Mysteriet med stjärnhimmel animationen

Hur får man chars att flytta sig runt i skärmen utan att det blinkar? Hur skriver man ut i flera lager?

Svaret är enkel. Det kan man inte göra. Consolfönstret kan inte göra det. Den har inte double buffer så man kan inte skriva på en osynlig skärm som inte visas förrän man skrivit allt klart. Så hur gör man?

Det finns olika sätt att göra det på, jag valde att köra med en helt vanlig char array. Varför ett char array undrar du? Ett char Array är lätt att skriva ut i consolen.

```
var arr = new char[] { '-', ' ', 'H', 'e', 'l', 'l', 'o', ' ', '-' };
Console.WriteLine(arr);
```

När det gäller att skapa känslan av en animation är hastighet en viktig aspekt, det ska se mjukt ut.

Så nu vet vi att ett char array kan skrivas ut, då använder vi den till att koda med. Detta tänkande skär sig lite när man använder färger, men då den här animationen ska vara svart & vit så kommer det att fungera.

För att förstå tänkandet bättre skapar vi en array med 10 tecken på bredden och 10 på höjden.

```
public class ScreenBuffer
{
    char[][] Buffer;
    public ScreenBuffer()
    {
        Buffer = new char[10][];
    }
}
```

Visst ser det konstigt ut? Vi använder en jagged array istället för en konventionell [10,20] array för att det är enklare att skriva ut via Console.WriteLine. Vi kan nämligen skriva Console.WriteLine (Buffer[0]) och då få med hela raden till WriteLine, det kan vi inte göra om vi använder en konventionell array. Med en konventionell måste vi skriva Console.WriteLine(Buffer[0,x]) som skriver bara ut en rad – och då måste vi loopa. Loopar är bra men långsamma, så det skippar vi.

Vår array är tom, så vi måste definiera innehåller i raderna.

```
public ScreenBuffer()
{
    Buffer = new char[10][];
    for (int y = 0; y < 10; y++)
    {
        Buffer[y] = new char[20];
    }
}
```

Och så gör vi en metod för att skriva ut vår array, i denna måste vi kontrollera att inte `char` 0 skrivs ut för den skrivs inte ut och då blir det knas med utskriften. Vi måste ersätta `char` 0 med `char` 32 som står för mellanslag.

```
public void PrintBuffer()
{
    Console.SetCursorPosition(0, 0);
    for (int y = 0; y < 10; y++)
    {
        var str = new string(Buffer[y]).Replace('\0', ' ');
        Console.WriteLine(str);
    }
}
```

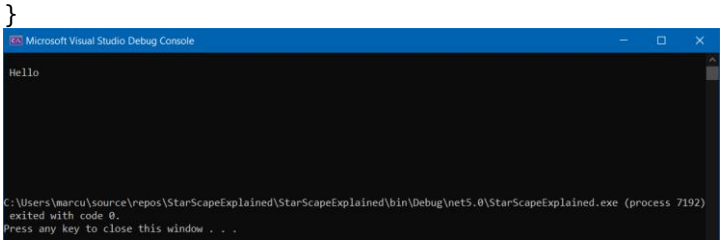
Och vi måste kunna placera tecken i vår array, vi skapar en metod för det med.

```
public void PrintAt(int x, int y, string text)
{
    for (int pos = 0; pos < text.Length; pos++)
    {
        if (pos + x >= 10) break;
        Buffer[y][pos + x] = text[pos];
    }
}
```

Metoden tar emot x, y och text. X är positionen i sidled, Y är raden den ska skrivas på och texten är en sträng med text. Vi loopar igenom texten och kopierar ett `char` i taget från strängen till vår array.

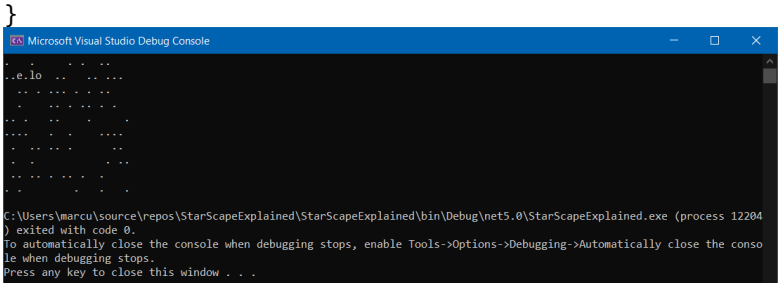
Vi kan nu testa koden

```
static void Main(string[] args)
{
    var buff = new ScreenBuffer();
    buff.PrintAt(1, 1, "Hello");
    buff.PrintBuffer();
}
```



Nu vill vi kunna skapa en stjärnhimmel i bakgrunden, det är enkelt, vi slumpar fram en massa punkter i arrayen.

```
public void AddStars()
{
    Random random = new Random();
    for (int i = 0; i < 100; i++)
    {
        Buffer[random.Next(10)][random.Next(20)] = '.';
    }
}
```



Det blev inte bra ☹

Det är nu vi egentligen förstår varför vi behöver lagra våra skärmobjekt i flera lager.

Så vi lägger till lager på vår array.

```
char[][][] Buffer;
public ScreenBuffer()
{
    Buffer = new char[2][][];
    for (int z = 0; z < 2; z++)
    {
        Buffer[z] = new char[10][];
        for (int y = 0; y < 10; y++)
        {
            Buffer[z][y] = new char[20];
        }
    }
}
```

Och så ändrar vi vår PrintAt till att skriva i lager 0

```
public void PrintAt(int x, int y, string text)
{
    for (int pos = 0; pos < text.Length; pos++)
    {
        if (pos + x >= 10) break;
        Buffer[0][y][pos + x] = text[pos];
    }
}
```

Och så måste vi ändra i vår stjärnhimmel så att den skriver i lager 1

```
public void AddStars()
{
    Random random = new Random();
    for (int i = 0; i < 100; i++)
    {
        Buffer[1][random.Next(10)][random.Next(20)] = '.';
    }
}
```

Och slutligen ändrar vi metoden som skriver ut allt, nu måste vi tyvärr loopa igenom båda lagren för att hitta vad som ska skrivas ut, men oroa dig inte, vi snyggar upp det här senare.

```
public void PrintBuffer()
{
    Console.SetCursorPosition(0, 0);
    for (int y = 0; y < 10; y++)
    {
        var print = new char[20];
        for (int x = 0; x < 20; x++)
        {
            if (Buffer[0][y][x] == '\0')
                print[x] = Buffer[1][y][x];
            else
                print[x] = Buffer[0][y][x];
        }
        var str = new string(print).Replace('\0', ' ');
        Console.WriteLine(str);
    }
}
```

Vi kör en loop som går igenom alla rader, sedan en loop som går igenom alla kolumner. Om tecknet i lager 0 på rad y och kolumn x är 0 så läser vi istället av lager 1 rad y och position x, detta betyder att om översta lagret inte innehåller något, då tar vi det som finns i lagret under.

Oavsett vilket vi väljer så lagrar vi det i ännu en array, denna array är bara en rad. Och när loopen är klar skriver vi ut den arrayen.

Nu kommer allt som finns i lager 0 att skrivas ut oavsett vad som finns i lager 1, men om det inte finns något i lager 0, då fyller vi på med innehåller i lager 1. Vi skulle kunna säga att 0 är lika med ett hål i lagret och då skiner nästa lager igenom.

Nu när vi förstår hur det fungerar ska vi refactora lite

Vi börjar med att ta bort de magiska siffrorna 10 och 20. I början av koden för klassen lägger vi variabler för bredd, höjd och djup.

```
private int Width { get; set; } = 110;
private int Height { get; set; } = 25;
private int Depth { get; set; } = 2;
```

Sedan ändrar vi konstruktorn.

```
public ScreenBuffer(int width, int height, int depth)
{
    Width = width;
    Height = height;
    Depth = depth;
    GenerateMap();
}
public ScreenBuffer()
{
    GenerateMap();
}

private void GenerateMap()
{
    Buffer = new char[Depth][][];
    for (int z = 0; z < Depth; z++)
    {
        Buffer[z] = new char[Height][];
        for (int y = 0; y < Height; y++)
        {
            Buffer[z][y] = new char[Width];
        }
    }
}
```

Sedan får vi anpassa metoderna

```
public void PrintBuffer()
{
    Console.SetCursorPosition(0, 0);
    for (int y = 0; y < Height; y++)
    {
        var print = new char[Width];
        for (int x = 0; x < Width; x++)
        {
            if (Buffer[0][y][x] == '\0')
                print[x] = Buffer[1][y][x];
            else
                print[x] = Buffer[0][y][x];
        }
        var str = new string(print).Replace('\0', ' ');
        Console.WriteLine(str);
    }
}
```

Och

```
public void PrintAt(int x, int y, string text)
{
    for (int pos = 0; pos < text.Length; pos++)
    {
        if (pos + x >= Width) break;
        Buffer[0][y][pos + x] = text[pos];
    }
}
```

Och självklart

```
public void AddStars()
{
    Random random = new Random();
    for (int i = 0; i < 100; i++)
    {
        Buffer[1][random.Next(Height)][random.Next(Width)] = '.';
    }
}
```

Och nu lägger vi till en metod till, denna ska rensa ett helt lager, allt i lagret raderas men andra lager kommer inte att påverkas.

```
public void Clear(int depth)
{
    Buffer[depth] = new char[Height][];
    for (int y = 0; y < Height; y++)
    {
        Buffer[depth][y] = new char[Width];
    }
    RefreshBuffer();
}
```

När vi gör stora ändringar i vår buffer är det bra att refresha det som ska skrivas ut. Metoden kommer att loopa igenom alla rader, och alla kolumner och kontrollera i alla lager för att se vad som ska läggas in i output arrayen.

```
public void RefreshBuffer()
{
    for (int y = 0; y < Height; y++)
    {
        for (int x = 0; x < Width; x++)
        {
            for (int d = 0; d < Depth; d++)
            {
                var ch = Buffer[d][y][x];
                if (ch != '\0')
                {
                    Output[y][x] = ch;
                    break;
                }
            }
        }
    }
}
```

Nu kan vi alltså testa koden såhär:

```
static void Main(string[] args)
{
    var buff = new ScreenBuffer();
    buff.PrintAt(60,10, "Hello");
    while (true)
    {
        buff.Clear(1);
        buff.AddStars();
        System.Threading.Thread.Sleep(200);
        buff.PrintBuffer();
    }
}
```

Nu kan vi se att bakgrunden ändras men inte förgrunden. Så nu är vår klass användbart i alla fall. Vi refactorar lite till.
Vi skapar en ny array kallad Output i början av vår klass

```
private char [][] Output;
```

vi ser till att den töms när vi skapar vår buffer

```
private void GenerateMap()
{
    Buffer = new char[Depth][][];
    Output = new char[Height][];
    for (int z = 0; z < Depth; z++)
    {
        Buffer[z] = new char[Height][];
        for (int y = 0; y < Height; y++)
        {
            Buffer[z][y] = new char[Width];
            Output[y] = new char[Width];
        }
    }
}
```

och så skapar vi en ny metod kallad PrintAt, vi har en redan men det är lugnt för att inparametrarna är olika

```
public void PrintAt(int z, int x, int y, char ch)
{
    Buffer[z][y][x] = ch;
    if (z > 0)
    {
        for (int d = 0; d < Depth; d++)
        {
            ch = Buffer[d][y][x];
            if (ch != '\0') break;
        }
    }

    Output[y][x] = ch;
}
```

Vad den här metoden gör är att den lägger in ett char i arrayen (som vanligt) men den kontrollerar sedan på djupet för att se om det finns något ovanför det lagret där vi skrivit in vårt char. Om det inte är noll kommer den att lämna loopen och lägga värdet i vår array. Om vi skriver i första lagret ska den inte bry sig om de lager som ligger under, för första lagret är den som syns först.

Nu kan vi uppdatera vår print metod till att använda vår nya metod för att skriva text

```
public void PrintAt(int x, int y, string text) => PrintAt(0, x, y, text);
public void PrintAt(int z, int x, int y, string text)
{
    for (int pos = 0; pos < text.Length; pos++)
    {
        if (pos + x >= Width) break;
        PrintAt(z, pos+x, y, text[pos]);
    }
}
```

Och vi uppdaterar stjärnorna till att använda den nya metoden

```
public void AddStars()
{
    Random random = new Random();
    for (int i = 0; i < 100; i++)
    {
        PrintAt(1, random.Next(Width), random.Next(Height), '.');
    }
}
```

Med dessa nya ändringar kan vi nu uppdatera vår print metod till

```
public void PrintBuffer()
{
    Console.SetCursorPosition(0, 0);
    for (int y = 0; y < Height; y++)
    {
        var str = new string(Output[y]).Replace('\0', ' ');
        Console.WriteLine(str);
    }
}
```

Om vi kör programmet kommer den att se ut såhär:

Vad har vi åstadkommit än så länge?

Vi kan skriva en text på skärmen och vi kan ändra vad som sker i den utan att texten påverkas. Vad har vi för nytta av det?

Vi kan göra en meny som inte påverkas av att innehållet i en ruta ändras. Vi kan göra en Pac-man i consolen, eller ett klassiskt break-out spel. Men förutom att göra consol-animationer och consol-spel.

Excel fungerar på ett liknande sätt, det är ett rutnät. I den finns värden, länkar eller beräkningar. Tänk om man skulle använda arrayen på samma sätt.

Lager 0 = värde
Lager 1 = värde från annan cell
Lager 2 = värde som räknas ut

Och när rutnätet ska visas på skärmen kollar man om lager 0 är tom, i så fall kollar man lager 1 om den hänvisar till en annan cell – är den tom så kollar vi lager 2. Självklart skulle vi kunna lagra allt i lager noll, men då måste vi köra en if eller switch för att se vilken typ det är innan det ska visas på skärmen.

Men i mångt och mycket är det enklare att använda listor. Enda orsaken till att gå tillbaka till arrays är för snabbheten. Arrays är i allmänhet snabbare än listor.

Innan vi avslutar det här experimentet så ska vi göra två saker.

1. Rörliga stjärnor
2. Refactoring igen…

För rörliga stjärnor behöver vi ett lager till. Så vi ordnar det nu.

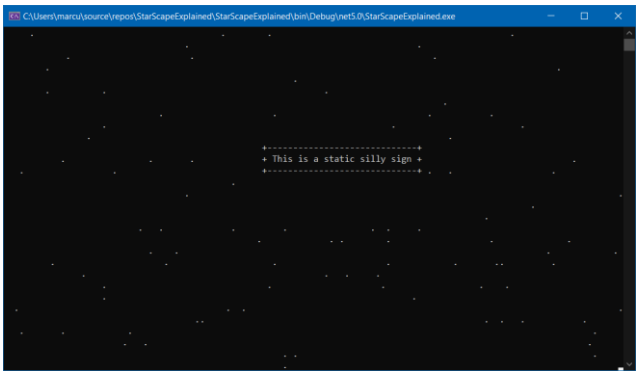
Vi börjar med att anpassa vår main()

```
static void Main(string[] args)
{
    var buff = new ScreenBuffer(Console.WindowWidth, Console.WindowHeight,3);
    buff.PrintAt(50, 10, "+----------------------------+");
    buff.PrintAt(50, 11, "+ This is a static silly sign +");
    buff.PrintAt(50, 12, "+----------------------------+");
    while (true)
    {
        buff.Clear(1);
        buff.AddStars();
        System.Threading.Thread.Sleep(200);
        buff.PrintBuffer();
    }
}
```

Om vi kör programmet nu kommer allt att blinka, för att raderna blir för långa och då scrollar fönstret automatiskt. Det måste vi korrigera innan vi kan fortsätta.

```
public void PrintBuffer()
{
    for (int y = 0; y < Height; y++)
    {
        Console.SetCursorPosition(0, y);
        var str = new string(Output[y]).Replace('\0', ' ');
        Console.Write(str);
    }
}
```

Nu fungerar den som den ska



Vidare till stjärnor då…

```
public class MovingStars
{
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
    public int Xdirection { get; set; } = 1;
    public int Ydirection { get; set; } = 0;
    public int XMax { get; set; }=110;
    public int YMax { get; set; }=25;

    public MovingStars(int x, int y, int xdirection, int ydirection,
                       int xmax, int ymax)
    {
        X = x;
        Y = y;
        Xdirection = xdirection;
        Ydirection = ydirection;
        XMax = xmax;
        YMax = ymax;
    }
    public void MoveStar()
    {
        X += Xdirection;
        Y += Ydirection;
        if (X < 0) X = XMax - 1;
        if (X >= XMax) X = 0;
        if (Y < 0) Y = YMax - 1;
        if (Y >= YMax) Y = 0;
    }
}
```

Varje stjärna har en X och Y position, så de kan röra sig genom arrayen, för att veta vilken riktning de ska gå emot så har vi XDirection och YDirection, dessa påverkar även hastigheten. I metoden MoveStar() så adderar vi helt enkelt position med Direction variablerna, och då vi egentligen inte vet vilken riktning de ska röra sig mot måste vi kontrollera att de inte går utanför arrayens gränser.

Nu får vi ändra i vår main() igen

```
static void Main(string[] args)
{
    var buff = new ScreenBuffer(Console.WindowWidth, Console.WindowHeight, 3);
    buff.PrintAt(50, 10, "+----------------------------+");
    buff.PrintAt(50, 11, "+ This is a static silly sign +");
    buff.PrintAt(50, 12, "+----------------------------+");
    buff.AddStars();
    int amountOfStars = 20;
    var stars = new MovingStars[amountOfStars];
    var random = new Random();
    for (int i = 0; i < amountOfStars; i++)
    {
        stars[i] = new MovingStars(
            random.Next(Console.WindowWidth),
            random.Next(Console.WindowHeight),
            random.Next(1,3), 0,
            Console.WindowWidth, Console.WindowHeight);
    }
    while (true)
    {
        System.Threading.Thread.Sleep(100);
        buff.PrintBuffer();
    }
}
```

Nu lägger vi till de rörliga stjärnorna. Vi skapar en variabel som håller koll på antal stjärnor, för att det ska bli enklare att ändra och för att slippa magiska tal. Sedan skapar vi en array av klassen MovingStar och tilldelar den värden för att få fram effekten. Om vi kör koden nu kommer vi inte att kunna se dem dock, vi behöver anpassa while satsen också.

```
while (true)
{
    for (int i = 0; i < amountOfStars; i++)
    {
        buff.PrintAt(2, stars[i].X, stars[i].Y, '\0');
        stars[i].MoveStar();
        buff.PrintAt(2, stars[i].X, stars[i].Y, '.');
    }
    buff.PrintBuffer();
    System.Threading.Thread.Sleep(100);
}
```

En viktig detalj är att först måste man radera stjärnan från sin ursprungliga plats. Sedan flyttar vi den till en ny plats och skriver ut stjärnan i lagret. Nästa gång loopen körs kommer den positionen att rensa och en ny stjärna placeras in.

Varför gör vi detta istället för att använda Clear metoden? För att Clear Metoden kommer att rensa hela det lagret i arrayen och sedan tvinga bufferten till att uppdatera sig, det innebär en loop som rensar och två loopar som uppdaterar bufferten – vilket i sin tur betyder många nanosekunder som slösas. Det går snabbare att uppdatera en char än 110x25x3 (8250) chars.

Nu kan vi köra koden och se våra fina stjärnor flytta på sig. Hur lägger vi till en flygande pacman eller något annat sådant? På samma sätt som stjärnorna. Skapa en klass som håller koll på vad som ska skrivas, håll koll på X och Y, tänk på att Max värdena ska vara skärmbredd-objektbredd. Så för din Pacman kommer MaxBredd att vara Console.WindowWidth-Pacman.Widh;

Nu kan du ta över projektet. Men innan du skapar ett spel tänk på följande ”*Vad kan gå fel?*” och ”*Vad kan optimeras*”.

Här är lite förslag:

- Vad händer om PrintAt får ett lagernummer som är större än definierad mängd?
- Vad händer om PrintAt får ett X som är större än definierad bredd?
- Vad händer om PrintAt får ett Y som är större än definierad höjd?
- Vad händer om PrintAt får ett text som är längre än definierad bredd?
    - Ingenting 😉 det är fixat, den bjuder jag på
- Vad händer om Clear får ett för högt eller för lågt värde?
- Vad händer om du mitt i programmet vill utöka antal lager? Kan man göra det?
- Vad händer om bredden på arrayen är större än Console.WindowWidth?
- Vad händer om höjden på arrayen är större än Console.WindowHeight?

Säkra upp koden och skapa tester som visar att det fungerar.

Optimeringsförslag:

- Kan man hålla koll på vilka rader som ändrats och bara skriva ut dem?
- Kan man skicka in en array till PrintAt istället för en rad i taget?
- Ska `PrintBuffer` metoden hantera stjärnornas position och rörelse?
- Ska alla rörliga objekt man skapar köras via Interface och sen låta `PrintBuffer` eller en Update metod hantera rörelserna och uppdatering?
- Ska man låta varje lager ha en egen klass så att de kan sköta sig själva?
- Koden är (medvetet 😉 slarvigt skrivet, den kan refactoras en hel del), snygga upp den
- Om du förstår metoderna, lägg till kommentarer

Förbättringsförslag:

- Tänk om man vill ha en buffer som är större än vad consolen kan visa, hur löser man det?

Utmaning

- Gör om hela projektet från scratch men använd Console färger iställer, då fungerar inte WriteLine men man kan optimera när man skriver ut dem…

När du är klar med uppgiften, gör en cool animation, lägg till lite ASCII grafik, zippa den och skicka den till mig så lägger jag det på Github i klassens mapp.