

Delegates og lambda-udtryk

# Delegates og lambdaudtryk

- Metoder som Data
- “Funktionspointere” ~

# Motiverende eksempel

- Produkt-katalog

```
class Product{  
    string Manufacturer { get; set; }  
    string Name          { get; set; }  
    decimal Price        { get; set; }  
}
```

```
class Products  
{  
    private List<Product> _products;  
    public List<Product> FindProductsBy...(...) {...}  
    /* forskjellige søgefunktioner*/  
}
```

Price (>,<,<=,>=,==,!=)

Name (startswith, endswith)

Manufacturer osv...

```

class Product{ Properties: string Manufacturer, string Name og decimal Price }

public class Products {
    private List<Product> _products;

    public List<Product> FindProductsByManufacturer(string name) {
        List<Product> result = new List<Product>();

        foreach (Product p in _products)
            if (p.Manufacturer == name)
                result.Add(p);
        return result;
    }

    public List<Product> FindProductsWithMinimumPrice(decimal price) {
        List<Product> result = new List<Product>();

        foreach (Product p in _products)
            if (p.Price > price)
                result.Add(p);
        return result;
    }

    public List<Product> FindProductsWithMaximumPrice(decimal price) {...}

    public List<Product> FindProductsBy...(...) {...}
}

```

# En mere fleksibel løsning: Filter/Strategy-pattern

```
public class Products
{
    private List<Product> _products;

    public List<Product> FindProducts(IFilter<Product> filter)
    {
        List<Product> result = new List<Product>();

        foreach (Product p in _products)
            if (filter.Predicate(p))
                result.Add(p);
        return result;
    }
}
```

```
interface IFilter<T>
{
    bool Predicate(T p);
}
```

//Klasser der implementerer Ifilter<Product> til at filtrere produkter →

```

class ManufacturerEqualsFilter : IFilter<Product> {
    public string Manufacturer { get; set; }
    public bool Predicate(Product p) {
        return this.Manufacturer == p.Manufacturer;
    }
}

class PriceBelowFilter : IFilter<Product> {
    public decimal Price;
    public bool Predicate(Product p) {
        return this.Price < p.Price;
    }
}

class ManufacturerEqualsAndPriceBelowFilter : IFilter<Product>
{
    public string Manufacturer { get; set; }
    public decimal Price { get; set; }

    public bool Predicate(Product p) {
        return
            this.Manufacturer == p.Manufacturer &&
            this.Price < p.Price;
    }
}

// Flere filtre - lige så mange som hjertet begærer...

```

```
static void Main(string[] args)
{
    Products products = new Products();
    //vi forestiller os at der er tilføjet 10.000 produkter...

    var sonyFilter = new ManufacturerEqualsFilter() { Manufacturer = "Sony" };
    List<Product> sonyProducts = products.FindProducts(sonyFilter);
}
```

- Fordel: Flexibilitet + mulighed for at udskifte strategi/metode på runtime
- Ulempe: Bøvlet at lave interface samt subklasser

```
static void Main(string[] args)
{
    Products products = new Products();
    //vi forestiller os at der er tilføjet 10.000 produkter...

    var sonyFilter = new ManufacturerEqualsFilter() { Manufacturer = "Sony" };
    List<Product> sonyProducts = products.FindProducts(sonyFilter);

    var samsungFilter = new ManufacturerEqualsFilter() { Manufacturer = "Samsung" };
    List<Product> samsungProducts = products.FindProducts(samsungFilter);

}
```

- Fordel: Flexibilitet + mulighed for at udskifte strategi/metode på runtime
- Ulempe: Bøvlet at lave interface samt subklasser



```

static void Main(string[] args)
{
    Products products = new Products();
    //vi forestiller os at der er tilføjet 10.000 produkter...

    var sonyFilter = new ManufacturerEqualsFilter() { Manufacturer = "Sony" };
    List<Product> sonyProducts = products.FindProducts(sonyFilter);

    var samsungFilter = new ManufacturerEqualsFilter() { Manufacturer = "Samsung" };
    List<Product> samsungProducts = products.FindProducts(samsungFilter);

    var sony1000Filter = new ManufacturerEqualsAndPriceBelowFilter()
    {
        Manufacturer = "Sony",
        Price = 1000
    };
    List<Product> cheapSonyProducts = products.FindProducts(sony1000Filter);
}

```

- Fordel: Flexibilitet + mulighed for at udskifte strategi/metode på runtime
- Ulempe: Bøvlet at lave interface samt subklasser

# Delegates

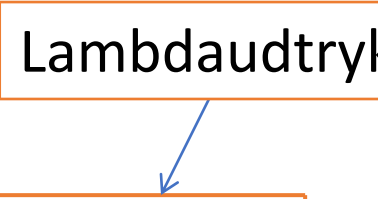
- **Strategi-mønstret** kan være bøvlet at bruge i simple situationer
  - Man skal implementere en ny klasse for hvert filter
  - Delegates tillader os at bruge en metode som parameter!
- Delegates

```
public delegate double IntToDoubleDelegate(int param);  
public static double FooMethod(int intParam){  
    return 0.5 * (double)intParam;  
}  
IntToDoubleDelegate bar = FooMethod;  
bar(10); // 5.0
```

# Hurtig intro til lambda-udtryk (mere senere)

- En hurtig måde at lave ny **oneshot**-funktionalitet

```
public delegate double Converter(int x);  
public class Foo  
{  
    void Bar()  
    {  
        Converter timesTwo = v => v*2;  
        Console.WriteLine(timesTwo(2));  
    }  
}
```



A diagram consisting of a rectangular box with an orange border containing the text "Lambdaudtryk". A blue arrow points from the bottom center of this box to the lambda expression "v => v\*2;" in the code snippet above.

## FindProducts med delegate

```
public class Products {  
    public List<Product> FindProducts(Predicate<Product> predicate) {  
        List<Product> result = new List<Product>();  
  
        foreach (Product p in _products)  
            if (predicate(p))  
                result.Add(p);  
        return result;  
    }  
}
```

Bruger indbygget prædikat (generisk delegate)

Generisk delegateerklæring

```
delegate bool Predicate<in T>( T obj )
```

## FindProducts med delegate

```
public class Products {  
    public List<Product> FindProducts(Predicate<Product> predicate) {  
        List<Product> result = new List<Product>();  
  
        foreach (Product p in _products)  
            if (predicate(p))  
                result.Add(p);  
        return result;  
    }  
}
```

Tillader brug af **lambda-udtryk** (mere om det senere)

```
static void Main(string[] args)  
{  
    //.. Der er tilføjet produkter til products  
    List<Product> sonyProducts2 =  
        products.FindProducts(p => p.Manufacturer == "Sony");  
    List<Product> samsungProducts2 =  
        products.FindProducts(p => p.Manufacturer == "Samsung");  
    List<Product> cheapSonyProducts2 =  
        products.FindProducts(p => p.Manufacturer == "Sony" && p.Price < 1000);  
}
```

## FindProducts med delegate

```
public class Products {  
    public List<Product> FindProducts(Predicate<Product> predicate) {  
        List<Product> result = new List<Product>();  
  
        foreach (Product p in _products)  
            if (predicate(p))  
                result.Add(p);  
        return result;  
    }  
}
```

Tillader brug af **lambda-udtryk** (mere om det senere)

```
static void Main(string[] args)  
{  
    //.. Der er tilføjet produkter til products  
    List<Product> sonyProducts2 =  
        products.FindProducts(p => p.Manufacturer == "Sony");  
    List<Product> samsungProducts2 =  
        products.FindProducts(p => p.Manufacturer == "Samsung");  
    List<Product> cheapSonyProducts2 =  
        products.FindProducts(p => p.Manufacturer == "Sony" && p.Price < 1000);  
}
```

# Delegates

- Lambda-udtryk er **delegates** i C#, og er den ene af de to primære anvendelser af delegates. Den anden er events.
- En delegate er en datatype der repræsenterer en reference til metoder med en bestemt returtype og parametre.
- Brug af delegates består af 3 skridt:
  1. Definer **delegate type**: protokol der angiver *parametre* og *returtype* for metoder
  2. Lav **delegate instans**: Et objekt der refererer til en (eller flere) target metoder der overholder protokollen.
  3. Kald (Invoke) delegate instans

# Svarer til funktionspointere fra C

```
void qsort(void *base, size_t nitems, size_t size, int (*comparer)(const void *, const void*))
```

```
int cmpfunc (const void * a, const void * b) {  
    return ( *(int*)a - *(int*)b );  
}
```

```
int values[] = { 88, 56, 100, 2, 25 };
```

```
qsort(values, 5, sizeof(int), cmpfunc);
```

flere) target metoder der overholder protokollen.

3. Kald (Invoke) delegate instans



# Delegatetyper

## 1. Definition:

- Adskiller sig fra andre type-definitioner i udseende (klasser, structs, interfaces).
- Ligner en abstrakt metode – med **delegate** keyword i stedet.

```
delegate double Converter(int x);
```

# Delegatet

## 1. Def

- Ac
- str
- Lig
- ste

delega

```
delegate double Converter(int k);

class Foo
{
    double Half(int val)
    {
        return val*0.5;
    }
    double Third(int val)
    {
        return val/0.3333;
    }
    void Foo()
    {
        Converter f = Half;
        Console.WriteLine(f(9));

        f = Third;
        Console.WriteLine(f(9));

        f = i => i*3.0;
        Console.WriteLine(f(9));
    }
}
```

asser,

f er en variabel af typen Converter

Lambda-udtryk

# Instanser og kald

```
double TimesTwo(int x) { return 2 * x; }
```



## 2. Instanser

- Ved instantiering er alt dog (næsten) ved det gamle:
- **Converter** t = *TimesTwo*;
  - **Converter** t2 = new *Converter(TimesTwo)*; <- lang syntax

## 3. Kald delegate instans

- Udseende: delegate-instans(parametre)
- t2.Invoke(4); //kalder *TimesTwo* metoden; result: 8
- t2(4); // kort af ovenstående – resultat: 8

# Uinitialiserede delegates

- Hvis en delegate-variabel ikke peger på en metode? – null

```
Converter c;
```

```
c(1);
```



Exception

```
if (c != null)
```

```
    c(1);
```

```

delegate double Converter(int x); //1. delegate type

class Example
{
    void Run()
    {
        List<int> tal = new List<int>() { 3, 5, 7, 8 };
        Converter convert = TimesTwo; //2. Lav delegateinstans
        foreach (int t in tal)
            Console.WriteLine(convert(t)); //3. Kald instans

        convert = new Converter(Half); //2. ny instans (lang syntaks)
        foreach (int t in tal)
            Console.WriteLine(convert.Invoke(t)); //3. Kald (lang syntaks)
    }
}

double TimesTwo(int x) { return 2 * x; }
double Half(int x) { return .5 * x; }

```

# Generiske delegate typer

- En delegatetype kan (ligesom f.eks. klasser og interfaces) laves generisk ved at introducere typeparametre.
- Vi kan lave generiske delegates for at opnå kodegenbrug →

```
class SomeRandomClass {  
    public void DoString1(string a1) { }  
    public void DoString2(string a2) { }  
  
    public void DoInt1(int b1) { }  
    public void DoInt2(int b2) { }  
}  
  
delegate void DoStringDelegate(string a);  
delegate void DoIntDelegate(int b);  
  
//generiske delegates (ingen returtype)  
delegate void DoOneParamGeneric<T>(T a);  
  
//generiske delegates (med returtype)  
delegate TResult FuncWithNoParam<TResult>();
```

The diagram consists of four vertical lines. The first line is positioned between the class methods and the first two delegates. The second line is between the second two class methods and the next two delegates. The third line is between the generic delegate and the first non-generic delegate. The fourth line is between the non-generic delegate and the generic delegate with a return type. Arrows point from each of these lines to the corresponding method signature in the class: the first line points to DoString1 and DoString2; the second line points to DoInt1 and DoInt2; the third line points to DoStringDelegate; and the fourth line points to DoIntDelegate.

```
void TestMyGenericDelegates() {  
    SomeRandomClass target = new SomeRandomClass();  
    DoOneParamGeneric<string> a = target.DoString1;  
    DoOneParamGeneric<int> b = target.DoInt1;  
}
```

# Indbyggede generiske delegates: `Action<>` og `Func<>`

- Vi behøver dog ikke engang skrive vores egne generiske delegates.
- De indbyggede delegates **`Action<>`** og **`Func<>`** dækker alle metoder op til 16 parametre – henholdsvis med og uden returtype



# Action<> for handling

- Action<> repræsenterer metoder med void returtype, med op til 16 parametre.
- Den i'te type parameter angiver typen på den i'te metode parameter.


```
public delegate void Action(); //Til void metoder uden parametre
public delegate void Action<in T>(T arg); //med én parameter
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2); //to

//osv... (op til T16)
```

```
public static void EnMetodeMedTreParametre(int a, string b, double c) { }

public static void TestInbuiltGenericDelegates() {
    Action<string> write = Console.WriteLine;
    write("Hej med dig");

    Action<int, string, double> noget = EnMetodeMedTreParametre;
    noget(5, "Halløj", 17.3);
}
```



# Func<> for funktioner

- Func<> repræsenterer metoder **med returtyper** (op til 16 parametre).
- Den sidste type parameter (TResult) angiver returtypen.

```
public delegate TResult Func<out TResult>(); //0 parametre
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);

//osv... (op til T16)
```

```
public static void TestInbuiltGenericDelegates() {
    Func<decimal, decimal> abs = Math.Abs;
    Func<double, double, double> pow = Math.Pow;

    Console.WriteLine(abs(-5)); //Output: 5
    Console.WriteLine(pow(2, 3)); //Output: 8
}
```

# Andre (kendte) indbyggede generiske delegates

- Som sagt kan man angive alle metoder (op til 16 parametre) med Action<> og Func<>
- Selvdefinerede delegates er dog ikke uddøde endnu, da de ofte er mere læsbare/informative
- Indbyggede eksempler: Predicate, Comparison, Converter:

```
public void TestMoreGenericDelegates() {  
  
    //Metode der afgør om den angivne parameter opfylder kriteriet  
    ≡ Predicate<int> isEven1 = x => ((x % 2) == 0);  
    Func<int, bool> isEven2 = x => ((x % 2) == 0);  
  
    //Metode der sammenligner to parametre af samme type.  
    ≡ Comparison<int> greater1 = (a, b) => a < b ? -1 : a > b ? 1 : 0;  
    Func<int, int, int> greater2 = (a, b) => a < b ? -1 : a > b ? 1 : 0;  
  
    //Metode der konverterer en parameter fra én type til en anden type  
    ≡ Converter<int, double> con1 = x => x / .33;  
    Func<int, double> con2 = x => x / .33;  
}
```

# Covarians og contravarians

Covarians (**out**) på returtype => Target metode med subtype returtype kan bruges

Contravarians (**in**) på parametre => Target metode supertype parametre kan bruges

( Note: Covarians og contravarians kan stadig kun bruges med reference typer )

```
class Vehicle { public void Drive() { /* */ } }
class Car : Vehicle { }

class Program
{
    public static Car GetNewCar() { return new Car(); }
    public static void DriveTheVehicle(Vehicle v) { v.Drive(); }

    static void Main(string[] args)
    {
        //Covarians (out) - muliggør følgende:
        Func<Vehicle> carFunc = GetNewCar;
        Vehicle v = carFunc();

        //Contravarians (in) - muliggør følgende:
        Action<Car> carAction = DriveTheVehicle;
        Car c = new Car();
        carAction(c); //Kun Car argumenter godtages
    }
}
```

# Quiz

- Func<> eller Action<>?

```
public static int Add(int a, int b)
{
    return a + b;
}

public static int Sum(int a, int b, int c)
{
    return a + b + c;
}

public static void HelloDello()
{
    Console.WriteLine("Hello Delegates");
}
```



Lambda-udtryk

# Lambdaudtryk

- Har formen: **(*parametre*) => *udtryk/statement-blok***
- => udtales “går til”
- Hvis kun én parameter kan () udelades.
- Hvis parametre går til udtryk kan **return** og {} udelades
- Ofte kan parameterens type udledes og udelades (typeinferens)

```
static void Main(string[] args)
{
    Func<double, double> full = (double val) => { return .5 * val; };
    //Med type-inferens, enkelt parameter og returnering af udtryk retur så:
    Func<double, double> ultra = val => .5 * val;

    //Gammel dags syntax
    Func<double, double> delegatesyntax = delegate(double val) { return .5 * val; };
}
```

# Ydre variable (1/3)

- Et lambda-udtryk kan referere lokale variable og metode-parametre hvor den er defineret.

```
static string HelloAndGoodbye(string name)
{
    string hello = "Hello ";
    Func<string, string> greeter = goodbye => hello + name + goodbye;
    return greeter(", and goodbye"); //e.g., "Hello Kaj, and goodbye"
}
```

- En ikke-lokal variabel kaldes en **ydre variabel** (eller **fri variabel**)
- En ydre variable **fanges** (eller **bindes**) af lambda-udtrykket.
- Et lambda-udtryk der indeholder en ydre variabel kaldes også en *closure*.



# Ydre variable (2/3)

- Ydre variable evalueres når delegate *kaldes* – *ikke* når variable fanges.

```
int power = 3;  
Func<double, double> KaPow = n => Math.Pow(n, power);  
power = 2;  
Console.WriteLine(KaPow(4)); //4^2=16
```

- Lambda-udtryk kan også ændre ydre variable:

```
int power = 10;  
Func<double, double> KaPow = n => { power++; return Math.Pow(n, power); };  
power = 1;  
Console.WriteLine(KaPow(4)); //pow = 2 -> 16  
power++;  
Console.WriteLine(KaPow(4)); //pow = 4 -> 256
```

# Ydre variable (3/3)

- Ydre variables levetid følger delegatens levetid

**NB: Her returneres en closure!!**

```
public static Func<int> Incrementer()
{
    int i = 1;
    return () => i++;
}

public static void Main(string[] args)
{
    Func<int> inc = Incrementer();
    Console.WriteLine(inc()); //1
    Console.WriteLine(inc()); //2
    Console.WriteLine(inc()); //3
}
```

# Brugen af delegates

```
delegate void Log(string message);
```

```
void Print(string text)
{
    Console.WriteLine(text);
}
```

```
void WriteLogFile(string text)
{
    File.WriteAllText(@"c:\program.log", text);
}
```

## Brugen af delegates

```
delegate void Log(string message);  
void fun(){  
    Log log;  
    log = Print;  
    log("This goes to the console");  
  
    log = WriteLogFile;  
  
    log("This goes to a file");  
}
```

# Delegate-aritmetik

```
void fun(){  
    Log log;  
    log = Print;  
    log += WriteLogFile;  
    log("This line goes to both!");  
    log -= Print;  
    log("This goes to a file");  
    log -= WriteLogFile;  
    log("This will throw null-reference!");  
}
```

# Multicast delegates

- Alle delegates har *multicast* egenskaber
- En delegate instans kan indeholde en *liste* af target metoder.
- Target metoder tilføjes/fjernes med +/-
- Target metoder kaldes i rækkefølge
- Værdi af sidste metodekald returneres (kun issue ved non-void)

```
void Main()
{
    Action p = PrintFirst;
    p += PrintSecond;
    p(); // "First \n Second"
    p -= PrintFirst;
    p(); // "Second"
    p -= PrintSecond;
    p(); // NullReferenceException
}
```

```
void PrintFirst()
{
    Console.WriteLine("First ");
}

void PrintSecond()
{
    Console.WriteLine("Second");
}
```

# Problemet med delegates (1)

- Delegates som **eventmekanisme** har 2 alvorlige problemer:
  - Problem # 1: Ingen indkapsling af subscription
  - En abonnent må ikke kunne blande sig i andres abonnement

```
void Fun() {  
    Thermostat terma = new Thermostat();  
    AirCon airCon = new AirCon();  
    AutomaticCoffeeMaker acm = new AutomaticCoffeeMaker();  
    IceMaker im = new IceMaker(); //har void GotNewTemperature(double) metode  
  
    terma.TemperatureChange += airCon.ReceivedNewTemperature;  
    terma.TemperatureChange += acm.RegisteredNewTemperature;  
    terma.TemperatureChange = im.GotNewTemperature;  
  
    //Aircon og kaffemaskines target metoder bliver ikke længere kaldt!!  
}
```

↑  
Problem #1: "=" Overskriver invocation list

# Problemet med delegates (2)

- Problem 2: Ingen indkapsling af publikation
- Kun Publisher bør kunne publicere.

```
void Fun()
{
    Thermostat terma = new Thermostat();
    AirCon airCon = new AirCon();
    AutomaticCoffeeMaker acm = new AutomaticCoffeeMaker();
    airCon.TargetTemperature = 22;
    acm.TriggerTemperature = 19;

    terma.TemperatureChange += airCon.ReceivedNewTemperature;
    terma.TemperatureChange += acm.RegisteredNewTemperature;

    terma.Temperature = 26;
    terma.Temperature = 19;
    terma.TemperatureChange(87); //ikke resultat af tilstandsændring!!!
}
```



## Specielle Delegates: **events**

```
delegate void FooEH(int i);  
  
class FooBar  
{  
    public event FooEH Foo;  
    void notifySubscribers(){  
        Foo(10);  
    }  
}
```

## Specielle Delegates: **events**

```
class Client{  
    FooBar fooBar = new FooBar();  
    void main()  
    {  
        fooBar.Foo += handleFoo;  
    }  
    void handleFoo(int i)  
    { // handle foo here }  
}
```

Hvad gør events specielle?

```
class Client{
    FooBar fooBar = new FooBar();
    void main()
    {
        fooBar.Foo += handleFoo;

        fooBar.Foo = handleFoo;
        fooBar.Foo(10);
    }
}
```

# Events

- **event** keywordet løser de to problemer med delegates
  - 1) **events** forhindrer assignment udenfor omkransende klasse.
  - Gør følgende ulovligt (opdages af compileren):
    - `terma.TemperatureChange = im.TemperatureChanged;`
  - 2) **events** forhindrer invokation udenfor omkransende klasse
  - Gør følgende ulovligt (opdages af compileren):
    - `terma.TemperatureChange(87);`

# Traversering af invokationsliste

- Vi kan (som vi har set) bruge delegates' multicast egenskaber til at lave én invoke.
- Vi kan imidlertid også manuelt gennemløbe invokationslisten, og invoke hver target metode individuelt.

```
if (TemperatureChange != null){  
    foreach (Action<double> handler in TemperatureChange.GetInvocationList())  
    {  
        handler(value);  
    }  
}
```

# Traversering af invokationsliste

- Fejlhåndtering
- Hvis der optræder en uhåndteret exception hos en af subscribers vil efterfølgende delegate instanser ikke bliver kaldt.
- Dette kan forhindres ved at gennemløbe delegaten's invokations-liste og bruge try-catch om hver invokation ->
- Ulempen er så, at vi kan risikerer at begrave bugs i observers der skal håndteres; brug derfor try-catch med omtanke

# Traversering af invokationsliste

```
if (TemperatureChange != null)
    foreach (Action<double> handler in
        TemperatureChange.GetInvocationList())
    {
        try
        {
            handler(value);
        }
        catch (Exception ex) //svært at forudsige konkret fejl
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```