Praksis eksempel

Arv

Interfaces

Values

Exception handling

Validation

Generics

Delegates

Lambda expressions

Events

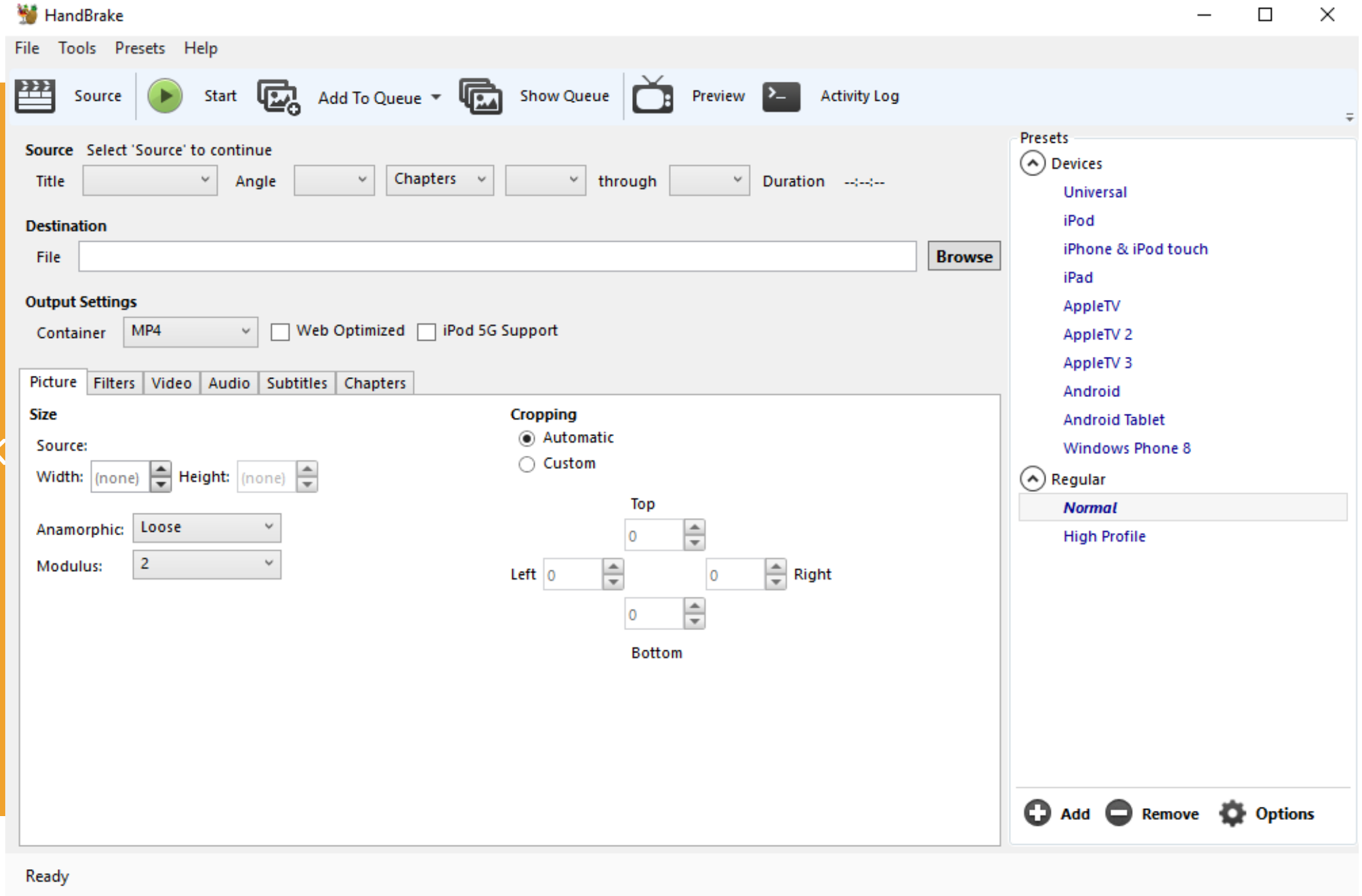
Grundlæggende programmering

Lektion 4

Praksis eksempel

Handbrake, et populært video konverterings program

Praksis ek


HandBrake
File Tools Presets Help
Source Start Add To Queue Show Queue Preview Activity Log
Source Select 'Source' to continue
Title Angle Chapters through Duration --:--:--
Destination
File Browse
Output Settings
Container MP4 Web Optimized iPod 5G Support
Picture Filters Video Audio Subtitles Chapters
Size
Source:
Width: (none) Height: (none)
Anamorphic: Loose
Modulus: 2
Cropping
Automatic
Custom
Top 0
Left 0 Right 0
Bottom 0
Presets
Devices
Universal
iPod
iPhone & iPod touch
iPad
AppleTV
AppleTV 2
AppleTV 3
Android
Android Tablet
Windows Phone 8
Regular
Normal
High Profile
Add Remove Options
Ready

Pra

```
1  namespace HandBrakeTools
2  {
3      using System;
4      using System.Diagnostics;
5      using System.IO;
6      using System.Security.Cryptography;
7
       0 references
8      public class Program
9      {
           0 references
10         static void Main(string[] args)
11         {
12             if (args.Length == 0)
13             {
14                 Console.WriteLine("Invalid Comand.  Either 'genkeys' or 'sign <filename>'");
15                 Console.Read();
16                 return;
17             }
18
19             string command = args[0];
20             string file = args.Length > 1 ? args[1] : null;
21             switch (command)
22             {
23                 case "genkeys":
24                     GenKeyFiles();
25                     return;
26                 case "sign":
27                     string hash = SignDownload(file);
28                     Console.WriteLine(VerifyDownload(hash, file) ? "Passed Verification Test" : "Failed Verification Test");
29                     Console.WriteLine("Hash: " + hash);
30                     File.WriteAllText("file.hash", hash);
31                     return;
32             }
```

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

Interfaces

Hvordan noget udføres

Interfaces

- Hvor klasser definerer *hvordan* noget skal udføres sørger interfaces for *hvad* der skal udføres.
- Interfaces definerer properties, methods og events, der er medlemmer af interfacet.
- Interfaces indeholder kun deklarationen af medlemmerne. Det er den afledte klasses ansvar at definere medlemmerne.
- Interfacet hjælper ofte med at give en standard struktur, som de afledte klasser vil følge.
- Abstrakte klasser har til en vis grad samme formål, men de bliver primært brugt når der kun skal erklæres få metoder af base-klassen og de underliggende klasser implementerer funktionerne.

Interfaces

- Interfaces erklæres med interface nøgleordet.
  - Det svarer til en klasse erklæring.
- Interface-udsagn er offentlige som standard.
- Følgende er et eksempel på et interface erklæring:

```
public interface ITransactions
{
    // interface members
    void showTransaction();
    double getAmount();
}
```

Inter

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

namespace InterfaceApplication
{
    1 reference
    public interface ITransactions
    {
        // interface members
        3 references
        void showTransaction();
        2 references
        double getAmount();
    }

    6 references
    public class Transaction : ITransactions
    {
        private string tCode;
        private string date;
        private double amount;
        0 references
        public Transaction()
        {
            tCode = " ";
            date = " ";
            amount = 0.0;
        }

        2 references
        public Transaction(string c, string d, double a)
        {
            tCode = c;
            date = d;
            amount = a;
        }

        2 references
        public double getAmount()
        {
            return amount;
        }

        3 references
        public void showTransaction()
        {
            Console.WriteLine("Transaction: {0}", tCode);
            Console.WriteLine("Date: {0}", date);
            Console.WriteLine("Amount: {0}", getAmount());
        }
    }
    0 references
    class Tester
    {
        0 references
        static void Main(string[] args)
        {
            Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
            Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
            t1.showTransaction();
            t2.showTransaction();
            Console.ReadKey();
        }
    }
}
```

Interfaces

Value types

Structs, boxing og enums

Value types

- En value type indeholder direkte sin værdi.
  - Navnet forbindes direkte med pladsen i hukommelsen, hvor data er gemt.
  - Hvis en anden variabel får værdien fra den første variabel laves der en kopi af værdien på den ny variabels plads, da to variabler aldrig kan referere til den samme plads.
    - Undtagelsen er hvis en eller begge har out eller ref parametrene, men således er de også defineret som alias for en anden variabel.

Value types
Struct

- En structure er en værdi data type.
- Den lader en enkelt variable indeholde relateret data i forskellige data typer.
- Man benytter *struct* keyword til at lave en structure.
  - Et struct statement definerer en ny data type, der har mere end et medlem.

```
struct Books
{
    public string title;
    public string author;
    public string subject;
    public int book_id;
};
```

```
using System;
2 references
struct Books
{
    public string title;
    public string author;
    public string subject;
    public int book_id;
};

0 references
public class testStructure
{
    0 references
    public static void Main(string[] args)
    {

        Books Book1;   /* Declare Book1 of type Book */
        Books Book2;   /* Declare Book2 of type Book */

        /* book 1 specification */
        Book1.title = "C Programming";
        Book1.author = "Nuha Ali";
        Book1.subject = "C Programming Tutorial";
        Book1.book_id = 6495407;

        /* book 2 specification */
        Book2.title = "Telecom Billing";
        Book2.author = "Zara Ali";
        Book2.subject = "Telecom Billing Tutorial";
        Book2.book_id = 6495700;

        /* print Book1 info */
        Console.WriteLine("Book 1 title : {0}", Book1.title);
        Console.WriteLine("Book 1 author : {0}", Book1.author);
        Console.WriteLine("Book 1 subject : {0}", Book1.subject);
        Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

        /* print Book2 info */
        Console.WriteLine("Book 2 title : {0}", Book2.title);
        Console.WriteLine("Book 2 author : {0}", Book2.author);
        Console.WriteLine("Book 2 subject : {0}", Book2.subject);
        Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

        Console.ReadKey();

    }
}
```

Value types
Struct

Egenskaber for structures

- Structures kan have methods, fields, indexers, properties, operator methods og events.
- Structures kan have definerede constructors, men ikke destructors.
- Modsat classes kan structures íkke arve fra andre structures eller classes.
- Structures kan ikke bruges som en base for andre structures eller classes.
- En structure kan implementere et eller flere interfaces.
- Structure members kan ikke specifceres som abstract, virtual eller protected.
- Når man opretter et struct objekt ved hjælp af New operator bliver det oprettet og den passende constructor kaldes. I modsætning til klasser, kan structs instantieres uden at bruge New operator.
- Hvis der ikke anvendes New operator forbliver felterne ikke-tildelt og objektet kan ikke anvendes før alle felter er initialiseret.

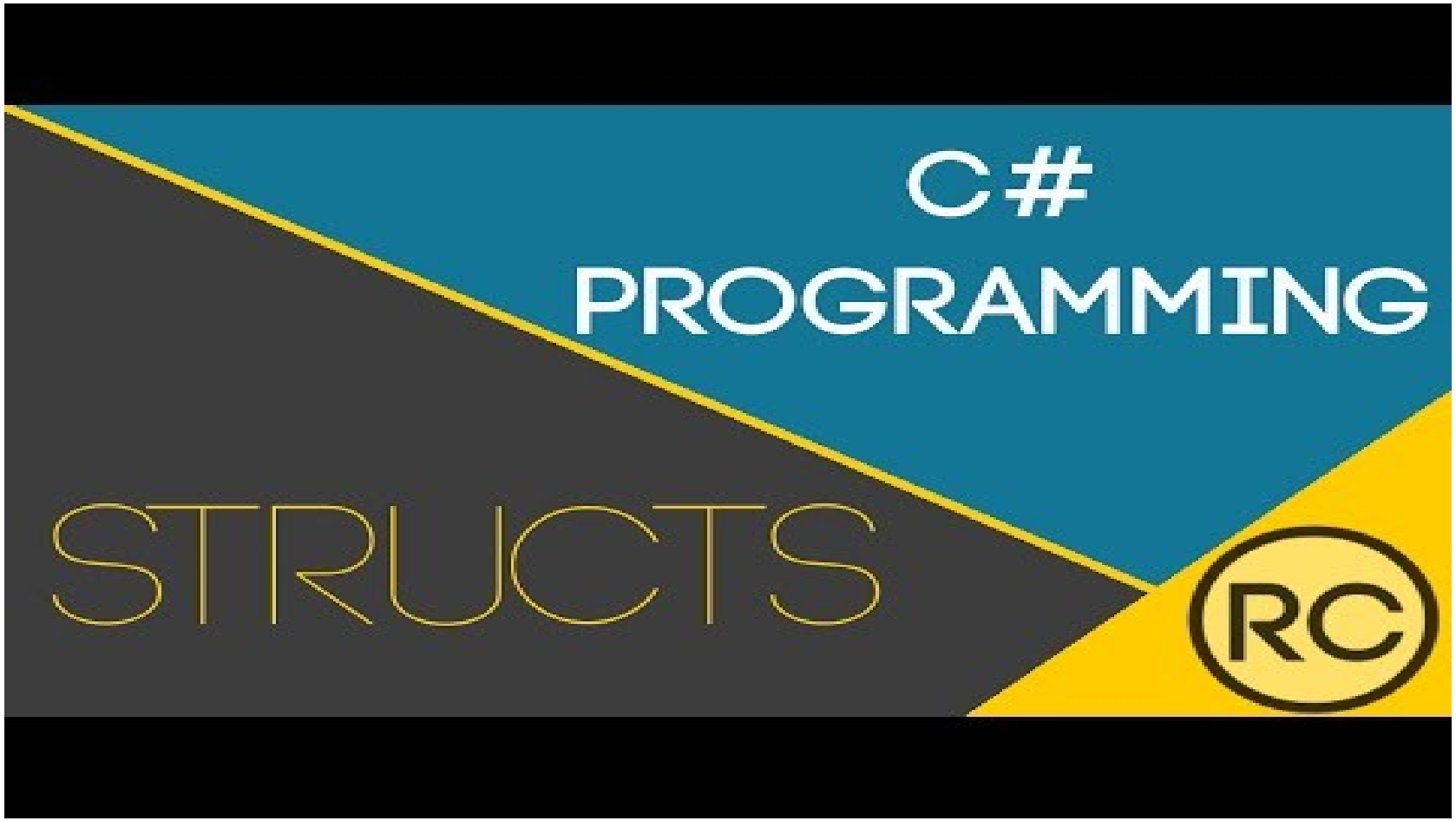
Value types Struct

Classes vs. structures

Classes og structures har følgende grundlæggende forskelle:

- Classes er referencetyper og structs er værdi typer.
- Structures understøtter ikke nedarvning.
- Structures kan ikke have default constructor.
  - I C# 6.0 har alle værdi typer en default constructor .

Value types
Struct



Value types Boxing

- Boxing og unboxing lader værdi typer blive behandlet som objekter.
- Boxing af en værdi typen pakker den ind i en instans af Object henvisningens typen.
- Unboxing udtrækker værditypen fra objektet.
- Her tages integer variablen i som bliver boxed og tildelt objektet o

```
int i = 123;
// The following line boxes i.
object o = i;
```

- Objektet o kan så unboxes og gives til integer variablen i

```
o = 123;
i = (int)o;  // unboxing
```

Value types
Enums

- En enumeration er et sæt navngivne integer konstanter.
- En enumerated type declares med enum keyword.
- C# enumerations er en værdi data type. Enumeration indeholder altså dens egne værdier og kan ikke arve eller videresende arv.
- Enum declares således

```
enum <enum_name>
{
   enumeration list
};
```

  - *enum_name* angiver enumeration typens navn.
  - *enumeration list* er en komma-separeret liste af identifiers.
- Hver af symbolerne i enumeration listen står for en heltalsværdi, en større end symbolet der er angivet foran. Som standard er værdien af det første enumeration symbol o.

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

Value types
Enums

```
1   using System;
2   namespace EnumApplication
3   {
        0 references
4       class EnumProgram
5       {
            2 references
6           enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
7
            0 references
8           static void Main(string[] args)
9           {
10              int WeekdayStart = (int)Days.Mon;
11              int WeekdayEnd = (int)Days.Fri;
12              Console.WriteLine("Monday: {0}", WeekdayStart);
13              Console.WriteLine("Friday: {0}", WeekdayEnd);
14              Console.ReadKey();
15          }
16      }
17  }
```

Exception handling

Problemer opstår så der må gøres noget

Exception handling

- En undtagelse (exception) er et problem, der opstår under udførelsen af et program.
- I C# er en undtagelse er en reaktion på en usædvanlig hændelse, der opstår, mens et program kører, f.eks. et forsøg på at dividere med nul.
- Exceptions sørger for en måde at transfer kontrol fra en del af et program til en anden.
- C# exception handling bygger på fire keywords: **try**, **catch**, **finally** og **throw**.
  - **try**: En try blok identificerer en blok af kode, for hvilken særlige undtagelser er aktiveret. Det efterfølges af en eller flere catch blokke.
  - **catch**: Et program fanger en undtagelse med en exception handler det sted i programmet, hvor du ønsker at håndtere problemet. Catch keyword angiver fangst af en undtagelse.
  - **finally**: Finally blokken anvendes til at udføre et givet sæt udsagn, om en undtagelse er thrown eller ikke thrown. For eksempel, hvis du åbner en fil, skal det være lukket om en undtagelse hæves eller ej.
  - **throw**: Et program kaster en undtagelse, når et problem dukker op. Dette gøres ved hjælp af throw keyword.

Exception handling Syntaks

Syntaks

- Hvis en blok udløser en exception fanger en metode en exception gennem en kombination af try og catch keywords.
- En try/catch blok placers omrking koden, der kan generere en exception. Kode i en try/catch omtales som protected (beskyttet) kode.
- Du kan liste flere catch statements til at fange forskellige typer af undtagelser i tilfælde af at en blok trigger mere end en enkelt undtagelse i forskellige situationer.

```
try
{
   // statements causing exception
}
catch( ExceptionName e1 )
{
   // error handling code
}
catch( ExceptionName e2 )
{
   // error handling code
}
catch( ExceptionName eN )
{
   // error handling code
}
finally
{
   // statements to be executed
}
```

Exception handling Klasser

Exception klasser i C#

- C# exceptions repræsenteres af classes.
- Exception classes i C# er primært direkte eller indirekte afledt af **System.Exception** klassen.
  - Nogle exception classes afledt af System.Exception klassen er **System.ApplicationException** og **System.SystemException**.
  - **System.ApplicationException** klassen understøtter exceptions genereret af application programmer. Derfor bør exceptions, som defineres af programmørerne, stamme fra denne klasse.
  - **System.SystemException** klassen er base klassen for alle prædefinerede system exceptions.
  - Nogle af de prædefinerede exception klasser afledt af System.SystemException klassen kan ses i skemaet på næste slide.

Exception handling Klasser

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

Exception handling
Håndtering

Håndtering af exceptions

- C # giver en struktureret løsning på håndtering af exceptions i form af try og catch blokke. Ved hjælp af disse blokke bliver kerne-program statements adskilt fra fejl håndterings statements.
- Disse fejl håndterings blokke implementers med **try**, **catch**, og **finally** keywords.

```
using System;
namespace ErrorHandlingApplication
{
    3 references
    class DivNumbers
    {
        int result;
        1 reference
        DivNumbers()
        {
            result = 0;
        }
        1 reference
        public void division(int num1, int num2)
        {
            try
            {
                result = num1 / num2;
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("Exception caught: {0}", e);
            }
            finally
            {
                Console.WriteLine("Result: {0}", result);
            }
        }
        0 references
        static void Main(string[] args)
        {
            DivNumbers d = new DivNumbers();
            d.division(25, 0);
            Console.ReadKey();
        }
    }
}
```

Brugerdefinerede exceptions

- Man kan også definer ens egne exceptions.
- Bruger definerede exception klasser er afledt af **Exception** klassen.

```
using System;
namespace UserDefinedException
{
    0 references
    class TestTemperature
    {
        0 references
        static void Main(string[] args)
        {
            Temperature temp = new Temperature();
            try
            {
                temp.showTemp();
            }
            catch (TempIsZeroException e)
            {
                Console.WriteLine("TempIsZeroException: {0}", e.Message);
            }
            Console.ReadKey();
        }
    }
}

3 references
public class TempIsZeroException : Exception
{
    1 reference
    public TempIsZeroException(string message) : base(message)
    {
    }
}

2 references
public class Temperature
{
    int temperature = 0;
    1 reference
    public void showTemp()
    {
        if (temperature == 0)
        {
            throw (new TempIsZeroException("Zero Temperature found"));
        }
        else
        {
            Console.WriteLine("Temperature: {0}", temperature);
        }
    }
}
```

Exception handling

Validation

Tjek af programmet

Validation

- Datavalidering er test-værdier introduceret til en app (via en tjeneste, fil eller indtastning af data) mod forventede værdier og ranges.
- Man gør det for at:
  - Undgå overflow.
  - Undgå forkerte resultater.
  - Undgå uønskede bivirkninger
  - Vejlede systemer eller brugere.
  - Forebyg sikkerheds indtrængen.
- Compileren validerer, at objekttypen er korrekt, den validerer ikke objektets værdi.

Generics

Lad programmet bestemme når det skal bruges

Generics

- Generics gør det muligt at forsinke specifikation af datatypen for program elementer i en klasse eller en metode, indtil de faktisk bliver brugt i programmet.
  - Med generics kan man altså skrive en klasse eller metode, der kan arbejde med alle datatyper.
- Man skriver specifikationerne for klassen eller den metode, med alternative parametre for datatyper.
  - Når compileren møder en constructor for klassen eller et funktionskald for metoden, genererer den kode til at håndtere den specifikke datatype.

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace GenericApplication
5  {
       5 references
6      public class MyGenericArray<T>
7      {
8          private T[] array;
               2 references
9          public MyGenericArray(int size)
10         {
11             array = new T[size + 1];
12         }
13
               2 references
14         public T getItem(int index)
15         {
16             return array[index];
17         }
18
               2 references
19         public void setItem(int index, T value)
20         {
21             array[index] = value;
22         }
23     }
24
           0 references
25     class Tester
26     {
               0 references
27         static void Main(string[] args)
28         {
29
30             //declaring an int array
31             MyGenericArray<int> intArray = new MyGenericArray<int>(5);
32
33             //setting values
34             for (int c = 0; c < 5; c++)
35             {
36                 intArray.setItem(c, c * 5);
37             }
38
39             //retrieving the values
40             for (int c = 0; c < 5; c++)
41             {
42                 Console.Write(intArray.getItem(c) + " ");
43             }
44
```

Fortsættes på næste slide med linje 45-66

Generics

```
45              Console.WriteLine();
46
47              //declaring a character array
48              MyGenericArray<char> charArray = new MyGenericArray<char>(5);
49
50              //setting values
51              for (int c = 0; c < 5; c++)
52              {
53                  charArray.setItem(c, (char)(c + 97));
54              }
55
56              //retrieving the values
57              for (int c = 0; c < 5; c++)
58              {
59                  Console.Write(charArray.getItem(c) + " ");
60              }
61              Console.WriteLine();
62
63              Console.ReadKey();
64          }
65      }
66  }
```

Forsat fra linje 1-44 på sidste slide

Generics Egenskaber

Generics egenskaber

Generics kan forbedre ens programmer på følgende måder:

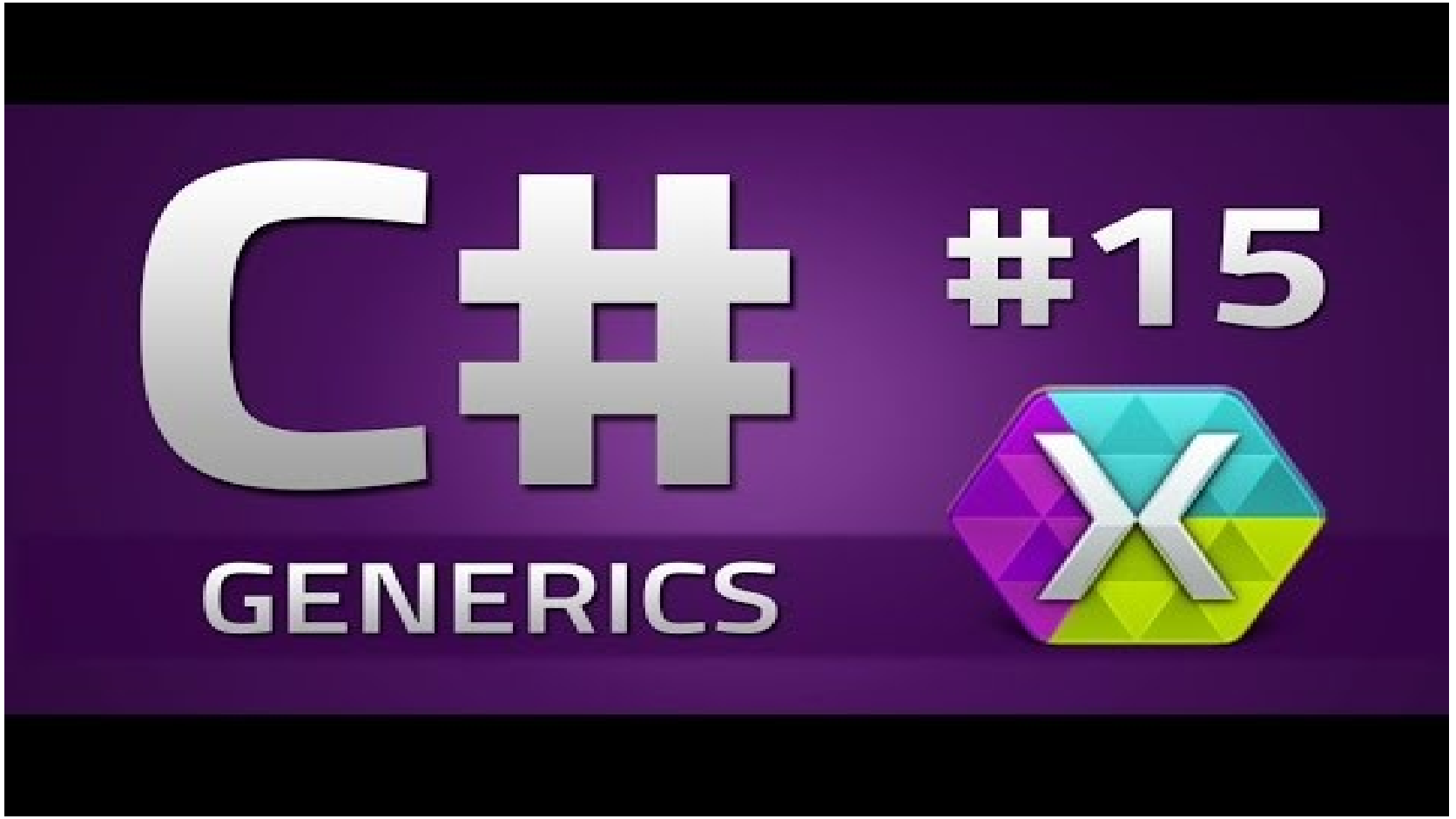
- De hjælper med at maksimere genbrug af kode, type sikkerhed og ydeevne.
- Du kan oprette generic collection klasser.
  - .NET Framework klasse bibliotek indeholder en række nye generic collection classes i System.Collections.Generic namespace. Man kan bruge disse generic collection classes i stedet for collection classes i System.Collections namespace.
- Man kan lave egne generic interfaces, classes, methods, events og delegates.
- Man kan lave generic classes tvunget til at give adgang til metoder for bestemte datatyper.
- Man kan få information om de typer, der anvendes i en generic datatype ved run-time ved hjælp af reflection.

Generic methods

- Man kan declare en generic method med et type parameter.

```
using System;
using System.Collections.Generic;

namespace GenericMethodAppl
{
    0 references
    class Program
    {
        2 references
        static void Swap<T>(ref T lhs, ref T rhs)
        {
            T temp;
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }
        0 references
        static void Main(string[] args)
        {
            int a, b;
            char c, d;
            a = 10;
            b = 20;
            c = 'I';
            d = 'V';

            //display values before swap:
            Console.WriteLine("Int values before calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values before calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            //call swap
            Swap<int>(ref a, ref b);
            Swap<char>(ref c, ref d);

            //display values after swap:
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            Console.ReadKey();
        }
    }
}
```

Validation



Delegates

Referencer til metoder

Delegates

- En delegate er en reference type variabel, der indeholder en reference til en metode. Referencen kan ændres ved runtime.
- Delegates er specielt brugt til at gennemføre events og call-back metoder.
- Alle delegates implicit afledt af System.Delegate klassen.
- Delegate deklarationen bestemmer metoderne der kan henvises til af delegate, da den kun kan referere til en metode der har den same signatur.

  public delegate int MyDelegate (string s);

  kan således kun bruges til at referere metoder der har en enkelt *string* parameter og returnerer en *int* type variabel.
- Den generelle declaration er således:

  delegate <return type> <delegate-name> <parameter list>

Delegates Instantiering

Instantiering af delegates

- Når en delegate type erklæres skal et delegate objekt oprettes med den new keyword og være forbundet med en særlig metode.
- Når en delegate laves angives argumentet videre til nye udtryk på en sådan måde at det ligner et metode kald, men uden argumenterne til metoden.

```
1   using System;
2
3   delegate int NumberChanger(int n);
4   namespace DelegateAppl
5   {
        0 references
6       class TestDelegate
7       {
8           static int num = 10;
            1 reference
9           public static int AddNum(int p)
10          {
11              num += p;
12              return num;
13          }
14
            1 reference
15          public static int MultNum(int q)
16          {
17              num *= q;
18              return num;
19          }
            2 references
20          public static int getNum()
21          {
22              return num;
23          }
24
            0 references
25          static void Main(string[] args)
26          {
27              //create delegate instances
28              NumberChanger nc1 = new NumberChanger(AddNum);
29              NumberChanger nc2 = new NumberChanger(MultNum);
30
31              //calling the methods using the delegate objects
32              nc1(25);
33              Console.WriteLine("Value of Num: {0}", getNum());
34              nc2(5);
35              Console.WriteLine("Value of Num: {0}", getNum());
36              Console.ReadKey();
37          }
38      }
39  }
```

Delegates Multicasting

Multicasting en delegate

- Delegate objekter kan sammensættes ved hjælp af "+" operator.
- En sammensat delegeret kalder de to delegates den blev sammensat af.
- Kun delegates af samme type kan være sammensat.
- "-" operatoren kan bruges til at fjerne en komponent delegate fra en sammensat delegate.
- Ved hjælp af denne egenskab kan man oprette en invocation (påkalde) liste over metoder, der vil blive kaldt, når en delegate kaldes. Dette kaldes multicasting af en delegeret.

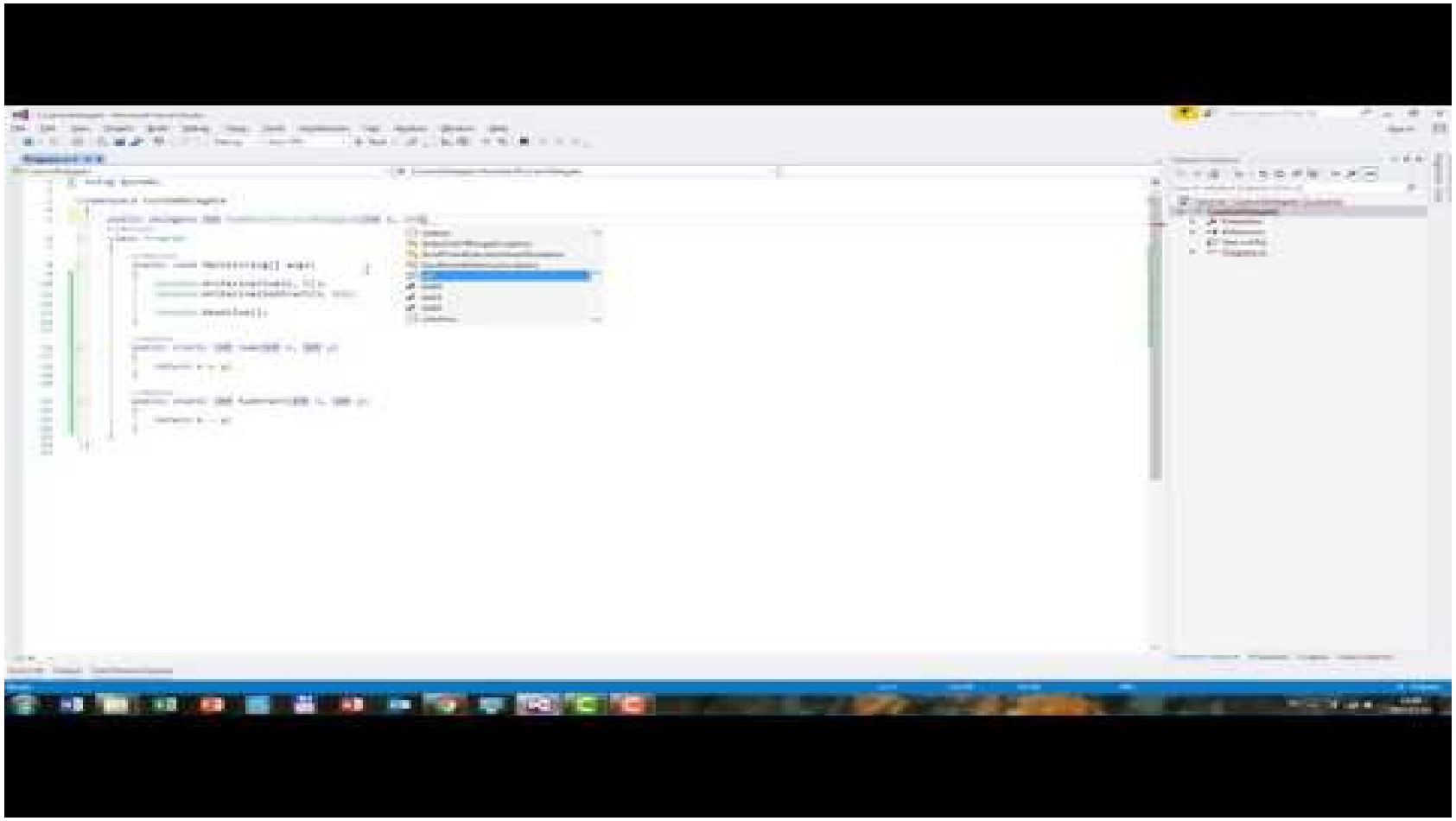
```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
    0 references
    class TestDelegate
    {
        static int num = 10;
        1 reference
        public static int AddNum(int p)
        {
            num += p;
            return num;
        }

        1 reference
        public static int MultNum(int q)
        {
            num *= q;
            return num;
        }

        1 reference
        public static int getNum()
        {
            return num;
        }

        0 references
        static void Main(string[] args)
        {
            //create delegate instances
            NumberChanger nc;
            NumberChanger nc1 = new NumberChanger(AddNum);
            NumberChanger nc2 = new NumberChanger(MultNum);
            nc = nc1;
            nc += nc2;

            //calling multicast
            nc(5);
            Console.WriteLine("Value of Num: {0}", getNum());
            Console.ReadKey();
        }
    }
}
```

Delegates Brug

Multicasting en delegate

- Delegate objekter kan sammensættes ved hjælp af "+" operator.
- En sammensat delegeret kalder de to delegates den blev sammensat af.
- Kun delegates af samme type kan være sammensat.
- "-" operatoren kan bruges til at fjerne en komponent delegate fra en sammensat delegate.
- Ved hjælp af denne egenskab kan man oprette en invocation (påkalde) liste over metoder, der vil blive kaldt, når en delegate kaldes. Dette kaldes multicasting af en delegeret.

Delegates



Lambda expressions

En anonym funktion

Lambda expressions

- Et lambda udtryk er en anonym funktion, som kan bruges til at oprette delegates eller udtryk træ typer.
- Ved at bruge lambda udtryk kan man skrive lokale funktioner, der kan overføres som argumenter eller returneres som værdien af funktionskald.
- Lambda udtryk er særligt nyttigt for at skrive LINQ query udtryk.
- For at oprette et lambda udtryk angiver man inputparametre (hvis nogen) på venstre side af lambda operatoren =>, og man sætter udtryk eller udsagn blokken på den anden side
  - Lambda udtrykket `x => x * x` angiver en parameter, x og giver den værdien af x potens. Du kan tildele dette udtryk til en delegeret type.

Lambda expressions

```
1   using System;
2   using System.Collections.Generic;
3
        0 references
4   class Program
5   {
            0 references
6       static void Main()
7       {
8           List<int> elements = new List<int>() { 10, 20, 31, 40 };
9           // ... Find index of first odd element.
10          int oddIndex = elements.FindIndex(x => x % 2 != 0);
11          Console.WriteLine(oddIndex);
12          Console.ReadKey();
13      }
14  }
```

Lambda expressions

Events

Referencer til metoder

Events

- Events er brugerhandlinger såsom tastetryk, klik, musebevægelser, etc., eller nogle hændelser såsom system genererede meddelelser.
- Applikationer er nødt til at reagere på begivenheder, når de opstår. For eksempel interrupts.
- Events anvendes til inter-proces kommunikation.

Events

- Begivenhederne erklæres og udføres i en klasse og forbindes med event handlers der bruger delegates inden for samme klasse eller en anden klasse.
- Klassen, der indeholder eventet, bruges til at udgive eventet.
  - Dette kaldes udgiver (publisher) klassen.
  - Andre klasser, der accepterer eventet, kaldes abonnent (subscriber) klassen.
- Derfor kaldes det at egivenheder bruger en udgiver-abonnent model.

Events

- En publisher er et objekt, som indeholder definitionen af eventet og den delegerede.
  - Event-delegate associationen er også defineret i dette objekt.
  - Et publisher klasse objekt invoker begivenheden og den er anmeldt til andre objekter.
- En subscriber er et objekt der accepterer eventet og leverer en event handler. Delegate i udgiver klassen invoket metoden (event handler) i abonnent klassen.

Events

- For at declar en event I en klasse skal der først laves en delegate type for eventet.

```
public delegate string MyDel(string str);
```

- Derefter declares eventet selv med event keyword.

```
event MyDel MyEvent;
```

Events

```
using System;
namespace SampleApp
{
    public delegate string MyDel(string str);

    3 references
    class EventProgram
    {
        event MyDel MyEvent;

        1 reference
        public EventProgram()
        {
            this.MyEvent += new MyDel(this.WelcomeUser);
        }

        1 reference
        public string WelcomeUser(string username)
        {
            return "Welcome " + username;
        }

        0 references
        static void Main(string[] args)
        {
            EventProgram obj1 = new EventProgram();
            string result = obj1.MyEvent("Student");
            Console.WriteLine(result);
            Console.ReadKey();
        }

    }
}
```

Events

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

Praksis eksempel

- https://en.wikipedia.org/wiki/HandBrake
- https://handbrake.fr/
- https://github.com/HandBrake/HandBrake

Arv

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

Kilder

Interfaces

- https://www.tutorialspoint.com/csharp/csharp_interfaces.htm
- https://youtu.be/IQpss9YAc4g

Value types

- https://www.tutorialspoint.com/csharp/csharp_data_types.htm
- https://www.tutorialspoint.com/csharp/csharp_struct.htm
- https://msdn.microsoft.com/da-dk/library/s1ax56ch.aspx
- https://msdn.microsoft.com/en-us/library/aa288471(v=vs.71).aspx
- https://youtu.be/rX_oqrHd8RM
- https://msdn.microsoft.com/en-us/library/yz2be5wk.aspx

Kilder

Value types

- https://msdn.microsoft.com/en-us/library/yz2be5wk(v=vs.80).aspx
- https://www.tutorialspoint.com/csharp/csharp_enums.htm
- https://msdn.microsoft.com/da-dk/library/sbbt4032.aspx
- https://www.dotnetperls.com/enum
- http://csharp.net-informations.com/statements/enum.htm

Exception handling

- https://www.tutorialspoint.com/csharp/csharp_exception_handling.htm
- https://youtu.be/gOtZSaLPu-E
- https://youtu.be/El8rlaE3LI8

Kilder

Validation

- https://mva.microsoft.com/en-US/training-courses/programming-in-c-jump-start-14254?l=KkOpp1SfB_8200115888

Generics

- https://www.tutorialspoint.com/csharp/csharp_generics.htm
- https://youtu.be/ZrjCGoFu5Ew

Delegates

- https://www.tutorialspoint.com/csharp/csharp_delegates.htm
- https://youtu.be/mlzYgppHXXM

Kilder

Lambda expressions

- https://msdn.microsoft.com/en-us/library/bb397687.aspx
- https://msdn.microsoft.com/en-us/library/bb397675.aspx
- https://www.dotnetperls.com/lambda
- https://youtu.be/xev-kNmz_ao

Events

- https://www.tutorialspoint.com/csharp/csharp_events.htm
- https://youtu.be/jQgwEsJISyo