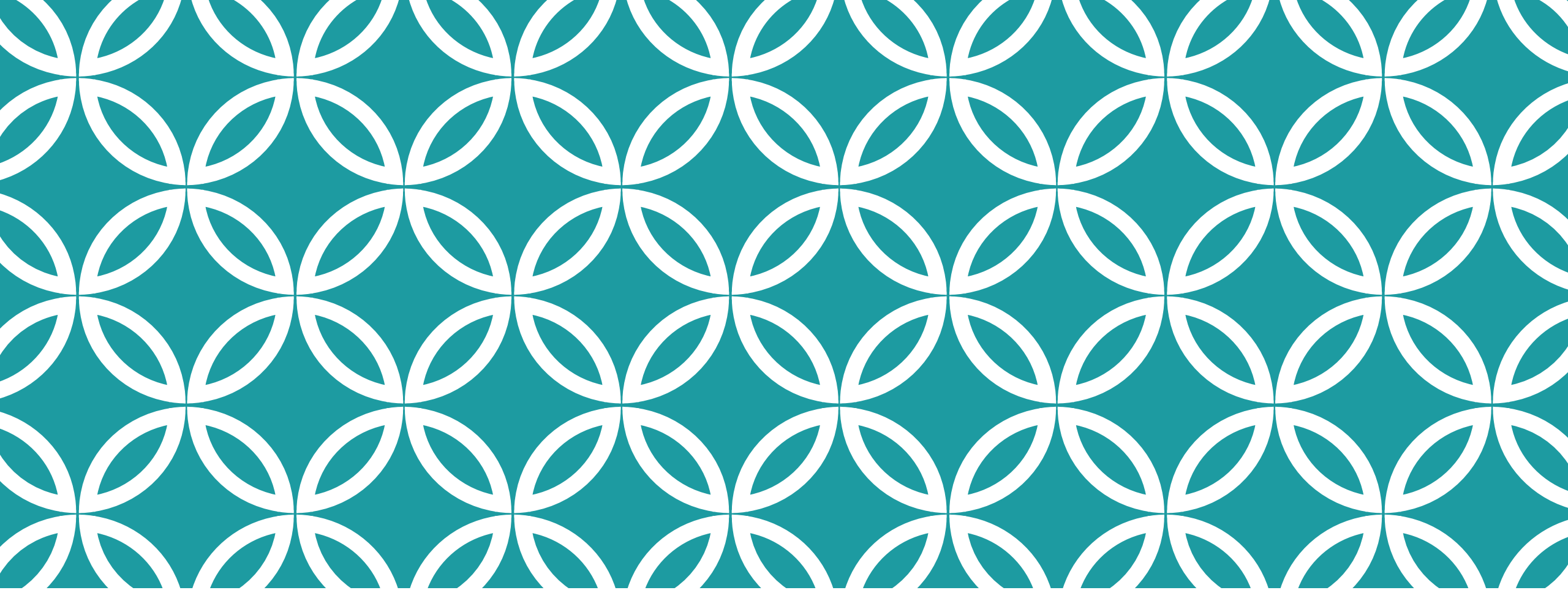
METODER
KLASSER

ARV

Grundlæggende programmering
Lektion 5



METODER

Sekvenser af kode om samme emne

REPETITION

- Række af statements der udfører en handling
- Mindst én metode der hedder main

Forskellen på en funktion og en metode

- Funktion er det proces-orienterede navn, metode er OO navnet. Da C # (og VB.NET) er objektorienterede sprog skal du bruge metode betegnelsen.
- I C # er er de indbyrdes udskiftelige (selvom metode er den korrekte betegnelse), fordi du ikke kan skrive en metode uden inkorporere det i en klasse.
  Hvis den var uafhængig af en klasse, så ville den være en funktion.
  Metoder er funktioner, der opererer gennem en udpeget klasse.

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 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.

```
1   using System;
2
3   // Using alias directive for a class.
4   using AliasToMyClass = NameSpace1.MyClass;
5
6   // Using alias directive for a generic class.
7   using UsingAlias = NameSpace2.MyClass<int>;
8
9   namespace NameSpace1
10  {
        3 references
11      public class MyClass
12      {
            1 reference
13          public override string ToString()
14          {
15              return "You are in NameSpace1.MyClass.";
16          }
17      }
18  }
19
20  namespace NameSpace2
21  {
        3 references
22      class MyClass<T>
23      {
            1 reference
24          public override string ToString()
25          {
26              return "You are in NameSpace2.MyClass.";
27          }
28      }
29  }
30
31  namespace NameSpace3
32  {
33      // Using directive:
34      using NameSpace1;
35      // Using directive:
36      using NameSpace2;
37
        0 references
38      class MainClass
39      {
            0 references
40          static void Main()
41          {
42              AliasToMyClass instance1 = new AliasToMyClass();
43              Console.WriteLine(instance1);
44
45              UsingAlias instance2 = new UsingAlias();
46              Console.WriteLine(instance2);
47
48              Console.ReadLine();
49          }
50      }
51  }
```

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.

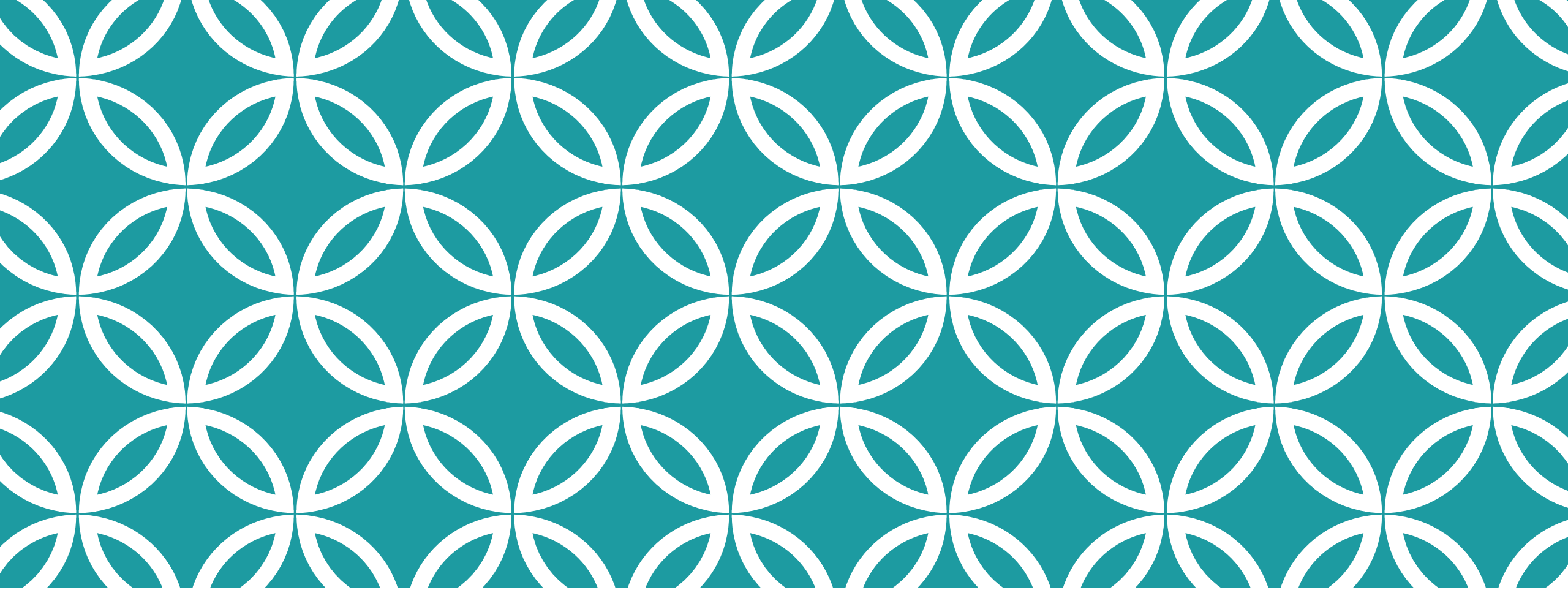
Main kan enten have en void eller int return type.

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

```
1   using System;
2
    0 references
3   class Program
4   {
        0 references
5       static void Main(string[] args)
6       {
7           if (args == null)
8           {
9               Console.WriteLine("args is null"); // Check for null array
10          }
11          else
12          {
13              Console.Write("args length is ");
14              Console.WriteLine(args.Length); // Write array length
15              for (int i = 0; i < args.Length; i++) // Loop through array
16              {
17                  string argument = args[i];
18                  Console.Write("args index ");
19                  Console.Write(i); // Write index
20                  Console.Write(" is [");
21                  Console.Write(argument); // Write string
22                  Console.WriteLine("]");
23              }
24          }
25          Console.ReadLine();
26      }
27  }
```

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).

KLASSER

```
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

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
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

```
using System;

4 references
class Example
{
    1 reference
    public Example()
    {
        Console.WriteLine("Constructor");
    }

    0 references
    ~Example()
    {
        Console.WriteLine("Destructor");
    }
}

0 references
class Program
{
    0 references
    static void Main()
    {
        Example x = new Example();
        Console.ReadKey();
    }
}
```

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.

KLASSER STRUCTS

```
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 afledte 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

```
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)

Operat0r 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

ARV

```
using System;
namespace PolymorphismApplication
{
    2 references
    class Printdata
    {
        1 reference
        void print(int i)
        {
            Console.WriteLine("Printing int: {0}", i);
        }

        1 reference
        void print(double f)
        {
            Console.WriteLine("Printing float: {0}", f);
        }

        1 reference
        void print(string s)
        {
            Console.WriteLine("Printing string: {0}", s);
        }

        0 references
        static void Main(string[] args)
        {
            Printdata p = new Printdata();

            // Call print to print integer
            p.print(5);

            // Call print to print float
            p.print(500.263);

            // Call print to print string
            p.print("Hello C++");
            Console.ReadKey();
        }
    }
}
```

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 operat0ren (+) 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

ARV

```
using System;
namespace PolymorphismApplication
{
    4 references
    class Shape
    {
        protected int width, height;
        2 references
        public Shape(int a = 0, int b = 0)
        {
            width = a;
            height = b;
        }
        3 references
        public virtual int area()
        {
            Console.WriteLine("Parent class area :");
            return 0;
        }
    }
    3 references
    class Rectangle : Shape
    {
        1 reference
        public Rectangle(int a = 0, int b = 0) : base(a, b)
        {

        }
        3 references
        public override int area()
        {
            Console.WriteLine("Rectangle class area :");
            return (width * height);
        }
    }
    3 references
    class Triangle : Shape
    {
        1 reference
        public Triangle(int a = 0, int b = 0) : base(a, b)
        {

        }
        3 references
        public override int area()
        {
            Console.WriteLine("Triangle class area :");
            return (width * height / 2);
        }
```

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

LEKTIE

Kig på dette til næste gang

LEKTIE

Se og løs opgaverne til https://mva.microsoft.com/en-US/training-courses/programming-in-c-jump-start-14254 - 03

Læs:

- https://www.tutorialspoint.com/csharp/csharp_classes.htm,
- https://www.tutorialspoint.com/csharp/csharp_inheritance.htm og
- https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm

KILDER

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

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

KILDER

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

https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm

https://en.wikipedia.org/wiki/Function_overloading

https://www.tutorialspoint.com/csharp/csharp_operator_overloading.htm