Anteckningar Programmering Grund

# Dag 1, Top-down design och bottom up.

Top-down: Bryt ned uppgiften i mindre delar och arbeta på dem succesivt.

Exempel, göra frukost.

Först skriver vi ned huvuddelarna.

**1: Förbered köket**

**2: Gör frukost**

**3: Ät frukost**

**4: Städa**

Sedan så går man igenom varje punkt i mer detalj.

**1: Förbered Frukost**

* **Gör en översikt över köket**
* **Städa det som behövs. Diska överbliven disk. Torka av köksbänken.**
* **Ta fram ingredienser och tillbehör.**

**2: Gör frukost.**

* **Använd det du tog fram i steg 1 och gör frukost.**

**3: Gör frukost**

**Bottom up:**

Du har små komponenter som du slår ihop för att få ihop ditt project.

Detta är en metod du kan använda när du har löst många problem. Du använder tidigare lösningar och lägger ihop dem på ett nytt vis för att lösa ett annat problem.

# Grundpelare I programmering.

## Data

**Lagras binärt I minnet.**

Ettor och nollor.

**Olika typer**

heltal, decimaltal, text etc.

**Olika strukturer.**

Olika vis att lagra data I minnet.

**Värden.**

2, 1337, “Hello world”, true etc.

**Aritmetik och jämförelse** Enkla matematiska lösningar.

## Variabler

En behållare för ett värde. Något vi behöver komma ihåg. Tex en lösning som vi sedan kan referera till och använda igen.

Variabler har två viktiga egenskaper. Data typ (vilken typ utav information den har) och en identifierare (ett namn som vi använder när vi behöver referera till den).

Variabeln kommer ihåg vart i minnet dess värde/data finns.

## Vilkor

Jämförelser. Kan skicka tillbaka två olika värden, sant eller falskt.

Finns 6 olika jämförelseoperatorer.

**==** Equals/Är lika med.

**!=** Not equal/Är inte like med.

**>** Greater than/Större än.

**<** less then/mindre än.

**>=** greater or equal/större eller lika med.

**<=** lesser or equal to/mindre eller lika med.

## Loopar

Man vill kunna upprepa instruktioner flera gånger. Då använder man loopar.

Man använder sig utav ett villkor för att bestämma hur många gånger man ska loopa en instruktion.

**Så länge X är sant så ska denna instruktion loopa.**

Funktioner

En rad kod som man vill kunna använda/anropa flera gånger. Sparade Instruktioner. De har en input och ett output. Data in och data ut.

## Statements & expressions

En ”statment” talar om för datorn gör det här. En jämförelse är ett ”statement”.

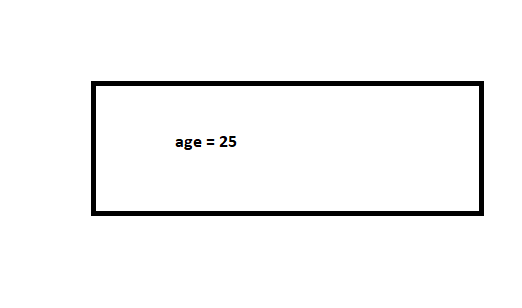
En expression

# Flödesscheman

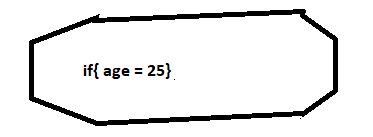
För att visualisera hur ett program ska göras så kan man använda sig utav flödesscheman. Dessa ska skrivas på ett vist sätt. En syntax

**Syntax= hur man ska skriva koden. Kan liknas med grammatik.**

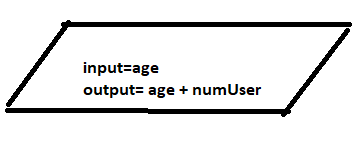
Rektanglar= en ändring i programmets tillstånd. Ändrar värde på en variabels tillstånd.



Loopar och villkor så använder man en sträckt hexagon.



Funktioner skrivs som en lutad rektangel.



# Tilldelning

För att tilldela ett värde ett värde på en variabel så använder man operatorn **=**.

**Age = 25**

När man ger en variabel en identifierare (ett namn) så ska namnet starta med en liten bokstav tex age.

Om det är ett sammansatt namn så ska den första bokstaven i varje ord vara stort.

Ex **numUser = 2**

## Villkor

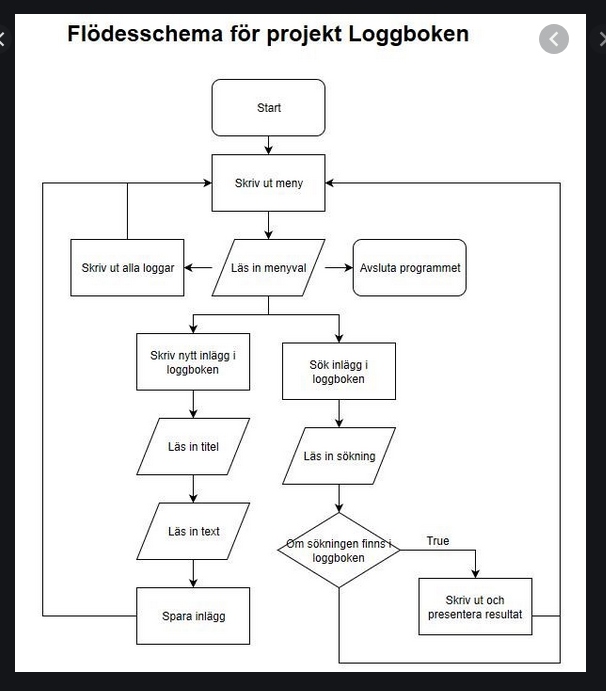
Om man vill göra ett villkor så skriver man en variabel följt utav en jämförelseoperator följt utav nästa variabel.

Ex

**Age <= numUser**

Om man använder **<, >, <=,** eller **>=** så måste variablerna vara en numerisk data typ.

Exempel på ett flödesschema med tilldelning och villkor:



# Iteration

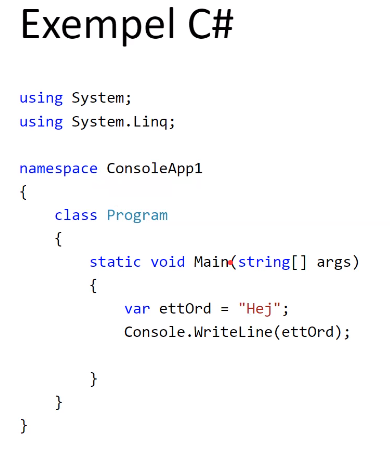
För att rita en iteration så använder du en hexagon med en cirkel framför sig. Sedan så ritar du en linje med de instruktioner som ska i utföras i iterationen. Så länge som villkoret är sant så är man kvar i loopen.

Visa skriver en ”diamant”. Se bilden ovan.

# Syntax & semantik i C#

## Syntax

Hur du ska formulera ett språk. I detta fall hur man ska skriva programmerings kod. I vilken ordning man ska skriva vissa tecken. Vilka nyckelord som är tillåtna. Inom programmering handlar det om innebörden av ett program.



**{} måsvingar, curly braces.** Start och slut på ett kodblock, klasser och namspaces.

**; semikolon,** markerar slutet på en instruktion.

## Keywords

Ord som säger vad en instruktion har för roll eller värde.

**Using =** används för att tala om för kompilatorn att det finns kod någon annanstans som ska användas.

**Class=** en bit kod som du vill kunna anropa och återanvända.

**If =** en selektionssats. Om ett vilkor är sant så gör ”X”. Annars (**else**) gör du ”Y”.

# Programmeringsspråk

Datorns språk består av ettor och nollor. Inte särskilt praktiskt för oss människor.

För att lösa det så har programmeringsspråk utvecklats som är lättare för oss att läsa men kan göras om (kompilera) till ett språk som datorer kan förstå.

## Kompilerade programmeringsspråk

Språk som översätts till maskin kod. Compile = att sammanställa.

Exempel: c#;

## Tolkade programmeringsspråk

Kan köras direkt utan att behöva kompileras.

Exempel på dessa: JavaScript, PHP, Python och Lisp.

## Visual Studio

Är ett IDE, Integrated Development Enviroment

Används för att skriva (bland annat) c# kod.

Kan kompilera koden till program som körs i Windows eller på webben.

# C#

Skapades av Microsoft 2000 och härstammar från språken C och C++. Språket utvecklas kontinuerligt.

Språket är ett objektorienterat.

Det är Statiskt, Starkt och Implicit typat.

Det finns stöd för funktionell programmering.

Kan används för väldigt många olika saker.

## Konventioner.

Regler för hur du ska skriva kod. De har ingen funktionell betydelse utan är enbart till för att koden ska bli lättare att läsa.

Som exempel så ska variabler ha en liten bokstav i början men metoder ska ha börjat med en stor bokstav. Datorn bryr sig inte om detta men det är bra att göra så då det underlättar när man ska läsa sin egen kod eller någon annans.

## Statments & Expression

Ett ”statement” talar om för kompilatorn ”gör det här” och måste avslutas med ett semikolon (;).

Ex **Consol.writeline(”Hej”)**

En ”expression” är ett ”statement” som evalueras till ett värde. Ex 1+1

## Konsolen

Ett program som följer med alla operativsystem.

**Console.WriteLine** Säger till datorn att skriva ett meddelande på konsolen.

**Console.ReadLine** väntar på att användaren ska skriva något i konsolen.

## Variabler i C#

Syntax

Man skriver först datatypen, följt utan identifieraren, följt utav = (tilldelningsoperatorn) och till sist det värde vi vill ge den följt utav semikolon.

**String text = Hej;** En string variabel med namnet text och värdet ”Hej”.

Man kan också enbart deklarera att en variabel finns utan att ge den något värde.

**String text;** Data typ följt utav identifieraren och sedan semikolon.

Vad det gäller identifieraren så får du ge den vilket namn som du vill men det är bra om du ger den ett namn som beskriver dess syfte.

Det finns en mängd olika variabler i C#. Några utav dessa och deras funktion är:

**Int** = ett 32-bitars numeriskt värde. Dvs att det tar upp 32 bytes utav minne då det är ett tal bestående utav 32 ettor och nollor. Kan vara både negativt och positivt. Har ett maxvärde på ca 2,1 miljarder och ett minvärde på ca -2,1 miljarder.

Det vanligaste numeriska värdet.

**Byte** = ett 8-bitars numeriskt värde. Kan ej vara negativt. Kan hålla värden mellan 0 och 255.

**Long** = 64-bitars numeriskt värde. Kan vara positivt eller negativt.

Maxvärde 9,223,372,036,854,775,80. Minvärde: -9,223,372,036,854,775,808

**Short** = 16-bitars numeriskt värde. Kan vara positivt eller negativt. Maxvärde: 32,767

Minvärde: -32,768

**Double** = 64-bitars decimaltal. Kan vara positiv eller negativ.

**Decimal** =128-bitars decimaltal. Kan vara positivt eller negativt.

**Bool**= en variabel med enbart två giltiga värden. Sant eller falskt.

**String** = en textsträng. Input från användaren via **Console.Readline()** är utav denna typ. Man kan skriva siffror här också men de går inte att göra några beräkningar med dem utan de måste konverteras.

Lagras inte direkt i minnet. Det som finns i variabeln är bara en referens.

**DateTime=** en speciel data typ som står för ett vist datum och tid. Likt strängar så sparas det enbart en referens i variabeln.

**Char** = förkortning för ”character”. Enstaka tecken. Skrivs med apostrof.

Alla numeriska värden (int, byte, long, short) har standardvärde 0 om du inte anger något.

Alla decimal numeriska (double) har standardvärde 0.0.

Du kan dock inte göra något med variabler som du inte har tilldelat något utan de måste ha ett givet värde.

## Operatorer

**=**, tilldelningsoperatorn. Tilldelar (ger) något ett värde.

**+**. Additionsoperatorn. Kan antagligen användas för vanlig matte (1 +2) eller för konkatenering, vilket är att lägga ihop strängar.

**-**, subtraktion operator

**/,** divison

**%,** modulo. För att räkna ut resten vid division utan decimal tal.

4%12=4. Det går ej att dividera 4 med 12 utan decimal tal så det blir kvar 4 då ingen division går att göra.

25&13=12. 13 går i 25 en gång och det blir kvar 12.

Man kan använda sig utav modulo för att ta reda på om ett tal är jämnt eller inte. Om man tar numret modulo 2 och det inte blir någon rest.

Ex: ”num”%2=0. ”num” är ett jämnt tal.

**++**, inkrement öka ett värde med 1.

**--**, minskar värde med 1.

**+=**, lägger till ett givet värde till det existerande värdet.

**Int tal = 4;**

**Tal += 4** Värdet utav ”Tal” är nu

## Implicit typat, var.

Man kan låta kompilatorn ta reda på vad för data typ en variabel genom att ange variabeln **var**. Man måste då tillge ett värde på samma rad.

**Var text = ”hej”** kompilatorn avgör nu själv vad det är för data typ baserat på vad som tillges, i detta fal en sträng.

# Typ konvvertering

Man vill ibland konvertera olika data typer till en annan data typ. Säg att en användare ska ange ett tal som ska multipliceras med 2.

Om man använder **Console.Readline** så får vi tillbaka en sträng. För att kunna göra några beräkningar måste vi konvertera det till ett numeriskt värde.

## Implicit typkonvertering.

Görs automatiskt när man konverterar en större data typ till en mindre. Text från int till long. Det går ej att göra tvärt om tex long till int.

**Int i = 10;**

**Long l = i;** så här gör man.

## Explicit konvertering.

Man säger till kompilatorn att man vill göra en konvertering. Då kan man konvertera mindre data typer till större.

**long i = 10;**

**int l = (int) i;** om man anger att det är ok så kan det funka.

Detta funkar enbart mellan numeriska data typer och om det värde man anger faktiskt får plats i den data typen man konverterar till.

## Hjälpklasser

**Convert,** en hjälpklass man kan anropa för att göra konverteringar mellan int och string.

**int i = 10**

**string tal = Convert.ToString(i)**

man kan också göra tvärtom med en annan hjälpklass.

**String i = ”10”**

**Int l = Convert.ToInt32(i)**

för att göra en konvertering till en long, byte eller short så måste man ange hur många bitar man vill konvertera till.

**Long k = Convert.ToInt64(i)** här konverterar jag till en long.

Man kan också konvertera numeriska värden till strängar på ett snabbare vis.

**Int tal = 123.ToString();**

## Kommentarer

För att skriva kommentarer i sin kod så skriver man:

//så här kommenterar jag en rad.

/\*så här kan jag kommentera flera rader

Kommentar

kommentar

\*/

Ctrl + k + c är en genväg man kan använda för att kommentera.

## Whitespace

Kompilatorn ignorerar alltid mellanslag och tabs.

# Dag 2, Villkor och loopar

## Villkor If, else och else if

Ett villkor är en uppsättning instruktioner som ska göras om ett villkor uppföljs.

**Om det är varmt.**

**Gå och bada.**

Man kan också skriva vad som ska hända om villkoret inte stämmer.

**Om det är varmt.**

**Gå och bada.**

**Annars.**

**Stanna hemma.**

**Syntax är:**

**Temp = 20;** en variabel som avgör vårt villkor.

**If (temp >= 20)** om variabeln är större än 20

**{**

**Console.Writeline(”Gå och bada”)** skriv att vi kan gå och bada

**}**

**Else** Annars

**{**

**Console.WriteLine(”Stanna hemma”)** Skriv att vi bör stanna hemma.

**}**

man kan också beskriva flera olika villkor som ska ha olika instruktioner.

**If (temp >= 20)** om variabeln är större än 20

**{**

**Console.Writeline(”Gå och bada”)** skriv att vi kan gå och bada

**}**

**Else if (temp >= 30)** om det första villkoret inte gäller men vi vill ha flera villkor så gör vi så här.

**{**

**Console.WriteLine(”Det är för varmt. Stanna hemma”)**

**}**

**Else**

**{**

**Console.WriteLine(”Stanna hemma”)**

**}**

Om if villkoret stämmer så kommer alla else if inte att göras utav datorn. Om du har flera vilkor och du vill att alla gås igenom så måste du använda flera if-satser följt utav varandra. Då kollas varje villkor.

Säg att vi vill kolla om det rengar också.

**int Temp = 20;**

**bool regn = true;**

**If (temp >= 20)** om variabeln är större än 20

**{**

**Console.Writeline(”Gå och bada”)** skriv att vi kan gå och bada

**}**

**If (regn == true)** vi kollar om det regnar också.

**{**

**Console.Writeline(”Eh vänta, det regnar. Vi stannar hemma”)**

**}**

Om man gör så här så skulle konsolen skriva både vi går och badar och sedan att vi inte går att bada men det är bara ett enkelt exempel.

### Flera jämförelser i ett villkor

Om man vill att flera villkor ska gälla för att en instruktion ska göras så kan man använda operatorerna && och ||

**&&** = och

**Int temp = 20**

**Bool regn = true**

**If (temp >= 20 && regn == false)** om temperaturen är högre än 20 och det inte regnar

**{**

**Console.WriteLine(”Vi går och badar”)** så kan vi gå och bada.

**}**

**Else**

**{**

**Console.WriteLine(”Vi stannar hemma”)** annars stannar vi hemma.

**}**

**||** = eller. Om man har två villkor som ska kollas men det räcker om ett ska vara sant för att instruktionen ska köras så använder man ||.

**If (temp <= 20 || regn== true)** om temperaturen är mindre än 20 eller det regnar.

**{**

**Console.WriteLine(”Vi går inte och badar”)**

**}**

## Switch

Om du har många olika villkor som skulle leda till många olika utfall så kan det bli lite rörigt om du använder enbart if-satser. Då kan du använda en switch. En switch är ett villkor som har olika fal som används beroende på n variabel.

**int tal = 3**

**switch** (tal)

{

**Case 1:** om variabeln ”tal” är 1 så gör vi det här

**Console.WriteLine(Hej)**

**Break;** man måste ha en sådan här break vid slutet ur varje fal. När man kommer till ett break hoppar vi ut ur switch-satsen.

**Case 2:**

**Console.WriteLine(Hej Hej)**

**Break;**

**Case 3:**

**Console.WriteLine(Hej Hej Hej)**

**Break;**

**Default:** om inget utav ovan stämmer så hamnar vi här på ”default”.

**Break;**

}

Switch är något mer prestanda effektivt (tar mindre datorkraft) än if-satser.

## Loopar. While, do while, for och foreach

En loop kan användas för att göra en instruktion flera gånger. Likt med villkor så gör de något så länge en variabel är san.

### While

**Int i = 0;**

**While (i<10)**  Så länge ”i” är mindre än 10

**{**

**Console.WriteLine(i);** så skriver vi ut värdet på ”i”.

**I++;** och ökar ”i” med 1.

**}**

vad denna loop skulle göra är att skriva 0-10.

### For

Om man vet att man behöver göra något ett antal gånger så kan man också använda sig utav en for-loop.

**For (int i = 0; i < 10; i++)**

**{**

**Console.ReadLine(i);**

**}**

Denna loop gör samma sak som while loopen ovan. Det som är inne i parenteserna är vad som styr loopen. I ordning så är de variabeln **int i = 0,** den som vi sätter vårt villkor efter. Villkor **i<10**, så länge detta stämmer så ska vi vara kvar i loopen. Och vår förändring **i++**, vad som kommer att förändras i varje varv i loopen. I detta fall så kommer ”i” att öka med 1 till dess att villkoret inte längre gäller.

Precis som med if-satser så kan man ha flera olika villkor som styr loopar om man använder operatorerna && och ||.

### Do while

Om man har något som ska köras åtminstone en gång även om vi inte ska in i en loop så kan man använda **do while**.

**Bool i = false;**

**Do**

**{**

**Console.WriteLine(”Spagetti”)**

**}**

**While (i == true)**

**{**

**Console.WriteLine(”Spagetti”)**

**}**

Loopen ovan kommer att skriva spagetti åtminstone en gång men då villkoret i while-loopen inte stämmer så kommer den inte göra det några fler gånger.

### Nesting

Att lägga loopar i varandra är möjligt och man måste ibland göra det men man ska försöka undvika det då koden blir väldigt krånglig.

## Constant

Man kan göra variabler konstanter. De går ej att ändra på. Man måste ge dem ett värde när man deklarerar dem.

**Const int tal = 32;** man skriver ”const” framför variabeln.

## Skillnaden mellan || och |.

Om man skriver alternativ 1 (||) och det första villkoret råkar vara sant så kommer datorn inte ens kolla på det andra villkoret då det enbart behöver en för att köra villkoret. Det behövs bara ett utav dem för att det ska vara ok. Om man skriver alternativ 2 (|) så kolar datorn på båda villkor ändå.

## Skillnaden mellan && och &.

Om man skriver alternativt 1 (&&) och det första villkoret råkar vara falskt så kommer det andra villkoret inte kollas. Med alternativ 2 (&) så körs det åtminstone en gång även om det första villkoret är sant.

# Dag 3 2020-11-11

## Klasser

### Objektorienterings-dicipliner

Vi försöker lösa ett problem i verkligheten med kod. Objekt orientering är ett sätt att se och representera denna verklighet med kod.

Två viktiga begrep

**Abstraktion**

Saker till hör olika grupper. Djur är ett IRL exempel på Abstraktion. Det är en stor grupp som det går att sortera in olika saker i. Vissa saker i verkligheten kan klassificeras inom djur.

**Inkapsling**

Att placera olika element i olika klasser är att inkapsla dem.

Våra klasser differerar ett gränssnitt.

**Klass**

Innehåller:

* Konstruktor
* Fält
* Egenskaper
* Metoder

**Metoder**

* Åtkomst-modifierare
* Returtyp
* Metodnamn
* Parameter

Skapa en klass med en metod i.

Utanför program klassen som innehåller main metoden.

För att skapa en ny klass i visual stuidio så kan du höger trycka på en klass i solution explorer, välja ”new” och sedan välja ”class”

Skriv:

**Class Person** Class följt utav namnet på klassen.

**{**

**Public string GetName(string hej)** skappa en metod. **Public=** metoden kan anropas utifrån klassen. **String=** datatypen som metoden ger tillbaka (output) **GetName=** namnet på metoden. Ange vad du vill. Ska starta med stor bokstav. **(string hej)=** input variabel. I detta fal en string variabel med namnet hej.

**{**

**Return ”Gustav”;** med return så skickar du tillbaka ett värde (output)

**}**

}

För att skapa en metod som inte har någon output så skriver man **void** istället för en data typ efter **public** innan data typen**.**

Man skapar metoder och klasser för att gruppera ihop sammanhängande data. Om man skapar alla variabler i samma klass blir det rörigt väldigt snabbt. Det kan även ledda till dålig prestanda.

Om man ska ha en metod i program klassen som ska gå att anropa från main-metoden, så måste man ge den metoden man skapar nyckelordet **static**.

## Egenskaper i metoder, fält.

För att definiera egenskaper i metoder så kan man använda sig utav fält.

Man skriver i en klass:

**Class Person**

**{**

**Private string \_name;** Alla objekt i denna klass får nu egenskaper från detta fält. De är klassvariabler. Alltså variabler som man kan komma åt i hela klassen. **Private** innebär att variabeln inte kan anropas/kommas åt utanför klassen.

**Public Person** metoden

{

}

}

Om ett fält har samma namn som en annan variabel i en metod så kan man specificera att det är fältet man vill ha genom att skriva **this.** Innan identifieraren (namnet) på variabeln.

**This.name = name**

Vanligtvis så ska man namnge alla fält med ett \_ så slipper man använda **.this**.

Så här: **\_namn**

Ibland så vill man kunna ändra på fält (klassvariabler). Då kan man använda metoder utan output (**void**) för att ändra på fältet.

**Class person**

{

**Private int \_age;**

**Public void SetAge (int newAge)**

**{**

**\_age = newAge**

**}**

När du skapar fält så är det bra att skapa metoder som sätter (set) och hämtar (get) dem.

Komplet exempel.

**Class Person**

**{**

**Private int \_age;** fältet

**Public int person(int ageIn)** metoden

**{**

**Get**

**{**

**Return \_age;**

**}**

**Set**

**{**

**\_age = ageIn**

**}**

## Properties

Ett enklare sätt att använda fält är att använda **properties.**

Syntaxen för det är:

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

Detta gör exakt det som jag skrev ovan. Skapar ett fält och en metod som skickar dess värde och en som ändrar dess värde.

## Samlingar och Loopar

**Array/vektor**

En samling utav värden som placeras under samma variabler. Man kan likna det med att skapa många variabler med samma namn som sedan kan åtskiljas med en siffra som representerar deras position i en lista.

En array(engelska)/vektor(svenska) har ett bestämt antal positioner (antal variabler som kan sparas i den) och det bestäms utav dig. För att skapa en vektor så gör man så här:

**Int[] vektor = new int[7]** data typ [] identifierare tilldelningsoperator ”new” datatyp [hur många positioner.]

Vektorn ovan har alltså 7 platser men datorn numrerar dem inte till 1-7 utan de är numrerade 0-6. Ett värdes position kallas des index.

För att lägga in något i en vektor så skriver man så här:

**Vektor[2] = 10;** här placerar jag värdet 10 på index 2, vilket är position 3.

Tänk att position = index + 1. Index = position -1

Man kan också lägga in värden direkt när man skapar vektorn.

**Int[] vektor = new in[10] {1,2,3,4,5,6,7,8,9,10}** en vektor med 10 platser med värdena 1-10.

Om jag vill skriv ut ett värde så anger jag dess index.

**Console.writeLine(vektor[2]);**

**3**

Den skriver 3 då index 2 = 3. Index = position -1.

Värden som ligger i en vektor kallas element.

För att ta reda på hur många element en vektor innehåller så kan man använda sig utav klassen **.lenght**

**Console.WriteLine(vektor.lenght);**  consolen skulle nu skriva 10.

Om man vill gå igenom alla element i en vektor så kan man använda sig utav en **for.**

**Int[] siffror = new int[10]**

**For (int i = 0; i<siffror.Lenght; i++)** så länge ”i” är mindre än vektorn ”siffror” positioner så är vi i loopen. Alltså så kör vi ett varv för varje position i loopen.

**{**

**Siffror[i] = i +1;** Vi anger värdena 1-10 på alla 10 positioner i vektorn.

**}**

**List**

Vektorer kan vara problematiska då man inte vet i förväg hur många positioner man behöver. Då kan listor vara bättra att använda. De skapar en ny position per värde/element som du lägger in. En list är en egen klass som man kommer åt genom att först ange:

**Using.system.collections.Generic**

För att sedan skapa en list så använder man denna syntax:

**List <int> siffror = new List <int>();** List <datatyp> identifierare

Precis som med vektorer så kallas elementens position i listan index.

**.lenght** funkar inte på listor utan man behöver använda ”**.count”**

För att sedan lägga in värden i listan så använder man metoden ”**.add”**

**Siffror.add(1); .** lägger in värdet 1 på ett nytt index.

För att lägga till ett värde på ett specifikt index så använder man klassen ”**.insert”**

**Siffror.insert(i;tal)** i = index (vart vi lägger in värdet)

Värt att notera med ”**.insert”** är att det inte tar bort det värdet som fans på det index tidigare. Den flyttar istället alla index efter den du placerade ett värde ett index högre.

För att ersätta ett värde på ett index gör du så här:

**Siffror [2] = 2;** index 2 får elementet (värdet) 2.

## For each

Om man har en samling eller en lista och man vill göra något med varje element så kan man använda for each loopen. Den är specifikt gjord för att fungera med samlingar och listor.

For each har inget villkor utan den går helt enkelt igenom hela samlingen.

**Int[] siffror = new int [10]**

**Foreach (var siffra in siffror)**

**{**

**Console.WriteLine(”siffror”);** denna foreach-loop skulle skriva alla element i vektoren.

**}**