överladdning där parametrarna äri den ordningen.

**}**

# Collections

Collections är klasser som kan innehålla olika funktionaliteter. Vanligt är olika sätt att sortera, lagra och allmänt hantera data.

Längst upp i din program-fil i VS så kan du se att det står ”**using system**”. Det är här du kan se vilka klasser du använder. Listor och alla de klasser nedan (queue etc) ligger i ett bibliotek kallat ”**System.Collections.Generic**”. Om du vill kunna använda några utav dessa så måste du lägga till detta bibliotek. Alltså:

**Using System.Collections.Generic;**

## <List>

En <list> är likt en array fast lite mer flexibelt. Man behöver inte ange specifikt vilken storlek den har utan den skapar en ny position per element som man placerar i listan. För att skapa en lista så behöver man först se till att man har tillgång till klassen.

Sedan så skapar man dem likt ett objekt.

**List <String> Namn = new List<string>();**

Likt array så behöver man specificera vilken datatyp som listan ska kunna lagra. Man gör det mellan <>.

För att få ett värde för hur stor en lista är så kan man använda ”**.Count**”. Den metoden returnerar ett värde lika med antalet element i listan.

**Int hurMångaNamn = Namn.Count**

För att lägga till ett element i en lista så använder man ”**.add()**”.

**Namn.add(”Gustav”);**

För att stoppa in ett element på en position i en lista så kan man använda ”**.insert()**”. Denna metod tar två argument. Vilket index, vilken position du vill stoppa in ett värde på (Index=position), samt vilket värde du vill stoppa in. Om det redan finns ett värde på det index/position som du vill placera ett värde på så flyttas det (och alla värden efter det) upp ett index.

**Namn.Insert(1, ”Adam”);**

Man kan också ta bort element på ett givet index. Till skillnad från en array så blir det inget tomrum när man gör så utan allt efter det elementet man tog bort hoppar ett steg framåt.

**Namn.RemoveAt(2);** tar bort elementet på index två.

**Namn.Remove(”Gustav”);** notera: tar bort det första elementet i listan med värdet Gustav.

**Namn.RemoveAll(”Gustav”);** tar bort alla element med värdet Gustav i hela listan.

I övrigt så kan man också hantera den som en array.

**Namn[0] = ”Filip”;**

**Console.WriteLine(Namn[0]);**

## Queue

En queue/kö är en samling värden där det enbart går att stoppa in värden längst fram och ta bort dem längst bak. Vi kan alltså inte lägga in ett värde på position 23 och sedan ta bort ett värde på position 12, utan det går bara att fylla på den sista första positionen och ta bort ifrån den sista.

Man lägger till värden med ”**.Enqueue()**”;

Man tar bort värden med ”**.Dequeue();**”

Ex:

Queue <int> kö = new Queue <int>();

**Kö.Enqueue(2);**

**Kö.Enqueue(5);**

**Kö.Enqueue(3);**

**Kö.Dequeue();** kommer att ta bort 2, den första inmatningen.

En kö kan hantera olika data-typer i samma kö. De är också väldigt prestanda effektiva och de kan vara oändligt långa.

## HashSet

Ett ”HashSet” är likt en påse som man bara slänger ned data i. Det finns inga positioner eller någon struktur. Det går inte använda en vanlig for-loop och loopa igenom för varje index då det inte finns några index. Det går dock att använda en foreach-loop.

Det som är bra med en HashSet är att de är extremt prestanda effektiva. Det går alltså snabbt att rota igenom allt och hitta det man letar efter.

En viktig funktion med HashSet är ”**.IntersectWith()**”. Med den metoden så kan man kolla om det finns några lika värden som finns i två olika HashSet. När man använder den metoden så genereras en ny påse som innehåller alla värden som är lika. Motsatsen till .IntersectsWith() är **”**.**ExceptWith()”**

Vi kanske har två påsar (HashSet) och vill kolla om de har några gemensamma värden.

**HashSet<int> num1 = new HashSet<int>(new int[] {2, 3, 4, 1});**

**HashSet<int> num2 = new HashSet<int>(new int[] {4, 7, 1, 19});**

**Num1.IntersectsWith(num2);**

## Dictionary

Ett dictionary sparar två värden per position. Ett utav dessa värden är en nyckel som bestämmer det andra värdets position. Det är alltså likt en lista eller en array men istället för att komma åt ett värde med ett index nummer (en int) så kan man sortera värdena med vilket typ utav värde man vill. Kanske en string eller en double.

Det går inte att placera två olika värden på två identiska nycklar. Om du försöker att placera ett värde med en nyckel vars värde redan existerar så kommer kompilatorn att säga ifrån.

För att skapa ett dictionary/uppslagsverk så gör du så här:

**Dictionary <string, int> ages = new Dictionary <string, int>();**

Tanken med uppslagsverket är att man ska kunna spara åldrar knutna till ett namn.

Jag lägger till några personer:

**Ages.Add(”Gustav”, 24) ;** ”Gustav” är nyckeln och ålder är 24.

**Ages.Add(”Marcus”, 30);**

**Ages[”Fredrik”] = 23;** Kan också göra så här.

Att loopa igenom en dictonary är lite speciellt.

**Foreach (KeyValuePair<string, int> element in ages)**

**{**

**String namn = element.Key;**

**Int age = element.Value;**

**Console.WriteLine($”Mitt namn är {namn} och jag är {age} år gammal.”);**

**}**

# LINQ Language Intergrated Query

Att arbeta med listor och arrayer fyllda med objekt och data är så pass vanligt att de har skapats en egen syntax för att loopa igenom olika arrayer och kunna bearbeta dem på olika sätt. Det kan vara att hämta data med ett vist värde ur en stor samling eller att sortera en stor samling i alfabetisk ordning. Den syntax som har skapats för detta ändamål kallas LINQ.

För att kunna använda LINQ så måste man lägga till det biblioteket likt med generics. Skriv:

**Using System.LINQ**

## Hämta vissa data och sortera data från en array med objekt.

Säg att vi har en array med personer som vanligt. Dessa personer är kunder och vi ska skicka reklam för en bil till alla dessa kunder. Det vore meningslöst att skicka denna reklam till personer som inte har något körkort eller några pengar så vi begränsar oss till de kunder som är över 30.

Arrayen är en array med objekt utav typen kunder som skapas efter klassen kund och det ser ut så här:

**Class kund**

**{**

**Public string Name;**

**Public int Age;**

**Public int Salary;**

**Public string City;**

**Public string Adress;**

**}**

Ingen konstruktor skriven här men tänk dig att den finns där.

Vi har alltså 5 olika data per person som vi har loggade i vår databas. Tänkt nu att vi har en array med massor utav kunder:

**Var kunder = new[]**

**{**

**New Kund(”Nils”, 26, 27000, ”Linköping”, ”Tallbodavägen 20”)**

**New Kund(”Elin”, 34, 37000, ”Örebro”, ”Örebrovägen 80”)**

**New Kund(”Philip”, 40, 50000, ”Motala”, ”Motalavägen 54”)**

**New Kund(”Eric”, 20, 20000, ”Stockholm”, ”Stockholmsvägen”)**

**Etc….**

**}**

Vi vill nu få tag på alla kunder över 30 år. Vi använder LINQ för detta.

När man använder LINQ för att hämta specifika data från en samling så skapar man en ny implicit variabel (var). Sedan använder man nyckelordet **”from”** för att skapa något som fungerar som en foreach loop. Sedan använder man nyckelordet ”**where”** tillsammans med ett villkor för attfiltrera vilka objekt man vill ha. Och till sist så använder man ”**select**” för att välja vad man vill hämta för data.

**Var reklamKunder = from kund in kunder**

**where kund.age >= 30**

**select kund;**

”kund” blir variabel namnet för varje element per iteration i loopen likt i en foreach loop.

När man använder LINQ så returneras ett objekt som implementerar IEnumerable interfacet. Detta innebär att man kan loopa igenom det med en foreach loop. Om man vill så kan man konvertera dem till en lista eller en array med de inbyggda metoderna ”**.ToList**” och ”**.ToArray()**”. Man kan också konvertera dem till dictionary eller hash set etc.

**List<kund> listaMedKunder = Reklamkunder.ToList();** Notera att man måste skapa en ny lista.

Vi kanske inte nöjer oss med att bara ha kunder utav en viss ålder utan vi vill också att de har en lön över 30 000. Vi kan lösa detta genom att implementera flera villkor med &&.

**reklamKunder = from kund in reklamKunder**

**where kund.age >= 30 && salary > 30 000**

**select kund;**

Man måste alltid avsluta en LINQ-sats med en ”select”.

Vi har nu alla våra kunder men vi vill kanske sortera vår lista i alfabetisk ordning. Då använder man nyckelordet ”**OrderBY**”.

**reklamKunder = from kund in reklamKunder**

**OrderBy kund.Name**

**Select kund;**

Om vi vill sortera efter flera variabler så kan vi bara lägga till flera ”OrderBy” uttryck.

Om man vill sortera i omvänd ordning (störst till minst/ö till a) så kan man lägga till nyckelordet ”**decending**” efter orderby uttrycket. Man kan också skriva ”**ascending**” men det är standard så det behövs sällan.

Nu vet vi vilka kunder som är utav korrekt ålder och tjänar tillräckligt med pengar så vill vi skriva ut vad deras namn, stad och adress är. För att göra detta så kan vi använda LINQ för att skapa en ny samling objekt där varje objekt har de tre fälten vi vill ha.

**Var result = from kund in reklamKunder**

**Select new {Name = kund.Name, City = kund.City, Adress = kund.Adress};**

Nu har vi en ny samling objekt med den data vi behöver för att skicka reklam till korrekt personer.

## Join, slå ihop och filtrera data från två olika samlingar.

Om man har mer än en datasamling med mestadels/vissa samma data men någon data som är unik och man vill komma åt data från båda för att skapa en ny samling, eller behandla det på annat sätt, så kan man det med ”**join**”.

Vi kanske har en array kallad ”spouse” där våra kunder partners är loggade. Vi har tre fält där. Namn, ålder och adress. Vi vill kanske veta namnet och åldern på kundens partner också så vi kan finslipa vår reklam ytterligare. Vi gör om ”result” så här:

**var reklamKunder = from kund in kunder**

**join partner in kunder2**

**on kund.Adress equals partner.Adress**

**where kund.Age > 30 && kund.Salary > 30000**

**select new**

**{**

**Name = kund.Name,**

**partnerName = partner.Name,**

**partnerAge = partner.Age,**

**Adress = kund.Adress**

**};**

## Callbacks

Likt i Java script så kan man använda metoder som argument (input) i metoder.

För att göra detta så använder man nyckelordet **Action** som typ i in-parametern i metoden som ska använda en callback.

**Static void CallbackMetod (Action metod, List<int> nums)**

**{**

**Foreach (var element in nums)**

**{**

**Metod();**

**}**

**}**

**Static void HejMetod()**

**{**

**Console.WriteLine(”Hej”)**

**}**

**Main**

**{**

**List<Int> nums = new list<int>();**

**CallbackMetod(HejMetod, nums)**

**}**

Observera att detta funkar enbart om den metoden man använder som callback returnerar void och inte har några in-parametrar. Detta gäller när man använder nyckelordet action.

## Delegates

En delegate är ett objekt som håller en referens till en metod, och det är ett vanligt sätt att skicka en metod som en callback i en annan metod. Skillnaden och fördelen med att använda delegates för att skicka en metod som en callback är att man kan skicka metoder som har en returtyp (alltså skickar tillbaka något) och/eller tar emot några argument. Man kan också lagra referenser till flera olika metoder i en delegat.

Ett annat sätt att använda delegater är att definiera olika data-typer i form utav delegater. Dvs att du kan definiera en viss typ som sedan används som en parameter för en metod. Men denna typ är i sig en annan metod som då används som en callback.

För att skapa en delegat så måste man först göra likt när man skapar klasser som man sedan skapar instanser utav i form av objekt. Så först så gör man en beskrivning som definierar vilken typ metoderna som delegatet refererar till får vara. Dvs vilken returtyp (om någon) samt om de ska ta några in argument. För att skapa en delegat så använder man nyckelordet **delegate**. Man skapar delegat likt fält i en klass.

Exempel nedan visar hur man kan använda delegat som en data-typ so används som parameter i en annan metod.

**Delegate void myDelegate();**

Ovan är en delegat där man kan lagra referenser till metoder som inte returnerar något och inte har några parameter.

**Delegate int myIntDelegate(int i);**

Ovan är en delegat där man kan lagra referenser till metoder som returnerar ett int värde och har en int parameter.

Vi kan nu skappa en metod som har en annan metod som in-parameter. Denna in parameter ska returnera ett int värde och har själv en int-parameter

**Static void Metod(myIntDelegate metod)**

**{**

**For (int i = 0; i < 100; i++;)**

**{**

**Console.WriteLine(metod(i));**

Om vi nu ska anropa ”Metod” så måste vi också ha en metod utav typen ”myIntDelegate” (dvs en metod med int som retur typ samt en int parameter) som argument. Vi kan skapa en och anropa ”Metod ” med den.

**Public static int PlusEttMetod(int i)**

**{**

**Return i++;**

**}**

Nu kan vi anropa ”Metod”.

**Metod(PlusEttMetod)**

Observera att vi inte skriver några parenteser efter ”PlusEttMetod” när vi skickar den som argument.

Vi kan också skapa instanser av delegater och använda dem som referenser till metoder. Om vi skapar en delegat likt tidigare:

**Class MyClass**

**{**

**Delegate void myDelegate();** Detta kan vi använda för att skapa en instans utav en delegat.

**Public myDelegate delegateStuff;** en instans utav typen”myDelegate” kallad ”delegateStuff”.

Vi kan sedan ange metoder som delegatet ska referera till.

**Static void sayHello() {Console.WriteLine(”Hello”);}**

**Main**

**{**

**delegateStuff += sayHello;**

som du ser ovan så kan vi använda operatorn += för att tildela metoder till en delegat. Du kanske tänker att man inte kan använda += för att operara egna typer som objekt och metoder utan att skapa en överladning (operator overload) inom klassen men det har gjorts åt dig och är standarfunktionalitet i VS.

Om vi nu använder oss utav ”delegateStuff” så kommer sayHello att anropas. Om vi vill att delegatet ska referera till fler metoder så är det bara att lägga till fler med +=.

Vi kanske vill ta bort ”sayHello”. För att ta bort sayHello så använder vi -=

**delegateStuff -= sayHello**

# Anonyma metoder

Det går att skapa metoder som gör något men inte har något namn eller objekt med data efter en klass som inte är tidigare skapad.

En anonym metod skapas ofta när man ska anropa en metod som har en action parameter. Dvs en metod som vill ha en metod utan retur typ och inga parameter som argument. I detta fall vill vi dock skicka med en metod med ett argument, så för att komma omkring detta problem så kan man skapa en anonym metod vars enda syfte är att anropa den metoden med ett argument.

**Static void KörMetod(Action metod)**

**{**

**For (int i = 0; i < 20; i++;)**

**{**

**Metod();**

**}**

**}**

**Static void SägHej(int i)**

**{**

**Console.Writeline(”Hej ” + i)**

**}**

Vi vill anropa ”SägHej” i ”KörMetod” men den har en action parameter så vi kan inte anropa den med ”SägHej” direkt:

**KörMetod(SägHej);** Funkar inte då ”SägHej” har en parameter (**int i**).

Vad vi istället får göra är att skapa en anonym metod som anropar säg hej med ett argument.

**KörMetod(()=>SägHej(2));**

Notera ”**()=>**” det är vår anonyma metod och den anropar ”SägHej”.

Anonyma metoder kallas också ”Lambda uttryck”.

# Asynkron kod

Synkron kod = en sak i taget.

Inte optimalt när man text ska hämta data. Om man anväder synkron (vanlig) kod så måste man vänta tills varje fetch är färdig innan man kan fortsätta. Om man har en GUI (graphical user inmterface) kopplad till sin kod så kan den bli oresponsiv om koden till en knapp ska hämta data över internet eller göra någon annan lång process. Lösningen till detta är att implementera asynkron kod.

**Asynkron kod** = kod som gör flera saker samtidigt.

*”****You*** *go to the store and buy pasta.* ***Let me know*** *when you are back.* ***Meanwhile*** *i will prepare the souce.”*

Asynkron kod funkar då moderna processorer i datorer har flera kärnor med flera trådar i varje kärna. När man skapar en bit asynkron kod så ber man kompilatorn att placera denna kod på en separat tråd medan resten utav programmet kör på så långe. Desto fler kärnor en processor har desto fler separata koder kan man köra.

Arbetet att placera och dela upp kod på olika trådar sköts utav ett program i backrunden som kallas WinRT (Windows RunTime) som är en del utav operativsystemet och .net ramverket. Detta innebär att man behöver själv inte anpassa koden för varje typ utav processor (ingen separat kod för två kärniga och fyrkärniga processorer som exempel) utan WinRT delar upp de olika uppgifterna (bitar av asynkron kod) och läger dem i olika köer för varje tråd.

Exempel:

Vi har en stor lista som vi ska loopa igenom och ta reda på om ett visst tal finns i listan. Ett sätt att korta ned tiden det skulle ta att hitta talet vore att dela listan i två och använda två asynkrona kod block för att loopa igenom halva var. Klassen man använder för att göra asynkrona kodblock kallas ”**task**”.

Vår lista med nummer:

**List<int> lista = new List<int>();**

**for (int i = 0; i <= 10000000; i++)**

**{**

**lista.Add(i);**

**}**

För att kunna använda ”task” så måste man använda bilbloteket system.threading.tasks

**Using System.threading.tasks;**

En task/uppgift är ett typ utav objekt så man måste skapa en instans med nyckelordet ”new”.

När man skapar en task så kräver konstruktorn en ”action”, alltså en void metod utan parameter. Då vi ska leta igenom en lista så behöver metoden som uppgiften ska skapas med kunna ta emot en lista att gå igenom, ett tal att leta efter, vart i listan den ska börja leta och vart den ska sluta. Denna metod ser ut så här:

**static void FinnsNummer(List<int> lista, int num, int start, int slut)**

**{**

**for (int i = start; i <= slut; i++)**

**{**

**if (lista[i] == num)**

**{**

**Console.WriteLine("Yes");**

**break;**

**}**

**}**

**}**

Vi har nu vår lista och vår metod. Nu behöver vi skapa våra asynkrona kodbitar. Jag nämnde tidigare att en task måste ha en ”action” som argument. Så vi kan inte bara stoppa in vår ”FinnsNummer” metod så här:

**Task upp1 = new Task(FinnsNummer);** Funkar ej.

För att få det att funka så kan vi använda en anonym metod som anropar ”FinnsNummer” med de parameter vi vill ha som **argument**:

**Task upp1 = new Task(() => FinnsNummer(lista, 80, 0, 49999999));**

Vi letar efter nummret 80. Vi har en task ovan som kollar mellan 0 och 49999999.

**Task upp2 = new Task(() => FinnsNummer(lista, 80, 50000000 , 10000000));**

Och den ovan här kollar mellan 50000000 och 100000000.

För att sedan starta dessa så använder man metoden “**.Start**”.

**Upp1.start();**

**Upp2.start();**

Nu har vi satt igång våra uppgifter och efter detta ska vi öka ett nummer med 1.

**någotNummer++;**

Efter detta nummer så ska vi öka ett annat nummer men det är kritiskt för programmet att våra uppgifter är färdiga innan vi gör det. Om man har en punkt vart efter resultatet utav en asynkron uppgift är kritiskt så kan man använda metoden ”**.wait()**” på task-objektet. Då pausar huvud tråden programmet går på till dess att task-objektet är färdigt.

**Upp1.wait()**

**viktigtNummer++;**

Du kan skapa hur många task/uppgifter som du vill men prestanda är inte obegränsat. Det är som tidigare nämnt en del utav operativsystemet som schemalägger och delar upp dina uppgifter på den tillgängliga hårdvaran. Om du skapar för många uppgifter så kan det resultera i att programmet går långsammare och inte snabbare.

Man kan skapa ett Task objekt och köra det direkt med en rad kod om man använder den statiska metoden **”.Run()”.** Vi har en void metod som ska skriva ut det namnet vi passerar in i metoden och vi vill köra den asynkront.

**Task uppgift = Task.Run(()=>SkrivNamn(”Gustav”));**

Notera att vi inte använder någon metod på instansen utav Task. ”Run” är som sagt statisk.

## Task<>

Vi kanske vill kunna göra en asynkron uppgift som kan returnera ett värde. Då kan vi använda en”**task<>**”. Likt en lista eller ett annat generiskt objekt så skriver vi vilken typ den ska vara i <>.

Vi kanske vill beräkna summan utav alla tal i ett span. Vi behöver först en metod.

**static int SummaAvSpan(int num1, int num2)**

**{**

**int resultat = 0;**

**for (int i = num1; i <= num2; i++)**

**{**

**resultat = resultat + i;**

**}**

**return resultat;**

**}**

Sedan kan vi anropa denna metod med en ”task<int>”

**Task<int> summa = new Task<int>(()=>SummaAvSpan(1, 5));**

**Summa.start();**  startar uppgiften.

**Summa.wait();** väntar tills den är klar.

**Console.writeLine(Summa.result);**

Om man vill använda vad uppgiften returnerar så har en task<> en egenskap som heter ”**.Result**”. När man försöker att använda ”.result” så kommer programmet att vänta tills den uppgiften är färdig så den ”.wait()” som skrevs är inte fult nödvändig.

## Task.ContinueWith()

Om man har flera uppgifter som ska kompleteras i följd och man vill att de alla ska gå på samma tråd så kan man använda metoden ”**.ContinueWith()**” för att skapa ett nytt Task-objekt som körs direkt efter att det tidigare är färdigt på samma tråd. Vi ska skriva ut flera namn i följd och vi gör det med en metod som returnerar void och tar ett string argument. Metoden heter **Namn()**

Startar en uppgift:\_

**Task task = Task.Run(()=>Namn(”Gustav”));**

När den ovan är färdig så vill vi fortsätta på samma tråd så vi vill använda ContiueWith men den metoden är lite krånglig och använda. Den metoden har en parameter som kräver en metod med ett Task-objekt som parameter. Vi måste alltså skicka in en metod med ett task-objekt som parameter. Det enklaste sättet att göra detta på är att göra en anonym metod som ser ut så här:

**(Task task)=>**

Sedan kan man använda den för att anropa den metoden man vill använda.

**Task task1 = task.ContinueWith((Task task)=>Namn(”Marcus”));**

**Task task2 = task.ContinueWith((Task task)=>Namn(”Sigvard”));**

**Task task3 = task.ContinueWith((Task task)=>Namn(”Evy”));**

**Task4.Wait();**

Alla de ovan kommer nu att köras asynkront i följd på samma tråd. Alltså en efter en vid sidan av huvud tråden.

Man kan också använda en Enum kallad ”**TaskContinuationOptions**” som argument in i ContinueWith för att diktera några vilkor huruvida tråden ska fortsätta. En utav dess värden är **OnlyOnRunToCompletion** som säger åt tråden att den ska bara fortsätta om den tidigare uppgiften kan köra klart utan problem. **NotOnFaulted** gör något liknande. Den kör vidare om inga ohanterade fel (exceptions) uppstår. **NotOnCanceled** kör enbart vidare tråden om uppgiften inte avbryts.

Så här skulle den tidigare koden se ut om vi implementerade TaskContinuationOptions:

**Task task1 = task.ContinueWith(**

**(Task task)=>Namn(”Marcus”), TaskContinuationOptions. OnlyOnRunToCompletion);**

**Task task2 = task.ContinueWith(**

**(Task task)=>Namn(”Sigvard”), TaskContinuationOptions.NotOnFaulted);**

**Task task 3 = task.ContinueWith(**

**(Task task)=>Namn(”Evy”), TaskContinuationOption.NotOnCanceled);**

**Task4.Wait();**

Alla utav enum värdena ovan har en motsatts: N**otOnRunToCompletion, OnlyOnFaulted, OnlyOnCanceled**.

Dessa kan vara nyttiga om något händer och vi vill göra något på samma tråd direkt för att fixa det eller försöka på något annat viss.

## PLINQ, Parallel Language Intergrated Query

Om man använder LINQ för att gå igenom stora samlingar så kan det ta lite tid. Om man då vill undvika att applikationen man jobbar på blir seg medan den går igenom en samling så kan man köra LINQ asynkront genom att lägga till metoden ”**.AsParrallel**” på samlingen i LINQ uttrycket.

**Var något = from element in lista.AsParallel() where element >= 20 select element**;

# Parprogrammering

Ett alternativ till att sitta och knappa själv är att sitta två och två. Även om det kan ses so matt det är lite slösaktigt så fins det studier som visar att det blir mindre buggar om man har två ögon på koden när den konstrueras.

Rent praktiskt så kan man likna det med en rallybil. Du har en person som styr och sitter och skriver koden medan den andra personen navigerar och förklarar vad som ska skrivas.