7 GRUNDERNA I PROGRAMMERING

Detta kapitel är bokens största kapitel och kanske det viktigaste. Vi kommer här att gå igenom grunderna för sekventiell programmering. Det vi går igenom kommer att ge dig en viktig grundinsikt i hur man kodar i C#.

Faktum är att det du efter att ha läst detta kapitel, kan skapa program som kan göra vilka matematiska beräkningar som helst. Med andra ord kan man säga att det du lär dig i detta kapitel, är vad som behövs för att ett programmeringsspråk ska vara turingkomplett (se kapitel 5.3)!

Det låter kanske konstigt, men i princip alla efterföljande kapitel (metoder och objektorientering, t.ex.) är till för att göra livet enklare för programmeraren.

Det vi i huvudsak kommer att gå igenom i detta kapitel är:

- variabler
- villkorssatser
- loopar

Vi kommer först att gå igenom objekt och variabler, som används för att lagra data i minnet. Därefter kommer vi att gå igenom villkorssatser, de används för att ge olika utfall, t.ex. beroende på vad användaren har matat in. Därefter går vi igenom loopar, ett sätt att köra samma kodstycke om och om igen.

Vidare går vi igenom något som kallas operatorer och tittar på hur kodblock fungerar i C#. Var lugn, allt kommer att förklaras i detalj!

Kursmål och mål i ämnesplanen som helt eller delvis behandlas i detta kapitel

Kursmål

- Programmeringsspråkets eller -språkens grundläggande datatyper samt fördefinierade strukturer, regler och syntax.
- Kontrollstrukturer, till exempel sekvens, selektion och iteration [...]
- Variablers och konstanters synlighet och livslängd.

Mål i ämnesplanen

- Förståelse av och färdigheter i att använda datalogiska begrepp och algoritmer.

7.1 Objekt och variabler

När man programmerar i C# är det i princip omöjligt att inte använda sig av något som kallas objekt eller variabler. Objekt och variabler används som sagt för att lagra data i minnet.

I tidigare programmeringsspråk gjorde man skillnad på variabler och objekt. Variabler var enklare och användes mest som behållare för t.ex. tal. Objekt hade funktionalitet och kunde hantera många variabler på en gång. I C# är det dock ingen skillnad på objekt och variabler, rent tekniskt sett. De skapas båda av klasser och innehåller viss funktionalitet. I denna bok kommer vi att använda termen variabel för så kallade enkla datatyper (se kapitel 7.1.5 för lista över enkla datatyper/variabeltyper). När vi talar om objekt syftas både enklare datatyper men också andra klassobjekt.

Om du inte förstår detta nu är det inte så konstigt. Vi kommer att lära oss mer om variabler, objekt och klasser längre fram. Nu släpper vi begreppet objekt ett litet tag och fokuserar på de enkla datatyperna, variablerna.

7.1.1 Vad en variabel är för något

Du känner kanske redan till begreppet variabler? Det återkommer bland annat i matematiken. Då talar man ofta om okända variabler som X. Variabler i C# används lite annorlunda.

En variabel ses kanske enklast som en låda, med en etikett på. På etiketten står det ett namn och i lådan ligger det något.

Figur 7.1 – Variabler kan ses som lådor med etikett och innehåll.

I C# är det också så att alla lådor är av olika typer. Man lägger t.ex. ner bananer i bananlådor och äpplen i äppellådor. Att försöka lägga en banan i en låda för äpplen går inte. Kompilatorn skulle säga ifrån.

I C# finns det många olika typer av variabler. I denna bok kommer vi främst att arbeta med heltal och textsträngar, men det finns också variabler som hanterar bilder och de som håller reda på vilken knapp på spelkonsolen som nyss blev nedtryckt. De mer komplexa variablerna kallas alltså objekt. Vi kommer att gå djupare in på objekt i kapitel 13.

7.1.2 Att deklarera (skapa) en variabel

När man deklarerar (skapar) en variabel anger man först vilken typ variabeln ska vara av, därefter namnet. Om man t.ex. vill göra en heltalsvariabel med namnet `nr` så skriver man:

Exempel 7.1

```
int nr;
```

Ordet `int` är en förkortning av integer (engelska för ordet heltal).

Det går också att skapa flera variabler samtidigt:

Exempel 7.2

```
int nr1, nr2, nr3;
```

Vad vi gör är att vi skapar utrymme i datorns minne för att få lagra 32 bitars heltal (se Bilaga 2 – Det binära talsystemet).

7.1.3 Tilldelningsoperatorn =

Nyskapade variabler innehåller inget. Innan vi kan använda dem måste vi tilldela dem ett värde. Det gör vi med hjälp av tilldelningsoperatorn ”=”:

Exempel 7.3

```
nr = 100;
```

Inom matematiken har du kanske lärt dig att det inte spelar någon roll på vilken sida ”lika med”-tecknet olika tal står. Tilldelningsoperatorn i C# fungerar dock lite annorlunda. Variabeln som ska få ett värde, står till vänster om tilldelningsoperatorn. Värdet som variabeln ska få, står till höger.

Detta är alltså inte okej:

Exempel 7.4

```
100 = nr; // ej ok!
```

Om man vill kan man direkt ge variabeln ett värde när man skapar den:

Exempel 7.5

```
int nr = 100;
```

Det som ligger på högersidan om ”lika med”-tecknet händer först. Det innebär att vi kan göra beräkningar på där. När det väl har räknats ut, så tilldelas variabeln värdet. Vi kan t.ex. plussa ihop en massa siffror och sedan lägga det i variabeln.

Här får variabeln nr värdet 110:

Exempel 7.6

```
int nr = 100 + 3 + 7;
```

Med andra ord sker det alltså i följande tre steg:

1. Variabeln nr skapas
2. Talen 100, 3 och 7 summeras till 110.
3. Variabeln nr tilldelas värdet 110.

Man kan öka en heltalsvariabels värde. Det gör man genom att lägga på variabelns (tidigare) värde till sig själv och addera ett nytt värde. På första raden i följande kodstycke får variabeln nr värdet 100, på andra raden får den värdet 150:

Exempel 7.7

```
int nr = 100; // nr har värdet 100
nr = nr + 50; // nr har värdet 150!
```

Man kan också plussa på två variabler med varandra. Här tilldelar vi variabeln sum det sammanlagda värdet av nr1 och nr2, alltså 655:

Exempel 7.8

```
int nr1 = 100;
int nr2 = 555;
int sum = nr1 + nr2;
```

Tecknet ”+” kategoriseras som en så kallad räkneoperator. Det går bra att använda andra räkneoperatorer också, såsom minus, gånger och delat. Se kapitel 7.2.3 för en lista över alla räkneoperatorer som finns i C#.

Om man vill skapa och tilldela flera variabler på en gång skriver man såhär:

Exempel 7.9

```
int nr1 = 100, nr2 = 179, nr3 = 348757;
```

7.3.1 Jämförelseoperatorn lika med ==

Det finns som sagt flera jämförelseoperatorer i C#, vi börjar med att gå igenom *lika med*. Den betecknas med *två* lika med-tecken, alltså ==. Den används för att jämföra om något är (exakt) lika med något annat. Det finns andra jämförelseoperatorer för att t.ex. kontrollera om något är större än och mindre än.

Vi har hittills bland annat lärt oss tilldelningsoperatorn =. I C# skiljer man på jämförelseoperatorn == och tilldelningsoperatorn =. Tilldelningsoperatorn ger en variabel ett värde. Jämförelseoperatorn jämför som sagt två värden med varandra. Man får under inga omständigheter blanda ihop dem med varandra. Gör man det kan man få mycket lustiga resultat, och det kan vara svårt att hitta felen!

7.3.2 if

Låt oss skrida till verket! En `if`-sats jämför alltså två värden med varandra. Vi kan skriva det på svenska. Att skriva något på svenska (eller engelska) brukar kallas pseudokod, man översätter det sedan till programmeringsspråk. Vi kommer att gå igenom detta mer i kapitel 8.1. Så, en `if`-sats på svenska blir alltså en om-sats:

OM något SÅ
gör detta.

T.ex. så kan man kontrollera hur varmt vatten är:

OM temperatur är 100 SÅ
skriv ut ”Nu kokar vattnet!” på skärmen.

Vi har nu skrivit en `if`-sats i svensk pseudokod. Läs mer om pseudokod i kapitel 8.1. Låt oss prova detta med exempel i C#:

Exempel 7.26

```
Console.Write("Ange temperatur: ");
string str = Console.ReadLine();
int temperature = Convert.ToInt32(str);
if (temperature == 100)
{
    Console.WriteLine("Nu kokar vattnet!");
}
```

Observera att det vi vill kontrollera ligger inom parenteser efter ordet `if`. Lägg också märke till att det inte är något semikolon `;` efter `if`-satsen, istället finns där två klammerparenteser `{` och `}`. I kodblocket (se kapitel 7.6) mellan klammerparenteserna ligger den kod som vi vill utföra, ifall villkorssatsen visar sig stämma.

Om vi anger att vattnet är 100 grader, så får vi alltså följande resultat i konsolfönstret:

```
Ange temperatur: 100
Nu kokar vattnet!
```

Vi kan såklart arbeta med textsträngar också. Här kontrollerar vad vatten är:

Exempel 7.27

```
Console.Write("Ange, vad är vatten: ");
string water = Console.ReadLine();
if (water == "blött")
{
    Console.WriteLine("Ja, vatten är blött!");
}
```

Vi får följande resultat:

```
Ange, vad är vatten: blött
Ja, vatten är blött!
```

7.3.3 else

Att använda en `if`-sats utan något mer, gör att vi kör en stycke kod (det mellan hakparenteserna) om villkorssatsen visar sig stämma. Annars gör vi ingenting speciellt utan programmet fortsätter bara att köra. Låt oss fortsätta med temperaturer. Om vi i Exempel 7.26 angav något annat än 100, så slutar programmet abrupt:

```
Ange temperatur: 89
```

Men ofta vill man ju faktiskt göra någonting annat, om det visar sig att villkorssatsen inte stämmer. Låt oss fortsätta med det kokande vattnet, först i pseudokod. Vi lägger nu till ett slut på OM-satsen, för att förtydliga vad som ligger inom den och inte:

OM temperatur är 100 SÅ
* Skriv ut ”Nu kokar vattnet!” på skärmen.*
ANNARS SÅ
* Skriv ut ”Vattnet är inte exakt 100 grader...”*
SLUT OM

Låt oss programmera detta i C#:

Exempel 7.28

```
Console.Write("Ange temperatur: ");
string str = Console.ReadLine();
int temperature = Convert.ToInt32(str);
if (temperature == 100)
{
    Console.WriteLine("Nu kokar vattnet!");
}
else
{
    Console.WriteLine("Vattnet är inte exakt 100 grader...");
}
```

Nu får vi i alla fall ett meddelande, om vi anger att temperaturen är annat än 100 grader:

```
Ange temperatur: 89
Vattnet är inte exakt 100 grader...
```

7.3.4 else if

Ibland så vill man ju ha möjlighet att göra fler alternativ, beroende på vilket värde något har. Låt oss göra en test som säger till när man är exakt halvvägs till 100. Vi gör det först i pseudokod:

OM temperatur är 100 SÅ
  Skriv ut ”Nu kokar vattnet!” på skärmen.
ANNARS OM temperatur är 50 SÅ
  Skriv ut ”Halvvägs till 100!”
ANNARS SÅ
  Skriv ut ”Vattnet är inte exakt 100 grader...”
SLUT OM

Observera att vi lägger in ANNARS OM mellan den första OM och den sista ANNARS. Vi hanterar alltså först alla specifika resultat (50 och 100). Sist tar vi hand om alla övriga resultat (alltså alla övriga temperaturer).

Kodat i C# blir det:

Exempel 7.29

```
Console.Write("Ange temperatur: ");
string str = Console.ReadLine();
int temperature = Convert.ToInt32(str);
if (temperature == 100)
{
    Console.WriteLine("Nu kokar vattnet!");
}
else if (temperature == 50)
{
    Console.WriteLine("Halvvägs till 100!");
}
else
{
    Console.WriteLine("Vattnet är inte exakt 100 grader...");
}
```

Man kan fylla på med hur många `else if` man vill för att kontrollera andra temperaturer. Det går också att använda sig av `else if` utan att använda sig av en `else` på slutet.

7.3.5 If-satser med mindre än-operatorn <

Vi kommer nu att gå igenom mindre än-operatorn. Precis som i matematiken betecknas den av tecknet <. Vi kan använda den för att kolla alla temperaturer under hundra grader. Ett exempel i pseudokod:

OM temperatur är mindre än 100 SÅ
Skriv ut ”Vattnet är inte tillräckligt varmt än...” på skärmen.
ANNARS SÅ
Skriv ut ”Vattnet kokar!”

Kodat i C# blir det:

Exempel 7.30

```
Console.Write("Ange temperatur: ");
string str = Console.ReadLine();
int temperature= Convert.ToInt32(str);
if (temperature < 100)
{
    Console.WriteLine("Vattnet är inte tillräckligt varmt än...");
}
else
{
    Console.WriteLine("Vattnet kokar!");
}
```

På samma sätt som med operatorerna == och <, så kan du använda `if`-satser med de övriga jämförelseoperatorerna som finns listade i kapitel 7.2.1.

7.3.6 If-satser med och-operatorn &&

Ibland vill man kontrollera om två eller flera saker stämmer, samtidigt. Då kan man använda och-operatorn. Låt oss göra ett litet program där vi ser om det går att bada. Vi vill kontrollera så att det finns vatten och att det är varmt nog:

(Se nästa sida)

11 METODER

I det här kapitlet ska vi gå igenom något som i C# kallas för metoder. Metoder är något som finns i många programmeringsspråk och har många olika namn. Ett vanligt namn är funktioner, ett annat är algoritmer. Ur C#:s perspektiv är det dock skillnad på dessa. Funktioner finns egentligen inte, då funktioner är metoder som ligger utanför klasser (och allt ligger ju i klasser i C#). I t.ex. C++ går det däremot att hitta funktioner. Algoritmer å andra sidan är en lösare term och måste inte vara något som ligger indelat i en metod – utan kan mycket väl vara en del av en metod eller flera metoder ihop.

Kursmål och mål i ämnesplanen som helt eller delvis behandlas i detta kapitel

Kursmål

- Programmeringsspråkets eller -språkens grundläggande datatyper samt fördefinierade strukturer, regler och syntax.
- Grunderna för [...] metoder.

Mål i ämnesplanen

- Förståelse av och färdigheter i att använda datalogiska begrepp och algoritmer.

11.1 Vad en metod är för något

Metoder är till för att strukturera kod och förenkla livet för programmerare. I teorin vore det möjligt att programmera genom att lägga all kod i `Main()`-metoden, men när man gör större program blir koden snart oöverskådlig. Med metoder klumpar man ihop kodstycken som hör ihop och lägger dem för sig.

När man arbetar i större projekt är det vanligt att en olika projektdeltagare får ansvar för att konstruera olika metoder. Man bestämmer i förväg vad metoden ska göra och hur den ska användas (in- och utdata, som vi snart ska gå igenom) men man går inte in på hur själva metoden ska programmeras. Detta är sedan upp till den enskilda programmeraren att lösa.

Det är vanligt att man använder en metod flera gånger om i ett program. Metoder gör att vi bara behöver skriva koden en gång – sedan kan vi anropa metoden (och därmed koden i metoden) hur många gånger vi vill. Vi behöver alltså inte köra copy & paste överallt med det kodstycke vi vill upprepa. När vi väl vill ändra något i kodstycket behöver vi bara ändra på ett ställe.

11.1.1 Exempel på ett anrop av en metod

Tänk dig följande exempel:

Skriv ut ”Ange temperatur i Celsius”
Mata in temperatur
Konvertera temperaturen till Fahrenheit
Skriv ut Fahrenheit-temperaturen

Detta exempel känns ganska enkelt, det är för att vi har lagt Celsius till Fahrenheit-konverteraren i en egen metod. När vi väl anropar metoden så behöver vi alltså inte veta hur den är konstruerad. Det kan vara hur krånglig matematik som helst.

Vi har redan använt oss av flera metoder i den här boken och har inte behövt bry oss om hur de funkar. T.ex. har vi använt oss av `Console.WriteLine()`. Vi skickat in text som kommer ut på skärmen utan att vi behöver förstå hur.

11.1.2 När ska vi skapa en metod och hur ska den kodas på bästa sätt?

Det är inte helt självklart när vi ska skapa en metod eller inte. Det finns dock några tumregler. En metod ska helst göra endast en sak. Om fler saker behövs göras skapar man hellre fler metoder och gör de olika delarna separat. Detta gör att en metod ofta är ganska kort – vilket förenklar felsökande då något gått snett.

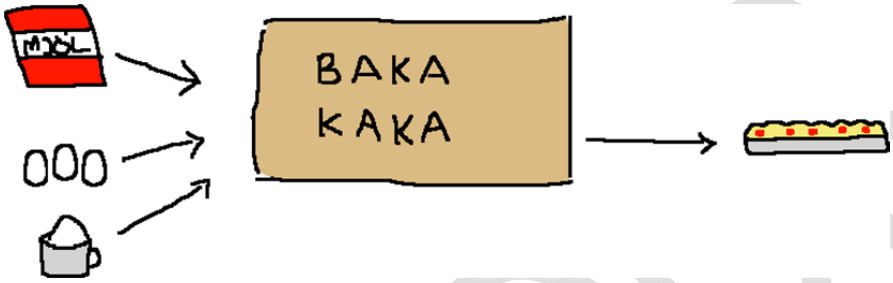
Det ska framgå av namnet vad metoden gör. Ibland är inte det fullt ut möjligt och därför är det bra att alltid skriva en kommentar ovanför metoden, som kort berättar vad den gör.

En metod ska heller inte vara direkt beroende av andra metoder. Ändrar man i en metod ska man inte behöva ändra något i en annan.

11.2 Indata och utdata

Man kan se en metod som en hemlig mojäng. I ena änden stoppar man in några saker och i andra ändan får vi ut något annat. Exakt vad mojängen gör med sakerna och hur – behöver vi inte bry oss om. I exemplet ovan skickade vi in grader i Celsius och fick ut grader i Fahrenheit.

Vi har tidigare jämfört programmering med kakrecept. Låt oss fortsätta med den jämförelsen då vi ska förstå metoder. Man skickar in en mjöl, mjölk, ägg och socker i ena änden av lådan och ut i andra änden kommer en kaka!



Figur 11.1 – Metoden BAKA KAKA.

Detta är möjligen ett något löjligt exempel på en metod. För övrigt saknas nog mjölk eller vatten för att det skulle bli en riktigt bra kaka. Men så är det med exempel, de är inte alltid helt verklighetstrogna.

Vi kommer strax att börja koda metoder i C#, men först ska vi gå igenom några viktiga saker vad gäller just in- och utdata.

11.2.1 Det som skickas in och ut är objekt (variabler)

När vi talar om in- och utdata så är det objekt (eller variabler) vi talar om. I `Console.WriteLine()` så skickar vi ju som bekant in en variabel av typen `string`. Men vi får inte ut någon variabel. Även om vi får ut text på skärmen, så har `Console.WriteLine()` alltså ingen utdata. Däremot har metoden `Convert.ToInt32()` både in- och utdata:

Exempel 11.1

```
Console.Write("skriv ett nummer:");
string strNr = Console.ReadLine();
int nr = Convert.ToInt32(strNr);
```

I detta exempel så skickar vi in en `string`-variabel, `strNr` till `Convert.ToInt32()` och får ut en `int`-variabel `nr`.

Vi kan illustrera detta på samma sätt som exemplet BAKA KAKA. Vi tänker oss att användaren har matat in ”42” i konsolfönstret och har det därmed lagrat i `strNr`:

Figur 11.2 – Convert.ToInt32

Vi har också arbetat med metoden `Console.ReadLine()` som inte har någon indata men utdata. Det är också möjligt att göra metoder som har varken in- eller utdata.

11.2.2 Fyra typer av metoder, gällande in- och utdata

En metod kan alltså vara av följande fyra typer, gällande in- och utdata:

- Metod med in- och utdata.
- Metod endast med indata.
- Metod endast med utdata.
- Metod med varken in eller utdata.

11.2.3 Regler för in- och utdata

Följande regler gäller för variabler i in- och utdata:

- En metod kan ha så många indata som man vill, men endast en utdata. (Om man vill ha fler än en utdata kan man lösa det med till exempel referenser, vilket vi går igenom i delkapitel 11.8.)
- Indata och utdata kan vara av samma typer som variabler. De kan med andra ord vara av typerna `bool`, `string`, `int` osv. De kan också vara objekt (eller variabler) av klasser man har skapat själv eller som någon annan har skapat.

11.2.4 Argument (parameter) och argumentlista

Ett annat ord för indata är argument. Ett argument är helt enkelt *ett objekt som skickas in i en metod.* När man har flera argument som man skickar in i en metod, heter det argumentlista.

Ett vanligt namn på argument är parameter. Dessa två namn syftar på samma sak. I denna bok används främst termen argument.

15.8 Exempel på program – en kalkylator

För att enklare förstå hur allt hänger ihop i ett Windowsprogram, låt oss steg för steg gå igenom hur man skapar en enkel kalkylator. Kalkylatorn ska innehålla följande:

- En textruta som visar talen
- Knapparna 0-9
- En plusknapp
- En lika med-knapp (=)
- En rensa-knapp (C)
- Ett välkomstmeddelande när programmet startas
- En meny för att avsluta och få hjälp

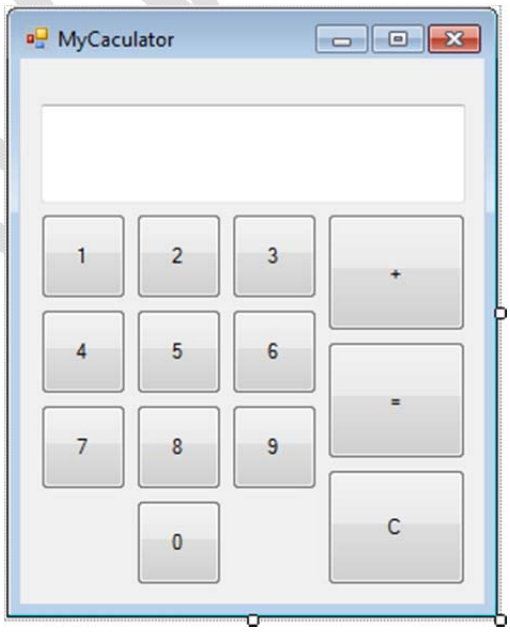
Steg 1 – Skapa layouten

Skapa ett nytt projekt, du kan kalla det vad du vill.

Vi börjar med att lägga till 12 knappar och en textruta till vår form. Vi ändrar storlek på knapparna och placerar dem snyggt. Placera knapparna så att `button1` ligger där knappen med texten ”1” ska ligga, osv.

Via egenskaper sätter vi vettig text på knappar och på formen. Vi lämnar lite utrymme längst upp för en meny.

När vi är klara bör det se ut något i stil med detta:



Figur 15.10 – Design av kalkylator

Steg 2 – Att ändra namn på kontrollerna

Nu heter knapparna `button1`, `button2` osv. Detta bör stämma med knapparna för talen 1-9. Däremot knappen med texten ”0” heter knappast `button0`. Därför döper vi om kontrollen för knapp ”0”, så att dess egenskap `name` blir `button0`.

Vi byter också namn på +-knappen till `buttonAdd`, =-knappen till `buttonSummarize` och C-knappen (rensaknappen) till `buttonClear`.

`texBox1` kan vi låta heta som den gör, likaså `Form1` (`Form1` borde kanske ha hetat `form1`, men det är Microsoft som har bestämt att den ska ha ett stort F, så låt gå…).

Steg 3 – Att koda funktionalitet för knapparna

Detta steg är det längsta steget i denna guide. Vi upp den i flera steg.

Steg 3.1 – Test att klistra in text i textBox1 via en knapp

Som programmerare är det alltid bra att dela upp problemet i mindre problem. Låt oss nu börja med ett litet enkelt test.

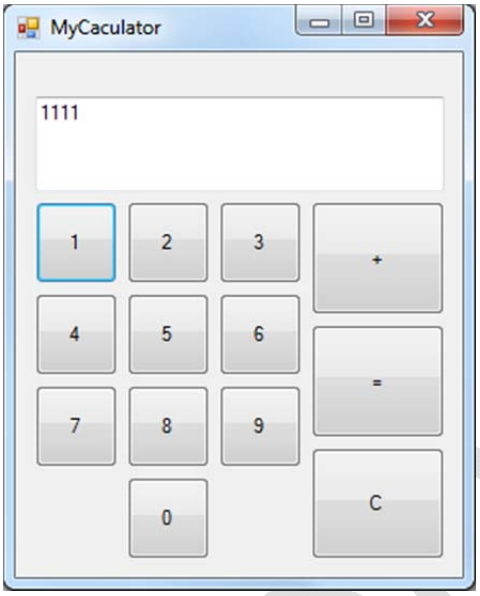
När man klickar på en siffra, så vill vi ju att talet ska hamna i textrutan. Vi dubbelklickar därför på `button1` och lägger till följande kod:

Exempel 15.1

```
textBox1.AppendText("1");
```

`Appentext()` är en metod i klassen `TextBox` som helt enkelt lägger till en textsträng efter den nuvarande texten.

Om vi nu startar programmet och trycker på ett bör vi få fyra 1:or i textrutan:



Figur 15.11 – Metoden `Appendtext()` lägger till texsträngen ”1” i textrutan fyra gånger

Okej! Det var inte så svårt. Nu kan vi jobba vidare!

Steg 3.2 – Att hålla reda på talen som räknas ut och som användaren skriver in

Okej. Vi vet nu hur man lägger till text i textrutan. För att summeringsfunktionen på plusknappen ska fungera, så behöver vi hålla oss till två tal. Dels det tal som skrevs in tidigare (eller talet 0, om inget tal skrivits in tidigare. Dels det tal som användaren skriver in just nu

För att hålla reda på detta skapar vi två variabler:

- En `int`-variabel, `sum` som håller reda på summan. Detta tal börjar på 0, men kommer sen innehålla summan av alla de tal som användaren har plussat ihop (till dess att användaren trycker på rensaknappen C).
- En `string`-variabel, `newNr` som innehåller det tal som användaren matar in just nu. Denna ska inte vara ett heltal, för precis som med textrutan vill vi *lägga till* siffror i den, *inte* plussa på. Om användaren t.ex. trycker 1 och sedan 3 så vill den ju ha talet 13 inte talet 4.

Vi skapar dessa variabler längst upp i klassen `Form1` och sätter deras initialvärden till 0 respektive ””:

Exempel 15.2

```
int sum = 0; // Håller reda på summan för uträknade tal
string newNr = ""; // Innehåller talet som användaren skriver just
nu
```

När man trycket på någon av knapparna 0-9, så vill vi dels att talet ska läggas till i `texBox1` och dels att det ska läggas till i `newNr`. Koden för `button1` skulle då bli:

Exempel 15.3

```
textBox1.AppendText("1");
newNr += "1";
```

Men eftersom vi ska göra detta 10 ggr, (för knapp 0, 1, 2 osv) så kan vi istället definiera en metod för det. Vi döper metoden till `AddDigit()` och låter det ta en `string`-variabel som indata. Ordet ”digit” betyder siffra på svenska. Tanken är att metoden ska lägga till en siffra till det tal användaren just nu matar in:

Exempel 15.4

```
private void AddDigit(string nr)
{
    textBox1.AppendText(nr); // Lägg till numret i textboxen
    newNr += nr; // Lägg till numret i talet som användaren skriver
nu
}
```

Denna metod lägger vi helt enkelt där vi tycker att det passar i klassen `Form1`. Jag lägger den under konstruktorn.

För att anropa denna metod skriver vi t.ex.:

Exempel 15.5

```
AddDigit("1");
```

… för 1-knappen, osv. Vi gör detta på alla knappar från 0 till 9.

Steg 3.3 – Att skapa funktionalitet för plusknappen

Vi har nu två tal som vi ska summera, `sum` och `newNr`. Summan av dessa två tal ska vi åter igen tilldela variabeln `sum`. `newNr` ska bli tom (””). I textboxen ska all text ändras till summan med ett plustecken efteråt.

Det är nog enklast att se detta i kod. Vi lägger till följande funktionalitet för `buttonAdd`:

Exempel 15.6

```
sum = sum + Convert.ToInt32(newNr); // Räkna ut summa av alla tal
newNr = ""; // Återställ så användaren kan skriva ett nytt tal
textBox1.Text = sum + "+"; // Sätt textboxen rätt
```