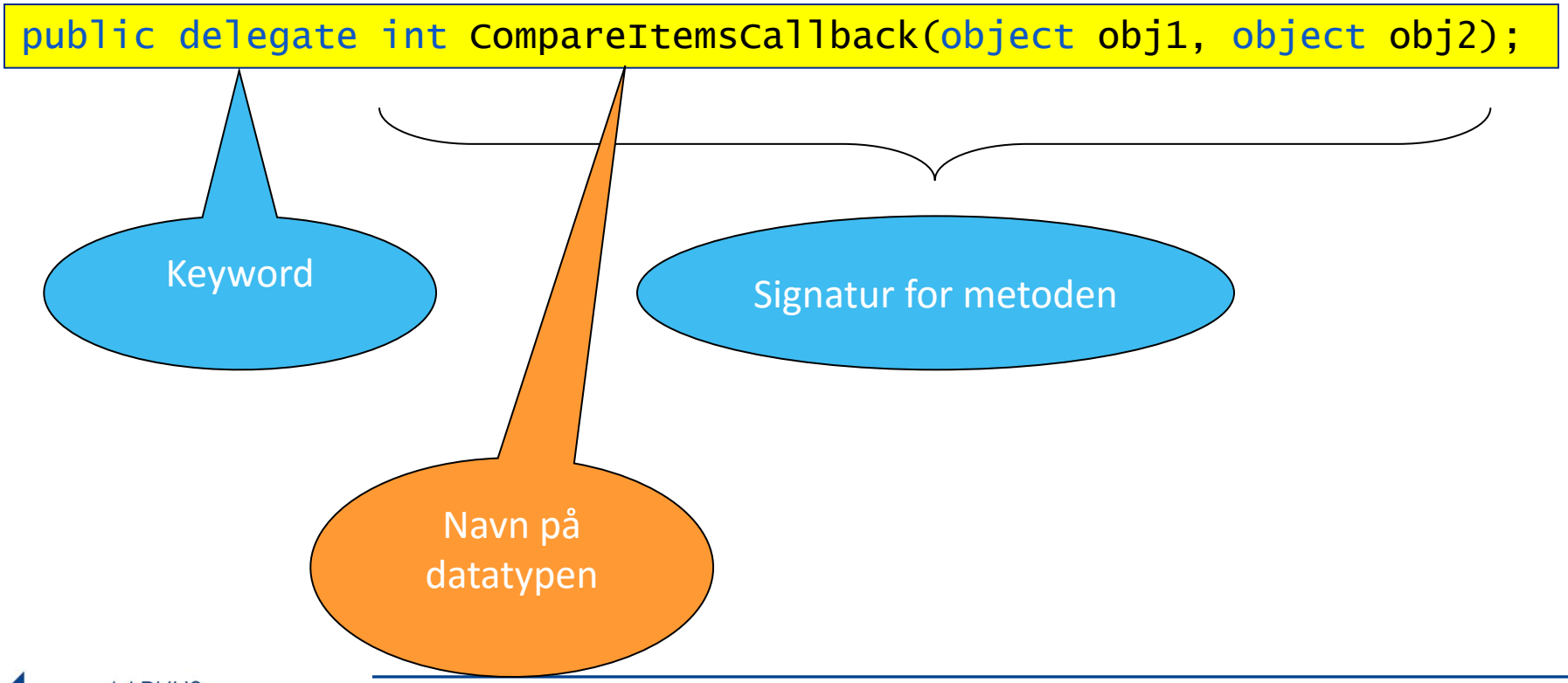
Delegates, Events, Anonymous Methods and Lambda expressions

In C#

Datatypen Delegate

- En delegate er en **reference** type der definerer en metode signatur

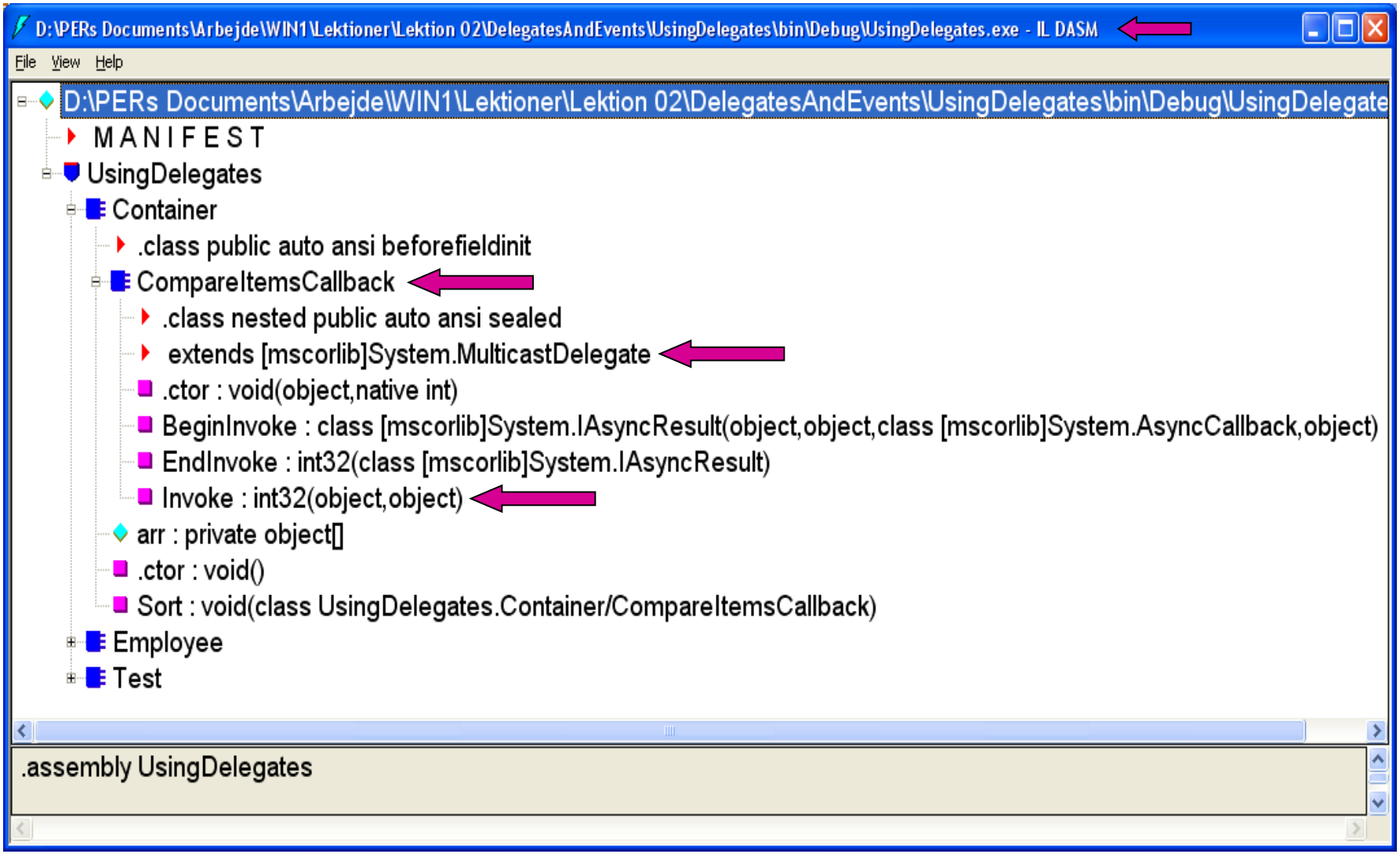


Karekteristika ved Delegate

- Delegates specificerer en kontrakt mellem en bruger (caller) og en implementør
  - *Det samme gør et interface*
- En delegate specificerer kun **en** funktion
  - *I modsætning til et interface der kan specificerer flere funktioner*
- En delegate svarer til en funktionspointer i C++
  - *Men kan lidt mere*
- En delegate bliver første skabt på runtime
  - I modsætning til interfaces der bliver skab på compile-time
- Kan bruges som en “low level” form for polymorfi
- Delegates er fundamentet for events i C#

Delegates Uncovered

```
public delegate int CompareItemsCallback(object obj1, object obj2);
```



Delegate to Static Function

Declare delegate type

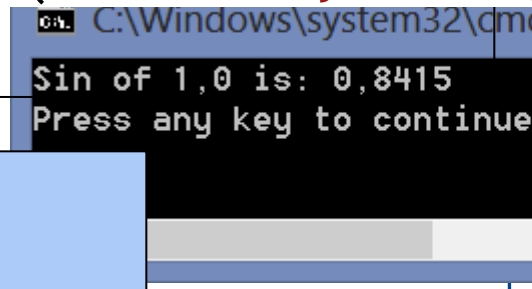
Instantiate a delegate

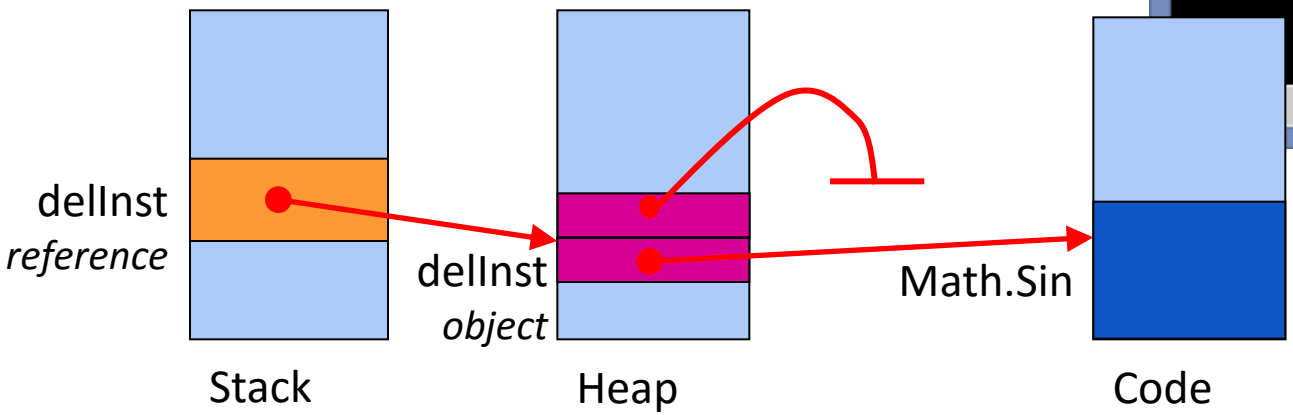
Invoke the delegate

```
delegate double Del(double x);

static void Main(string[] args)
{
    Del delInst = new Del(Math.Sin);

    double x = delInst(1.0);
    Console.WriteLine("Sin of 1,0 is: {
}
```





Delegate to Instance Member

```
public class User  {
  string name;
  public User(string name) {
    this.name = name;
    }
  public void Process(string message) {
    Console.WriteLine("{0}: {1}", name, message);
    }
}

class Test {
  delegate void ProcessHandler(string message);

  public static void Main() {
    User aUser = new User("George");
    ProcessHandler ph = new ProcessHandler(aUser.Process);
    ph("Wake Up!");
    }
}
```

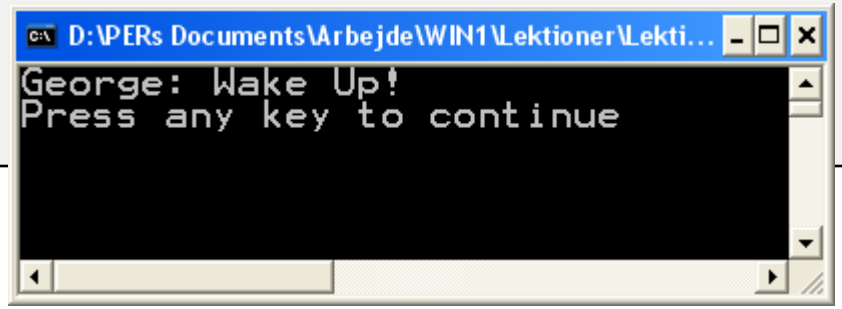
Same signature

Declare delegate

Instantiate delegate

To point to an instance method on an object

Invoke delegate

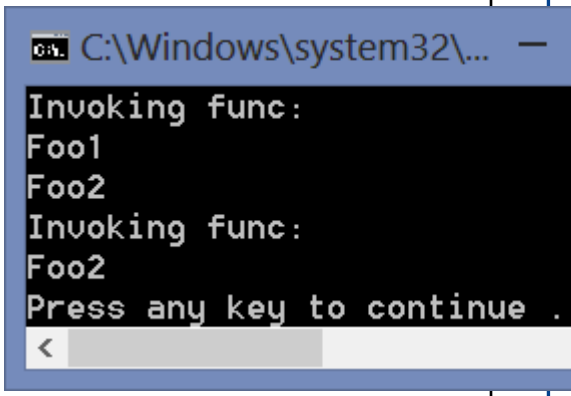


Multicast Delegates

- A delegate can hold and invoke multiple methods
  (Each delegate has an invocation list)
  - Multicast delegates must contain only methods that return void, else there is a run-time exception
  - If an exception is thrown by one of the sub-delegates, the remaining sub-delegates will not be called!
  - Methods are invoked sequentially in the order added
- The += and -= operators are used to add and remove delegates, respectively
  - += and -= operators are thread-safe

Multicast Delegate Example

```
delegate void SomeEvent(int x, int y);
static void Foo1(int x, int y) {
    Console.WriteLine("Foo1");
}
static void Foo2(int x, int y) {
    Console.WriteLine("Foo2");
}
static void Main(string[] args) {
    SomeEvent func = new SomeEvent(Foo1);
    func += new SomeEvent(Foo2);
    Console.WriteLine("Invoking func:");
    func(1, 2);                 // Foo1 and Foo2 are called
    func -= new SomeEvent(Foo1);
    Console.WriteLine("Invoking func:");
    func(2, 3);                 // Only Foo2 is called
}
```



Delegates Compared to Interfaces

- Offen you could use interfaces instead of delegates
- Interfaces are more powerful
  - Multiple methods
  - Inheritance
- Delegates are more elegant for event handlers
  - **Loose coupling**
  - **Less code**
  - Can easily implement multiple event handlers on one class/struct

Events

(Just a limited delegate)

Events Overview

- Event handling is a style of programming where one object notifies another that something of interest has occurred
  - A publish-subscribe programming model
  - The event model in C# is an implementation of the observer design pattern by use of delegates instead of subclassing.
- Events allow you to tie your own code into the functioning of an independently created component
  - This is loose coupling
- Events are a type of “callback” mechanism

Where to Use Events

- Events are well suited for user-interfaces
  - The user does something (clicks a button, moves a mouse, changes a value, etc.) and the program reacts in response
- Many other uses, e.g.
  - Time-based events
  - Asynchronous operation completed
  - Email message has arrived
  - A web session has begun

Events in C#

- C# has native support for events
- Based upon delegates
- An event is essentially a field holding a delegate
- However, public users of the class can only register delegates
  - **They can only call += and -=** (*the rest is private*)
  - They can't invoke the event's delegate
- Delegates allow multiple objects to register with the same event

Events Component-Side

- Define the event signature as a delegate

```
public delegate void EventHandler(object sender,
                                  EventArgs e);
```

- Define the event and firing logic

```
public class Button {
  public event EventHandler Click;

  protected void OnClick(EventArgs e) {
    // This is called when button is clicked
    if (Click != null)
       Click(this, e);
  }
}
```

Only invoke if someone has attached a delegate

Events User-Side

- Define and register an event handler

```
public class MyWindow: Window {
  Button okButton;

  public MyWindow() {
    okButton = new Button(...);
    okButton.Caption = "OK";
    okButton.Click += new EventHandler(OkClicked);
  }

  void OkClicked(object sender, EventArgs e) {
    MessageBox.Show ("You pressed the OK button");
  }
}
```

Note:

In C# 2.0 and later it is possible to use a short hand form (because the language now supports delegate inference) :

```
okButton.Click += OkClicked;
```

ANONYMOUS METHODS AND LAMBDA EXPRESSIONS

Anonymous Methods

- Anonymous methods allow you to specify the method for a delegate instance inline as part of the delegate instance creation expression

```
// Create a delegate type
delegate void Del(int x);

static void Main(string[] args) {
  // Instantiate the delegate using an anonymous method
  Del d = delegate (int k) {
    Console.WriteLine($"Hello {k} times from anonymous");
  };

d(27);
```

- By using anonymous methods, you reduce the coding overhead in instantiating delegates because you do not have to create a separate method

Anonymous Event handler

```
// Create a handler for a click event.
button1.Click += delegate(Object o, RoutedEventArgs e)
    { MessageBox.Show("Click!"); };
```

Lambda Expressions

- A lambda expression is an **anonymous function** that can contain expressions and statements, and can be used to create:
  - delegates or
  - expression tree types
- All lambda expressions use the lambda operator:

  =>

  which is read as "goes to"
- The left side of the lambda operator specifies the input parameters (if any)
- The right side holds the expression or statement block
- The lambda expression x => x * x is read "x goes to x times x"

Lambda Expressions

```
Func<string,int> returnLength;
```

- Using an anonymous method to create a delegate instance:

```
returnLength = delegate (string text) { return text.Length; };
Console.WriteLine(returnLength("Hello"));
```

- A long-winded lambda expression:

```
returnLength = (string text) => { return text.Length; };
```

- If you can express the whole of the body in a single expression

```
returnLength = (string text) => text.Length;
```

- If the compiler can "guess" the parameter types:

```
returnLength = (text) => text.Length;
```

- When the lambda expression only needs a single parameter, and that parameter can be implicitly typed:

```
returnLength = text => text.Length;
```

Lambda Demo

```
// Create a delegate type
delegate void Del(int x);

static void Main(string[] args) {
  // Instantiate the delegate using an anonymous method
  Del d = k => {
    Console.WriteLine($"Hello {k} times from anonymous");
  };

d(27);
```

Expression Trees

- Expression trees represent code in a tree-like data structure, where each node is an expression
- When a lambda expression is assigned to a variable of type
  `Expression<TDelegate>`
  the compiler emits code to build an expression tree that represents the lambda expression
- The C# compiler can only generate expression trees from expression lambdas (single-line lambdas)
- Example:

```
Expression<Func<int, bool>> myLambda = num => num < 5;
```

The Use of Expression Trees

- Both LINQ to Objects and LINQ to SQL start with C# code and end with query results
- The ability to execute the code remotely as LINQ to SQL does comes through expression trees