Praksis eksempel
Operatorer
Løkker
Metoder
Klasser
Arv

Grundlæggende programmering

Lektion 3

Praksis eksempel

Et program der benytter Unity til at lave et 3D spil

Praksis eksempel

Spil der benytter Unity og er scriptet i C#

Unity miljøet til spil og 3D app udvikling er ekstremt populært, og det benyter C# som scripting sprog – uanset om man vælger at compilere ens app til Windows, Linux eller Mac. Der er også webplayers og mulighed for at benytte Unity til mobil-apps.

Praksis eksempel

Kort introduktion til Unitys brug af C#

En Unity klasse-fil ser nogenlunde sådan her:

```
1   using UnityEngine;
    0 references
2   public class Mook : MonoBehaviour
3   {
4       private float health;
        0 references
5       void Start()
6       {
7           health = 100;
8       }
        0 references
9       void Update()
10      {
11          if (health > 0)
12          {
13              //search for player
14              //if you encounter the player on the road, kill him
15              //if you get shot, remove a random amount of health
16          }
17      }
18  }
```

Praksis eksempel

Kort introduktion til Unitys brug af C#

- *using UnityEngine;* –Fortæller C# at vi vil benytte Unitys libraries, der lader os forbinde til Unity spil enginen.
- *public class Mook : MonoBehaviour {* – Denne linje deklarerer klassen og dens navn ("Mook");
- *private float health;* –Dette erklærer en privat klasse variabel (som kun kan ændres inde fra klassen). Variablen får en værdi i Start ().
- *void Start () {* –Dette deklarerer en metode kaldet "Start." Start er en særlig metode, der kører kun én gang, når spillet første gang startes.
- void Update(){ -Update er en anden særlig metode, som kører på hver frame. De fleste af ens spil logik vil blive lagt her.

PaddleHandler.cs

Praksis eksempel

```
using UnityEngine;
using System.Collections;

0 references
public class PaddleHandler : MonoBehaviour
{
    //this creates an empty ParticleSystem variable that we can attach to the correct particlesystem in the editor
    public ParticleSystem collision;
    // Use this for initialization
    0 references
    void Start()
    {
    }
    // Update is called once per frame
    0 references
    void Update()
    {
        //this if statement moves the paddle down
        if (transform.position.z > 0 && Input.GetKey(KeyCode.UpArrow))
        {
            float zup = transform.position.z - 16 * Time.deltaTime;
            transform.position = new Vector3(transform.position.x, 0, zup);
        }
        //this if statement moves the paddle up
        if (transform.position.z < 7 && Input.GetKey(KeyCode.DownArrow))
        {
            float zdown = transform.position.z + 16 * Time.deltaTime;
            transform.position = new Vector3(transform.position.x, 0, zdown);
        }
    }
    //this plays the particle effect during a collision
    0 references
    void OnCollisionEnter(Collision other)
    {
        collision.Play();
    }
}
```

Operatorer

Syntaks til at udføre forskellige beregninger og handlinger

Booleans og hvorfor de er essentielle og praktiske

Operatorer
Ternære
Flow kontrol

Null-Coalescing Operator (??)

Null-coalescing operatoren bruges når hvis den ene værdi er null så en anden skal bruges.

expression1 ?? expression2

Denne operator kører også en slags kortslutning. Hvis expression1 ikke er null er resultatet af operationen dets værdi og det andet udtryk behandles ikke. Hvis expression1 *er* null bliver expression2 værdien af operatoren.

Operatorer
Ternære
Flow kontrol

Null-Conditional Operator (?.)

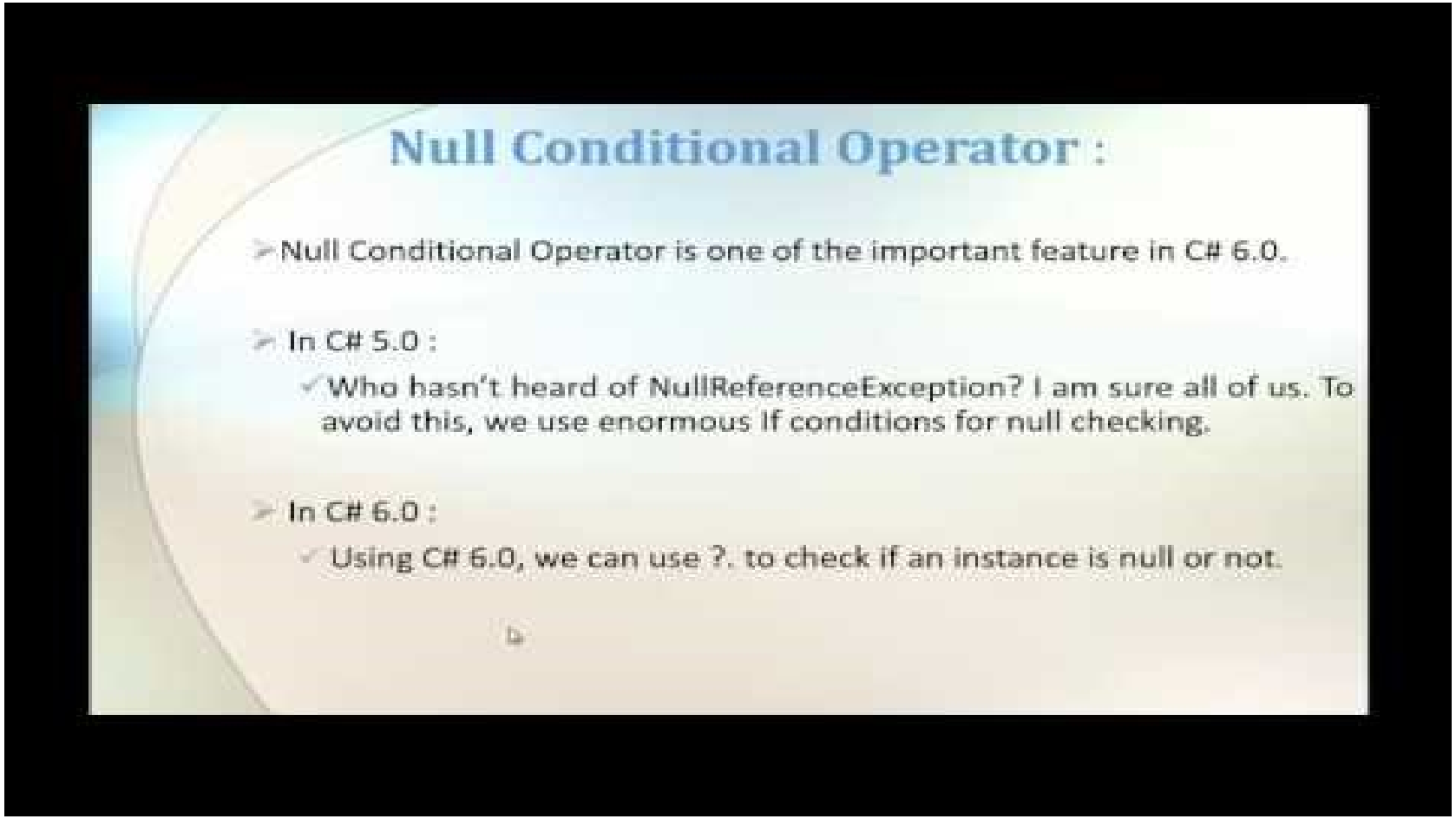
Når man benytter en metode på en værdi, der er null, vil runtime smide en System.NullReferenceException, som næsten altid viser en fejl i programmeringens logik.

For at anerkende dette problem har C# **null-conditional operator**. Den kontrollerer, om operand (de første i eksemplet herunder) er null før den kalder metoden eller property (length i eksemplet).
Den logisk ækvivalente eksplicitte kode ville være følgende (selvom værdien af args i C # 6.0 syntaks kun evalueres én gang)

```
(args != null) ? (int?)args.Length : null
```

Hvad gør null-conditional operator praktisk er at den kan lænkes.
Hvis man kalder args [0] ?. ToLower ().StartsWith ("File:") vil både ToLower () og StartsWith() blive kaldt hvis args [0] ikke er nul.
Når udtrykket er lænket vil det betyde at hvis den første operand er nul vil udtrykkets evaluering kortsluttes, og ingen yderligere kald i udtrykket vil forekomme.

Operatorer
Ternære
Flow kontrol



Løkker

Løkker får verden til at køre rundt

Løkker

Løkker

- En løkke er når en blok kode køres flere gange.
- Termen loop body henviser til et statement, typisk en kode blok, der køres i et loop indtil afslutnings-kravet er mødt.

Forskellige former for løkker

- Man bruger while til at gentage (iterate) så længe kravet er true.
- En for løkke bruges mest hensigtsmæssigt, når antallet af gentagelser er kendt, såsom når der tælles fra o til n.
- En do/while ligner en while-løkke, men den vil altid udføre løkken kroppen mindst én gang.

Løkker While

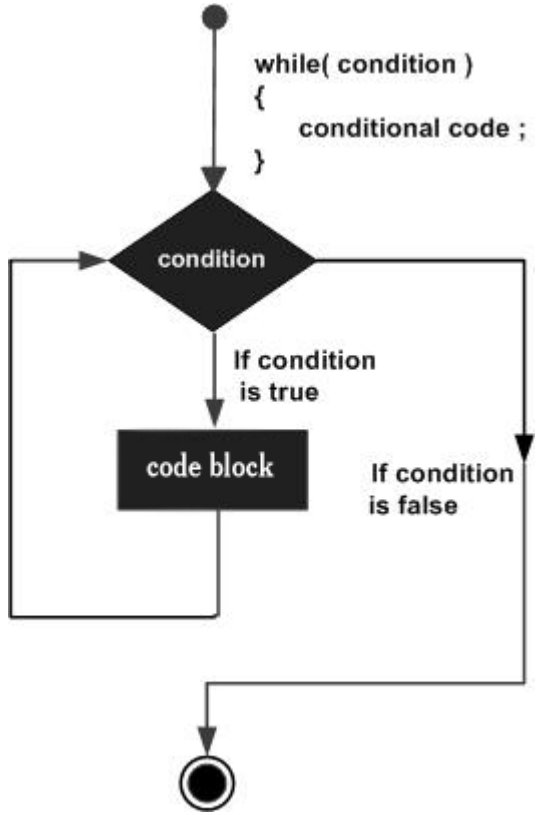
While løkker

While løkken er det enkleste betingede loop. Den almindelige form af while-sætningen er:

while (*condition*)

statement

Loopet kører det statement, der danner kroppen i udtrykket, rundt så længe at conditionen (der *skal* være boolean) er true. Hvis den bliver *false* dropper udførelsen af koden kroppen og går videre til koden efter loop statementet.



Løkker
While

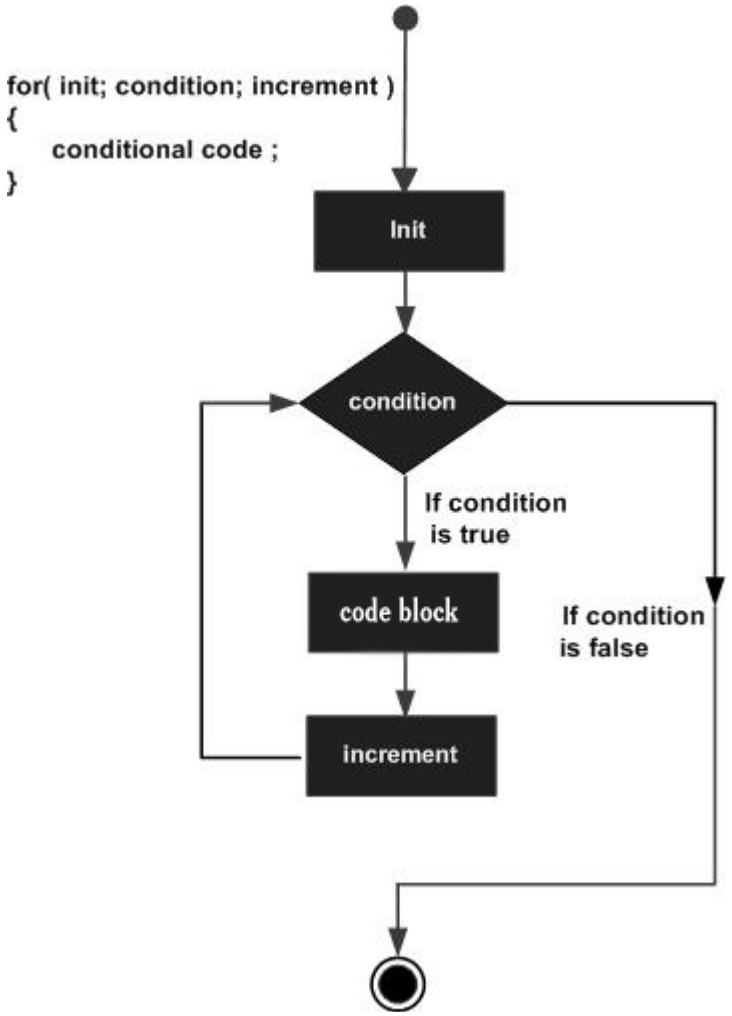
While løkker

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace ConsoleApplication7
8   {
        0 references
9       class Program
10      {
            0 references
11          static void Main(string[] args)
12          {
13              /* local variable definition */
14              int a = 10;
15
16              /* while loop execution */
17              while (a < 20)
18              {
19                  Console.WriteLine("value of a: {0}", a);
20                  a++;
21              }
22              Console.ReadLine();
23          }
24      }
25  }
```

Løkker
For

For løkker

- En for-løkke gentager en kode blok indtil en bestemt betingelse er nået.
- I forhold til while løkken har for-løkken indbygget syntaks for initialisering, forøgelse og afprøvning af værdien af en tæller, kaldet loop variablen.
- Fordi der er en specifik placering i loopets syntaks for en tilvækst operation, anvendes increment og decrement operatører ofte som en del af en for-løkke.



Løkker
For

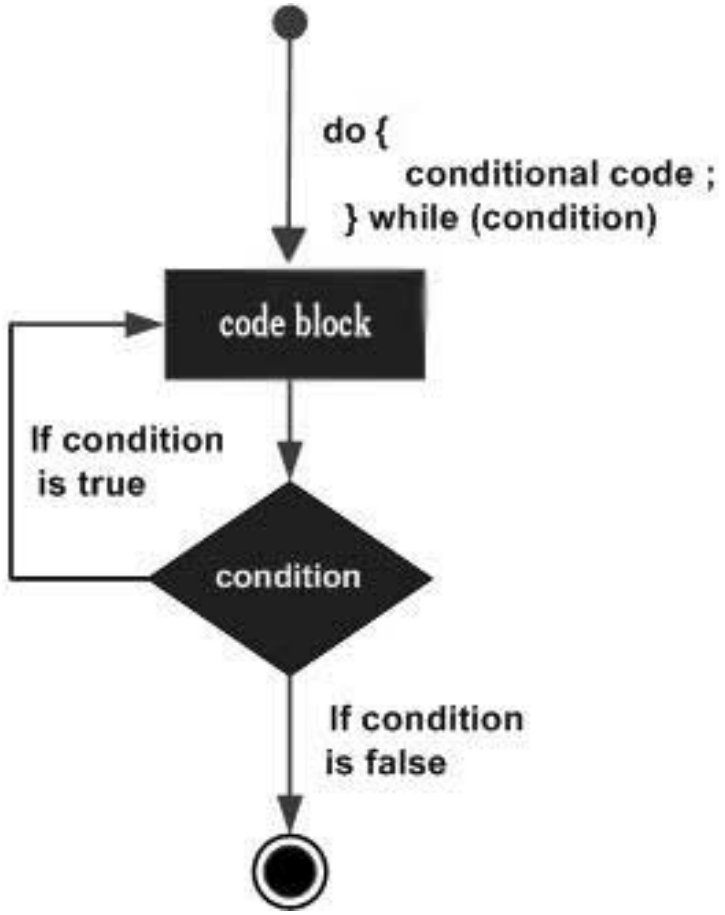
For løkker

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace ConsoleApplication7
8   {
        0 references
9       class Program
10      {
            0 references
11          static void Main(string[] args)
12          {
13              /* for loop execution */
14              for (int a = 10; a < 20; a = a + 1)
15              {
16                  Console.WriteLine("value of a: {0}", a);
17              }
18              Console.ReadLine();
19          }
20      }
21  }
```

Løkker
Do… while

Do… while løkker

- Do / while-løkken er meget lig while løkken, bortset fra at do / while løkken foretrækkes, når antallet af gentagelser er fra 1 til n, og n er ikke er kendt når iterationen begynder.
  - Dette opstår ofte når man spørger en bruger efter input.



Løkker
Do… while

Do… while løkker

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication7
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* do loop execution */
            do
            {
                Console.WriteLine("value of a: {0}", a);
                a = a + 1;
            }
            while (a < 20);
            Console.ReadLine();
        }
    }
}
```

Løkker

for Loop

```
class Program
    {
        static void Main(string[] args)
        {
            for(int i=1; i<10; i++)
            {
                Console.WriteLine("Count is: " + i);
            }
            Console.ReadLine();
        }
    }
```

Metoder

Sekvenser af kode om samme emne

Metoder

- En metode er en måde at samle en sekvens af statements der udfører en bestemt handling eller beregner et bestemt resultat.
  - Dette giver større struktur og organisation for de statements, der opbygger et program.
- Hvert C # program har mindst en klasse med en metode kaldet Main.
- For at benytte en metode skal man:
  - Definere metoden.
  - Kalde metoden.
- Vi har brugt metoder hidtil, primært main() metoden, hvori hele vores programmer lå, men med mere komplekse programmer kan det være praktisk eller nødvendigt at benytte flere metoder.

Metoder

- En metodes grund struktur er

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
  Method Body
}
```

- **Access Specifier** (scope): Bestemmer synligheden af en variabel eller en metode for andre klasser.
- **Return type**: En metode kan returnere en værdi. Hvis metoden ikke returnerer nogen værdier, så er retur typen ugyldig (void).
- **Method name**: Metode navn er et entydigt id, og det er følsomt overfor store og små bogstaver. Det kan ikke være det samme som en anden identifikator erklæret i klassen.
- **Parameter list** (parametre og argumenter): Lukket inde mellem parenteser finder man de parametre, der anvendes til at sende og modtage data fra metoden. Parameterlisten refererer til typen, rækkefølgen og antallet af parametre i metoden. Parametre er valgfri, det vil sige at en metode ikke behøves have nogen parametre.
- **Method body**: Metodens krop indeholder det sæt af instruktioner, der er nødvendige for at gennemføre den ønskede aktivitet.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication7
{
    2 references
    class NumberManipulator
    {
        1 reference
        public int FindMax(int num1, int num2)
        {
            /* local variable declaration */
            int result;

            if (num1 > num2)
                result = num1;
            else
                result = num2;

            return result;
        }
    }

    0 references
    class Test
    {
        0 references
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();

            //calling the FindMax method
            ret = n.FindMax(a, b);
            Console.WriteLine("Max value is : {0}", ret);
            Console.ReadLine();
        }
    }
}
```

Metoder

Rekursive metode kald

En metode kan kalde sig selv. Dette er kendt som rekursion. Følgende er et eksempel, som beregner fakultetet for et givet tal ved hjælp af en rekursiv funktion:

```
9     class NumberManipulator
10    {
          4 references
11        public int factorial(int num)
12        {
13            /* local variable declaration */
14            int result;
15            if (num == 1)
16            {
17                return 1;
18            }
19            else
20            {
21                result = factorial(num - 1) * num;
22                return result;
23            }
24        }
          0 references
25        static void Main(string[] args)
26        {
27            NumberManipulator n = new NumberManipulator();
28            //calling the factorial method
29            Console.WriteLine("Fakultet for 6 er : {0}", n.factorial(6));
30            Console.WriteLine("Fakultet of 7 er : {0}", n.factorial(7));
31            Console.WriteLine("Fakultet of 8 er : {0}", n.factorial(8));
32            Console.ReadLine();
33        }
34    }
```

Metoder

- En metode er *ikke* et statement selvom man kunne tro det.
  - System.Console .WriteLine ("! Hej {o}", System.Console.ReadLine ()); er et enkelt statement, der indeholder to metodeopkald. Et statement indeholder ofte en eller flere udtryk, og i dette eksempel er to af disse udtryk er metodekald. Derfor danner metodekald dele af udsagn.
  - Selvom kodning af flere metodekald i et enkelt statement ofte vil reducerer mængden af kode, betyder det ikke nødvendigvis at det øger læsbarheden, og det giver sjældent en betydelig ydelses fordel. Udviklere bør altid favorisere læsbarheden over kortfattethed.

Metoder Refactoring

- Det at flytte en række udsagn ind i en metode i stedet for at lade dem være inline i en større metode er en form for refactoring.
- Refactoring reducerer kode dobbeltarbejde, fordi man kan kalde metoden fra flere steder i stedet for at duplikere koden.
- Refactoring øger dermed også kodens læsbarhed.
- Som del af kodning processen det er en bedste praksis øbende at gennemgå ens kode og se efter muligheder for at refactore.
  - Dette indebærer at blokke af kode, der er vanskelige at forstå og få et overblik over, flyttes ind i en metode med et navn, der klart definerer kodens adfærd.
  - Denne praksis er ofte foretrukket frem for at kommentere en blok af kode, fordi metodens navn tjener til at beskrive, hvad implementeringen gør.

Metoder Namespaces

- Som I har bemærket begynder vi alle C# programmer med at definere et namespace.
- Namespaces er en kategoriserings mekanisme til gruppering af alle typer relateret til et bestemt område af funktionalitet.
- Namespaces er hierarkiske og kan have vilkårligt mange niveauer i hierarkiet, omend namespaces med mere end en halv snes niveauer er sjældne.
- Typisk begynder hierarkiet med et firmanavn, og derefter et produktnavn, og derefter den funktionelle område.
  - For eksempel i Microsoft.Win32.Networking, er det yderste navneområde Microsoft, som indeholder et indre navneområde Win32, som igen indeholder endnu et mere dybt indlejret Networking namespace.
  - Det burde virke bekendt i forhold til using i starten af vores programmer.

Metoder Namespaces

- Namespaces bruges primært til at organisere typer efter område af funktionalitet så de lettere kan findes og forstås.
- De kan dog også anvendes til at undgå Typenavn kollisioner. Compileren kan f.eks. skelne mellem to typer med navnet Button så længe hver type er under forskellige namespaces.
  - Således er System.Web.UI.WebControls.Button og System.Windows.Controls.Button forskellige.
- Det er ikke altid nødvendigt at angive et namespace når du kalder en metode. Compileren kan f.eks. udlede at hvis det kaldte udtryk optræder i det samme namespace som den kaldte metode er namespace det samme som det, der indeholder den type.

God programmerings skik

- Benyt PascalCasing (hvert ord begynder med stort) til namespace navne.
- Overvej at organisere bibliotekets hierarki for kildekode filer så det matcher namespace hierarkiet.

Metoder Using

- Fulde namespace navne bliver hurtigt lange og klodsede, så man kan "importere" indholdet af et eller flere namespaces ind i en fil så man ikke behøves angive de fulde navne.

```
using System;
...
Console.WriteLine("Hello, my name is Inigo Montoya");
```

- Eksemplet lader os benytte .Console fra System uden at skulle angive det forrest.
- Du kan dog ikke benytte elementer fra child namespaces, så hvis man vil have fat i StringBuilder fra System.Text namespacet skal det også kobles på med using System.Text; direktivet eller man må angive typen som System.Text.StringBuilder – ikke bare Text.Stringbuilder!

Metoder Using

- static direktivet giver mulighed for at udelade både namespace og type navnet fra et medlem af den angivne type.

```
7   namespace ConsoleApplication7
8   {
9       using static System.Console;
            0 references
10      class HeyYou
11      {
                0 references
12          static void Main()
13          {
14              string firstName;
15              string lastName;
16              WriteLine("Hey you!");
17              Write("Enter your first name: ");
18              firstName = ReadLine();
19              Write("Enter your last name: ");
20              lastName = ReadLine();
21              WriteLine($"Your full name is { firstName } { lastName }.");
22              ReadLine();
23          }
24      }
25  }
```

Metoder Using

- using direktivet giver også mulighed for aliasing af et namespace eller type. Et alias er et alternativt navn, som man kan bruge i teksten som direktivet påvirker.
- De to mest almindelige årsager til aliasing er til at kende forskel på to typer, der har det samme navn og til at forkorte et langt navn.

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    3 references
    public class MyClass
    {
        1 reference
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    3 references
    class MyClass<T>
    {
        1 reference
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    // Using directive:
    using NameSpace1;
    // Using directive:
    using NameSpace2;

    0 references
    class MainClass
    {
        0 references
        static void Main()
        {
            AliasToMyClass instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            UsingAlias instance2 = new UsingAlias();
            Console.WriteLine(instance2);

            Console.ReadLine();
        }
    }
}
```

Metoder Main

Main metode er indgangen til en C # konsol applikation eller et Windows-program. (Biblioteker og tjenester kræver ikke en Main metode som indgang.). Når programmet startes er Main metoden den første metode, der invokes.

- Main metode er indgangen i et .exe program; det er hvor program styring starter og slutter.
- Main erklæres inde i en klasse eller struct. Main skal være static, og den bør ikke være offentlig. Den omsluttende klasse eller struct er ikke forpligtet til at være statisk.
- Main kan enten have en void eller int return type.
- Main metoden kan erklæres med eller uden en string [] parameter, der indeholder kommandolinjeargumenter. Når du bruger Visual Studio til at oprette Windows Forms applikationer, kan du tilføje parameteren manuelt eller andre bruge Environment klassen til at få fat i kommandolinjeargumenter. Parametre læses som nul-indekserede kommandolinjeargumenter.
  I modsætning til C og C ++, er navnet på programmet ikke behandlet som det første kommando-line argument.

Metoder Main

Main metode er indgangen til en C # konsol applikation eller et Windows-program. (Biblioteker og tjenester kræver ikke en Main metode som indgang.). Når programmet startes er Main metoden den første metode, der invokes.

- Main metode er indgangen i et .exe program; det er hvor program styring starter og slutter.
- Main erklæres inde i en klasse eller struct. Main skal være static, og den bør ikke være offentlig. Den omsluttende klasse eller struct er ikke forpligtet til at være statisk.
- Main kan enten have en void eller int return type.
- Main metoden kan erklæres med eller uden en string [] parameter, der indeholder kommandolinjeargumenter. Når du bruger Visual Studio til at oprette Windows Forms applikationer, kan du tilføje parameteren manuelt eller andre bruge Environment klassen til at få fat i kommandolinjeargumenter. Parametre læses som nul-indekserede kommandolinjeargumenter.
  I modsætning til C og C ++, er navnet på programmet ikke behandlet som det første kommando-line argument.

Metoder Main

```
using System;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        if (args == null)
        {
            Console.WriteLine("args is null"); // Check for null array
        }
        else
        {
            Console.Write("args length is ");
            Console.WriteLine(args.Length); // Write array length
            for (int i = 0; i < args.Length; i++) // Loop through array
            {
                string argument = args[i];
                Console.Write("args index ");
                Console.Write(i); // Write index
                Console.Write(" is [");
                Console.Write(argument); // Write string
                Console.WriteLine("]");
            }
        }
        Console.ReadLine();
    }
}
```

Metoder

Klasser

Kode der gør ting

Klasser

Klasser og objekter og objekter lavet ud fra dem er hvad der gør C# til et objekt orienteret programmerings sprog.

- Når man definerer en klasse definere en man en plan for en datatype.
  - Dette definerer faktisk ikke nogen data, men det definerer, hvad klasse navnet betyder. Det vil sige, hvilket objekt klassen består af og hvilke operationer der kan udføres på det pågældende objekt.
  - Objekter er instanser af en klasse. De metoder og variabler, der udgør en klasse, kaldes medlemmer af klassen.
- En klasse er en skabelon for hvordan et objekt vil se ud på instantiering tidspunktet. Et objekt er derfor en instans af en klasse.
  - Det at lave et objekt fra en klasse kaldes derfor instantiation.

Klasser

- En klasse definition starter med keyword class efterfulgt af navnet på klassen og klassens krop indesluttet af et par krøllede parenteser.

```
<access specifier> class  class_name
{
  // member variables
  <access specifier> <data type> variable1;
  <access specifier> <data type> variable2;
  ...
  <access specifier> <data type> variableN;
  // member methods
  <access specifier> <return type> method1(parameter_list)
  {
    // method body
  }
  <access specifier> <return type> method2(parameter_list)
 {
    // method body
  }
  ...
  <access specifier> <return type> methodN(parameter_list)
  {
    // method body
  }
}
```

Klasser

Noter:

- Access specifiers angiver adgangs-reglerne for medlemmerne såvel som klassen selv. Hvis den ikke er angivet er standard access specifier for klasse typen *internal*. Standard adgang for members er *private*.
- Data type angiver variabel typen, og return type angiver typen af data som metoden sender tilbage, hvis nogen.
- For at tilgå klasse medlemmer bruger man punktum (.) operatoren.
- Punktum operatoren forbinder navnet på et objekt med navnet på et medlem.

Klasser
Encapsulation

Medlems variabler, funktioner og indkapsling

Et af de vigtigste aspekter af objektorienteret design er den gruppering af data der giver struktur.

Det generelle objektorienteret betegnelse for en variabel, der gemmer data inden en klasse er medlems variabel (member variable) . Dette udtryk er velkendt i C #, men en mere standard betegnelse er felt (field), som er en navngivet lagerenhed tilknyttet indeholdende type.

Et medlems funktion af en klasse er en funktion, der har sin definition eller dets prototype af definitionen i klassen ligesom enhver anden variabel. Det fungerer på ethvert objekt af klassen, som den er medlem af, og har adgang til alle medlemmer af en klasse for dette objekt.

Medlems variablerne variabler er attributter af et objekt (fra et design perspektiv), og de holdes private for at implementere indkapsling. Disse variabler kan kun tilgås ved hjælp af de offentlige medlem funktioner (public member functions).

```
using System;
namespace BoxApplication
{
    4 references
    class Box
    {
        public double length;   // Length of a box
        public double breadth;  // Breadth of a box
        public double height;   // Height of a box
    }
    0 references
    class Boxtester
    {
        0 references
        static void Main(string[] args)
        {
            Box Box1 = new Box();   // Declare Box1 of type Box
            Box Box2 = new Box();   // Declare Box2 of type Box
            double volume = 0.0;    // Store the volume of a box here

            // box 1 specification
            Box1.height = 5.0;
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // box 2 specification
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // volume of box 1
            volume = Box1.height * Box1.length * Box1.breadth;
            Console.WriteLine("Volume of Box1 : {0}", volume);

            // volume of box 2
            volume = Box2.height * Box2.length * Box2.breadth;
            Console.WriteLine("Volume of Box2 : {0}", volume);
            Console.ReadKey();
        }
    }
}
```

Klasser
Constructors

- En klasse constructor er en særlig member function af en klasse der udføres hver gang vi laver ny objekter af klassen.
- En constructor har det præcist same navn som klassen og ingen return type.

```
1   using System;
2   namespace LineApplication
3   {
        3 references
4       class Line
5       {
6           private double length;   // Length of a line
            1 reference
7           public Line()
8           {
9               Console.WriteLine("Object is being created");
10          }
11
            1 reference
12          public void setLength(double len)
13          {
14              length = len;
15          }
16
            1 reference
17          public double getLength()
18          {
19              return length;
20          }
21
            0 references
22          static void Main(string[] args)
23          {
24              Line line = new Line();
25
26              // set line length
27              line.setLength(6.0);
28              Console.WriteLine("Length of line : {0}", line.getLength());
29              Console.ReadKey();
30          }
31      }
32  }
```

Klasser Constructors

- En default constructor har ikke nogen parameter, men hvis man ønsker det kan en constructor have det.
- Sådanne constructors kaldes parameterized constructors. Denne teknik hjælper dig med at tildele oprindelige værdier til et objekt ved tidspunktet for dets oprettelse.

```
1   using System;
2   namespace LineApplication
3   {
        3 references
4       class Line
5       {
6           private double length;   // Length of a line
            1 reference
7           public Line(double len)  //Parameterized constructor
8           {
9               Console.WriteLine("Object is being created, length = {0}", len);
10              length = len;
11          }
12
            1 reference
13          public void setLength(double len)
14          {
15              length = len;
16          }
            2 references
17          public double getLength()
18          {
19              return length;
20          }
21
            0 references
22          static void Main(string[] args)
23          {
24              Line line = new Line(10.0);
25              Console.WriteLine("Length of line : {0}", line.getLength());
26
27              // set line length
28              line.setLength(6.0);
29              Console.WriteLine("Length of line : {0}", line.getLength());
30              Console.ReadKey();
31          }
32      }
33  }
```

Klasser
Access modifiers

Yderligere indkapsling

Access modifiers identificerer niveauet for indkapsling forbundet med medlemmet (member) de hører til.

De findes fem access modifiers:

- Public
- Protected
- Internal
- Protected internal
- Private

Klasser
Access modifiers

Public

Hvis et member eller en type har public access modifier har det det mest eftergivende adgangsniveau. Der er ingen restriktioner på adgang til public members.

Protected

En beskyttet medlem er tilgængeligt inden for sin klasse og ved afledte klasse instanser.

Internal

Adgangen er begrænset til det aktuelle assembly (f.eks. samme fil).

Klasser
Access modifiers

Protected internal

Adgangen er begrænset til det aktuelle assembly eller typer afledt af den indeholdende klasse.

Private

Private access er det mindst eftergivende adgangsniveau. Private medlemmer er kun tilgængelige i kroppen af klassen eller struct, hvor de er blevet erklæret.

Klasser
Properties

Protected internal

Adgangen er begrænset til det aktuelle assembly eller typer afledt af den indeholdende klasse.

Private

Private access er det mindst eftergivende adgangsniveau. Private medlemmer er kun tilgængelige i kroppen af klassen eller struct, hvor de er blevet erklæret.

Klasser
Destructors

En destructor er en særlig medlems funktion af en klasse, der udføres, når en genstand i sin klasse går ud af scope.

En destructor har nøjagtig samme navn som klassen med en foranstillet tilde (~).

Destructor kan være meget nyttig for at frigive hukommelses ressourcer før du afslutter programmet

- Destructors kan ikke defineres i structs. De anvendes kun med klasser.
- En klasse kan kun have én destructor.
- Destructors kan ikke arves eller overbelastet.
- Destructors kan ikke kaldes. De bliver automatisk kaldt
- Destructor har ikke nogen værdi.
- En destructor tager ikke imod modifikatorer eller har parametre.

Klasser
Destructors

```
1   using System;
2
    4 references
3   class Example
4   {
        1 reference
5       public Example()
6       {
7           Console.WriteLine("Constructor");
8       }
9
        0 references
10      ~Example()
11      {
12          Console.WriteLine("Destructor");
13      }
14  }
15
    0 references
16  class Program
17  {
        0 references
18      static void Main()
19      {
20          Example x = new Example();
21          Console.ReadKey();
22      }
23  }
```

Klasser
Static

Vi kan definere klasse members som statiske ved hjælp af static nøgleordet. Når vi erklærer et medlem af en klasse som statisk, betyder det ligegyldigt hvor mange objekter af klassen der er lavet, der er kun én kopi af det statiske element.

Statiske variabler anvendes til at definere konstanter, fordi deres værdier kan hentes ved at påberåbe klassen uden at oprette en instans af det.

Statiske variabler kan initialiseres udenfor medlems funktioner eller klasse definitioner. Man kan også initialisere statiske variabler inde i klassen definitionen.

Klasser Static

```
1   using System;
2   namespace StaticVarApplication
3   {
        4 references
4       class StaticVar
5       {
6           public static int num;
            6 references
7           public void count()
8           {
9               num++;
10          }
            2 references
11          public int getNum()
12          {
13              return num;
14          }
15      }
        0 references
16      class StaticTester
17      {
18          static void Main(string[] args)
19          {
20              StaticVar s1 = new StaticVar();
21              StaticVar s2 = new StaticVar();
22              s1.count();
23              s1.count();
24              s1.count();
25              s2.count();
26              s2.count();
27              s2.count();
28              Console.WriteLine("Variable num for s1: {0}", s1.getNum());
29              Console.WriteLine("Variable num for s2: {0}", s2.getNum());
30              Console.ReadKey();
31          }
32      }
33  }
```

Klasser
Structs

Structs er defineret af struct nøgleordet, for eksempel:

```
public struct PostalAddress
{
   // Fields, properties, methods and events go here...
}
```

Structs deler det meste af syntaks med klasser, selv om structs er mere begrænset end klasser:

- Inden for en struct erklæring, kan felterne ikke initialiseres, medmindre de angives som const eller static.
- En struct kan ikke erklære en standard-constructor (en konstruktør uden parametre) eller en destructor.
- Structs kopieres ved assignment. Når en struct tildeles en ny variabel bliver al data kopieret, og enhver ændring til den nye kopi ændrer ikke data for den originale kopi. Det er vigtigt at huske, når du arbejder med samlinger af værdi typer såsom Dictionary <string, myStruct>.
- Structs er værdi typer og klasser er referencetyper.

Klasser
Structs

- I modsætning til klasser kan structs instantieres uden brug af en ny operator.
- Structs kan erklære konstruktører, der har parametre.
- En struct kan ikke arve fra en anden struct eller klasse, og den kan ikke danne basis for en klasse. Alle structs arver direkte fra System.ValueType, som arver fra System.Object.
- En struct kan implementere grænseflader.
- En struct kan bruges som en nullable type og kan tildeles en null-værdi.

Klasse
Struct

```
1   using System;
    2 references
2   struct SimpleStruct
3   {
4       private int xval;
        1 reference
5       public int X
6       {
7           get
8           {
9               return xval;
10          }
11          set
12          {
13              if (value < 100)
14                  xval = value;
15          }
16      }
17      public void DisplayX()
18      {
19          Console.WriteLine("The stored value is: {0}", xval);
20      }
21  }
22
    0 references
23  class TestClass
24  {
        0 references
25      public static void Main()
26      {
27          SimpleStruct ss = new SimpleStruct();
28          ss.X = 5;
29          ss.DisplayX();
30          Console.ReadKey();
31      }
32  }
```

Klasser

Arv

Et program der benytter Unity til at lave et 3D spil

Arv

En af de vigtigste begreber i objektorienteret programmering er arv. Arv giver os mulighed for at definere en klasse i form af en anden klasse, hvilket gør det lettere at oprette og vedligeholde en applikation. Dette giver også mulighed for at genbruge kodens funktionalitet og fremskynder dermed implementeringstid.

Når man opretter en klasse, i stedet for at skrive helt nye data medlemmer og medlems funktioner, kan man udpege, at den nye klasse skal arve medlemmerne fra en eksisterende klasse. Denne eksisterende klasse kaldes base klasse eller forælder, og den nye klasse betegnes som den afledte klasse eller barn.

Ideen om arv implementerer IS-A relationer. F.eks er pattedyret IS-A dyr, hund IS-A pattedyr dermed er hund IS-A dyr også, og så videre.

Arv

En klasse kan være afledt af mere end én klasse eller grænseflade, hvilket betyder, at det kan arve data og funktioner fra flere base klasser eller interfaces.

En normal forælder-barn / base-afledt struktur kan illustreres sådan:

```
<acess-specifier> class <base_class>
{
  ...
}
class <derived_class> : <base_class>
{
  ...
}
```

Arv

```
using System;
namespace InheritanceApplication
{
    1 reference
    class Shape
    {
        1 reference
        public void setWidth(int w)
        {
            width = w;
        }
        1 reference
        public void setHeight(int h)
        {
            height = h;
        }
        protected int width;
        protected int height;
    }

    // Derived class
    2 references
    class Rectangle : Shape
    {
        1 reference
        public int getArea()
        {
            return (width * height);
        }
    }

    0 references
    class RectangleTester
    {
        0 references
        static void Main(string[] args)
        {
            Rectangle Rect = new Rectangle();

            Rect.setWidth(5);
            Rect.setHeight(7);

            // Print the area of the object.
            Console.WriteLine("Total area: {0}", Rect.getArea());
            Console.ReadKey();
        }
    }
}
```

Arv

Den afledt klasse arver basisklassens medlemsvariabler og medlems metoder. Derfor bør super klasse objektet laves før underklassen oprettes. Du kan give instruktioner til superklassens initialisering i medlemmets initialiserings liste.

```
1   using System;
2   namespace RectangleApplication
3   {
        2 references
4       class Rectangle
5       {
6           //member variables
7           protected double length;
8           protected double width;
            1 reference
9           public Rectangle(double l, double w)
10          {
11              length = l;
12              width = w;
13          }
14
            2 references
15          public double GetArea()
16          {
17              return length * width;
18          }
19
            1 reference
20          public void Display()
21          {
22              Console.WriteLine("Length: {0}", length);
23              Console.WriteLine("Width: {0}", width);
24              Console.WriteLine("Area: {0}", GetArea());
25          }
26      }//end class Rectangle
27
```

```
28      class Tabletop : Rectangle
29      {
30          private double cost;
            1 reference
31          public Tabletop(double l, double w) : base(l, w)
32          { }
            1 reference
33          public double GetCost()
34          {
35              double cost;
36              cost = GetArea() * 70;
37              return cost;
38          }
            1 reference
39          public void Display()
40          {
41              base.Display();
42              Console.WriteLine("Cost: {0}", GetCost());
43          }
44      }
        0 references
45      class ExecuteRectangle
46      {
            0 references
47          static void Main(string[] args)
48          {
49              Tabletop t = new Tabletop(4.5, 7.5);
50              t.Display();
51              Console.ReadLine();
52          }
53      }
54  }
```

Arv
Abstrakte klasser

Abstrakte klasser er kun designet til afledning. Det er ikke muligt at instantiere en abstrakt klasse, undtagen i forbindelse med at instantiere en klasse, der stammer fra den. Klasser, der ikke er abstrakte og i stedet kan instantieres direkte er konkrete klasser.

Abstrakte klasser repræsenterer abstrakte entities. Deres abstrakte medlemmer definerer hvad en genstand afledt af en abstrakt enhed skal indeholde, men de omfatter ikke implementeringen.

Abstrakte klasser har følgende egenskaber:

- En abstrakt klasse kan ikke instantieres.
- En abstrakt klasse kan indeholde abstrakte metoder og adgangsmetoder (accessors).
- Det er ikke muligt at ændre en abstrakt klasse med det sealed modifier, fordi de to modifers har modsatte betydninger. Sealed modifieren forhindrer en klasse i at blive arvet og den abstrakte modifier kræver at en klasse skal være nedarvet.
- En ikke-abstrakte klasse afledt fra en abstrakt klasse skal indeholde konkrete implementeringer af alle nedarvede abstrakte metoder og accessors.

Arv
Abstrakte klasser

Abstrakte klasser er kun designet til afledning. Det er ikke muligt at instantiere en abstrakt klasse, undtagen i forbindelse med at instantiere en klasse, der stammer fra den. Klasser, der ikke er abstrakte og i stedet kan instantieres direkte er konkrete klasser.

Abstrakte klasser repræsenterer abstrakte entities. Deres abstrakte medlemmer definerer hvad en genstand afledt af en abstrakt enhed skal indeholde, men de omfatter ikke implementeringen.

Abstrakte klasser har følgende egenskaber:

- En abstrakt klasse kan ikke instantieres.
- En abstrakt klasse kan indeholde abstrakte metoder og adgangsmetoder (accessors).
- Det er ikke muligt at ændre en abstrakt klasse med det sealed modifier, fordi de to modifers har modsatte betydninger. Sealed modifieren forhindrer en klasse i at blive arvet og den abstrakte modifier kræver at en klasse skal være nedarvet.
- En ikke-abstrakte klasse afledt fra en abstrakt klasse skal indeholde konkrete implementeringer af alle nedarvede abstrakte metoder og accessors.

Arv
Abstrakt

```
using System;
namespace RectangleApplication
{
    1 reference
    abstract class ShapesClass
    {
        2 references
        abstract public int Area();
    }
    3 references
    class Square : ShapesClass
    {
        int side = 0;

        1 reference
        public Square(int n)
        {
            side = n;
        }
        // Area method is required to avoid
        // a compile-time error.
        2 references
        public override int Area()
        {
            return side * side;
        }

        0 references
        static void Main()
        {
            Square sq = new Square(12);
            Console.WriteLine("Area of the square = {0}", sq.Area());
            Console.ReadKey();
        }

        1 reference
        interface I
        {
            1 reference
            void M();
        }
        0 references
        abstract class C : I
        {
            1 reference
            public abstract void M();
        }
    }
}
```

Arv
Polymorfisme

Ordet polymorfisme betyder at have mange former. I objektorienteret programmering paradigme, er polymorfi ofte udtrykt som 'en grænseflade, flere funktioner'.

Polymorfi kan være statisk eller dynamisk. I statisk polymorfi fastsættes reaktionen på en funktion ved kompilerings tidspunktet. I dynamisk polymorfi bliver den besluttet ved run-time.

Arv
Polymorfisme

Statisk polymofisme

Mekanismen med at koble en funktion med en genstand under kompileringen kaldes tidlig binding (early binding). Det kaldes også statisk binding. C# har to teknikker til at gennemføre statisk polymorfi. De er:

- Funktions overbelastning (function overloading)
- Operator overbelastning (operator overloading)

Arv
Polymorfisme

Statisk polymofisme- Funktions overbelastning

Man kan have flere definitioner for det samme funktions navn i samme scope. Definitionen af funktionen skal afvige fra hinanden gennem typer og/eller antallet af argumenter i argument listen.
Du kan ikke overbelaste funktionserklæringer som kun adskiller sig ved return type.

Det følgende eksempel bruger funktionen **print()** til at vise forskellige data typer.

Statisk polymofisme- Funktions overbelastning

```
1   using System;
2   namespace PolymorphismApplication
3   {
        2 references
4       class Printdata
5       {
            1 reference
6           void print(int i)
7           {
8               Console.WriteLine("Printing int: {0}", i);
9           }
10
            1 reference
11          void print(double f)
12          {
13              Console.WriteLine("Printing float: {0}", f);
14          }
15
            1 reference
16          void print(string s)
17          {
18              Console.WriteLine("Printing string: {0}", s);
19          }
            0 references
20          static void Main(string[] args)
21          {
22              Printdata p = new Printdata();
23
24              // Call print to print integer
25              p.print(5);
26
27              // Call print to print float
28              p.print(500.263);
29
30              // Call print to print string
31              p.print("Hello C++");
32              Console.ReadKey();
33          }
34      }
35  }
```

Arv
Polymorfisme

Statisk polymorfisme- Operator overbelastning

Du kan omdefinere eller overbelaste de fleste af de indbyggede operatorer til rådighed i C#.
Således kan man også bruge operatorer med brugerdefinerede typer.

Overbelastede operatører er funktioner med særlige navne: nøgleordet operator efterfulgt af symbolet for operatoren der defineres.

Ligesom andre funktioner har en overbelastet operatør en return type og en parameter liste.

```
public static Box operator+ (Box b, Box c)
{
  Box box = new Box();
  box.length = b.length + c.length;
  box.breadth = b.breadth + c.breadth;
  box.height = b.height + c.height;
  return box;
}
```

Ovenstående funktion implementerer additions operatoren (+) til en brugerdefineret klasse Box. Det tilføjer attributterne for to Box objekter og returnerer det resulterende Box objekt.

Arv
Polymorfisme

Dynamisk polymorfisme

Dynamisk polymorfi implementeres af abstrakte klasser og virtuelle funktioner.

Når du har en funktion defineret i en klasse, som du ønsker skal gennemføres i en nedarvet klasse(r), benytter man virtuelle funktioner. De virtuelle funktioner kan gennemføres forskelligt i de forskellige nedarvede klasser, og kald til disse funktioner vil blive afgjort ved runtime.

Dynamisk polymorfisme

```
1   using System;
2   namespace PolymorphismApplication
3   {
        4 references
4       class Shape
5       {
6           protected int width, height;
            2 references
7           public Shape(int a = 0, int b = 0)
8           {
9               width = a;
10              height = b;
11          }
            3 references
12          public virtual int area()
13          {
14              Console.WriteLine("Parent class area :");
15              return 0;
16          }
17      }
        3 references
18      class Rectangle : Shape
19      {
            1 reference
20          public Rectangle(int a = 0, int b = 0) : base(a, b)
```

```
21          {
22
23          }
            3 references
24          public override int area()
25          {
26              Console.WriteLine("Rectangle class area :");
27              return (width * height);
28          }
29      }
        3 references
30      class Triangle : Shape
31      {
            1 reference
32          public Triangle(int a = 0, int b = 0) : base(a, b)
33          {
34
35          }
            3 references
36          public override int area()
37          {
38              Console.WriteLine("Triangle class area :");
39              return (width * height / 2);
40          }
```

Arv
Polymorfisme

Dynamisk polymorfisme

```
41      }
        2 references
42      class Caller
43      {
            2 references
44          public void CallArea(Shape sh)
45          {
46              int a;
47              a = sh.area();
48              Console.WriteLine("Area: {0}", a);
49          }
50      }
        0 references
51      class Tester
52      {
53
            0 references
54          static void Main(string[] args)
55          {
56              Caller c = new Caller();
57              Rectangle r = new Rectangle(10, 7);
58              Triangle t = new Triangle(10, 5);
59              c.CallArea(r);
60              c.CallArea(t);
61              Console.ReadKey();
62          }
63      }
64  }
```

Arv

Opgave

Jeg står til rådighed til svar på spørgsmål og forklaring af fejl

Opgave

OPGAVE

- Begynd at tage prøverne inde på https://mva.microsoft.com/en-US/training-courses/c-fundamentals-for-absolute-beginners-16169 fra en ende af.
- I behøves ikke se videoerne, men brug dem endelig som reference / genopfriskning.

Kilder

Materiale benyttet i denne lektion
Noget af det er udover pensum-listen!

Kilder

Program eksempel

- http://www.makeuseof.com/tag/programming-game-unity-beginners-guide/#chapter-10

Operatorer

- https://msdn.microsoft.com/da-dk/library/ms173224(v=vs.100).aspx
- https://msdn.microsoft.com/da-dk/library/dn986595.aspx
- http://www.informit.com/articles/article.aspx?p=2421572
- https://weblogs.asp.net/sreejukg/the-null-conditional-operator
- https://youtu.be/_MaKC978z80

Kilder

Løkker

- https://www.tutorialspoint.com/csharp/csharp_while_loop.htm
- https://www.tutorialspoint.com/csharp/csharp_for_loop.htm
- https://www.tutorialspoint.com/csharp/csharp_do_while_loop.htm
- https://youtu.be/Yvc_iTy_BtI

Metoder

- https://www.tutorialspoint.com/csharp/csharp_methods.htm
- https://msdn.microsoft.com/da-dk/library/sfodf423.aspx
- https://msdn.microsoft.com/da-dk/library/c3ay4x3d.aspx
- https://www.dotnetperls.com/main
- https://msdn.microsoft.com/da-dk/library/acy3edy3.aspx
- https://youtu.be/QwygwfqOHsI

Kilder

Klasser

- https://www.tutorialspoint.com/csharp/csharp_classes.htm
- https://msdn.microsoft.com/en-us/library/x9afc042.aspx
- https://msdn.microsoft.com/en-us/library/ms173109.aspx
- https://msdn.microsoft.com/da-dk/library/wxh6fsc7.aspx
- https://www.dotnetperls.com/destructor
- https://msdn.microsoft.com/en-us/library/aa288471(v=vs.71).aspx
- https://youtu.be/s2hHjpZaSyI

Arv

- https://www.tutorialspoint.com/csharp/csharp_inheritance.htm
- https://msdn.microsoft.com/da-dk/library/sf985hc5.aspx
- https://msdn.microsoft.com/da-dk/library/ms173150.aspx

Kilder

Arv

- https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm
- https://en.wikipedia.org/wiki/Function_overloading
- https://www.tutorialspoint.com/csharp/csharp_operator_overloading.htm
- https://youtu.be/EiBCF7rYRtI