

ECNF2 Zusammenfassung

Roland Hediger

17. Juni 2013

Inhaltsverzeichnis

1	Essenzielle Sachen	4
1.1	Feature Vergleich	4
1.2	Echte Unterschiede	4
1.3	Typ Baum	4
1.4	Boxing and Unboxing	5
1.5	Klassen	5
1.6	Properties	5
1.6.1	Automatische Properties	5
1.7	C# modifiers	6
1.8	Indexers	6
1.9	Abstract Classes	6
1.10	Interfaces	7
1.11	Standard Collections	8
1.12	Switch Anweisungen	8
1.13	Control Statements	9
1.14	Arrays	9
1.15	Variablen - Stack + Heap	9
1.15.1	Dynamische Variablen	10
1.16	Parameter	10
1.17	Partial Classes	10
1.18	Typechecking und Casting	10
1.19	Vererbung	10
1.20	Override /Methodenüberschreibung	11
1.21	Methoden verstecken	11
1.21.1	Base Konstruktor aufrufen	12
1.22	Generics	12
1.22.1	Generics und Vererbung	13
1.22.2	Generic Methods	13
1.23	FILE/IO	14
2	Delegates	14
2.1	Java vs c# Delegate	14
2.2	Nutzung von Delegates	15
2.3	Delegate Characteristics	15
2.4	Observer Pattern	16
2.5	Predicate Delegates	17
2.6	Func + Action Delegates	18
2.7	Exceptions	18
2.7.1	Exception Characteristics	18
3	Operator Overloading	19

4	Extension Methods	19
4.1	Objekt erweitern (vorsicht)	20
5	Yield	20
5.1	Yield Einschränkungen	20
6	Sicher und Unsicheren code	21
6.1	Pointer Operations	21
6.2	Pinning Objects	21
7	Reflection	22
7.1	Reflection Grundkonzepte	22
7.2	Type Info zur Laufzeit	22
7.3	Laden von Assemblies	22
7.4	Innerhalb Typinfo	23
7.4.1	MemberInfo	23
7.5	Bigger Picture	24
8	Attributes	24
8.1	Benutzer definierte Attributen	24
8.2	AttributeUsage	25
9	LINQ + Lambda	25
9.1	Lambda	25
9.1.1	Lambda Execution Context	26
9.2	LINQ	26
9.3	Advantages and bla bla	26
9.3.1	LINQ General Syntax	27
9.3.2	LINQ Key Features	27
9.3.3	Query Chaining	27
9.3.4	Formulating LINQ Expressions	28
9.3.5	LINQ and objects	28
9.3.6	Local vs Interpreted Query	28
9.3.7	Feature Summary	28
10	Dynamic Programming	28
10.1	Dynamic vs Static Languages	28
10.2	Dynamic Language Architecture	29
10.3	DLR Concepts	29
10.4	Using Dynamic	30
10.5	Implementing Dynamic	30
10.5.1	API	30
10.6	Dynamic Language Integration - Python	30
10.6.1	Data Passing	31
10.7	COM Interop	31
10.8	Native Code	31
10.8.1	DLL Import Structure	32
10.8.2	Parameter Marschling	32
10.8.3	Passing Structs	32
10.9	Implicit P Invoke C++	32
11	Diagnostics and Garbage Collection	33
11.1	General	33
11.2	Object Creation	33
11.3	GC in .net	34
11.3.1	Reachability	34
11.3.2	Cleaning up objects	34

11.3.3	The collection process	34
11.3.4	Finilisation guidelines	34
11.4	Explicit Resource Management	35
11.4.1	Semantics	35
11.4.2	Dispose Pattern	35
11.4.3	Finalise vs Dispose	36
11.5	System.GC	36
11.6	GC Explained	36
12	Assembly and CIL	37
12.1	Role of Assemblies	37
12.2	Tools	37
12.3	Assembly Info	38
12.4	Managing Private Assemblies	38
12.5	Make assembly available publicly	38
12.5.1	Signing Assembly	39
12.6	What is CIL	39
12.7	CIL	39
12.7.1	Hello world example	39
12.7.2	CIL Programming model	40
12.7.3	Execution State	40
12.7.4	Base Instruction Set	40
12.7.5	Names and Directives in CIL	41
12.7.6	Value Classes /Structs	41
12.7.7	Class Declarations	42
12.7.8	Defining Methods	42
12.7.9	Method bodies	43
13	Concurrency	43
13.1	Basic Concepts	43
13.2	.Net Async Concepts	43
13.3	Threads vs Tasks	43
13.4	Task class API	44
13.4.1	Task Based Async Pattern /TAP	44
13.4.2	Task Combinators	44
13.5	Low Level Threading API	44
13.5.1	Background Threads	45
13.5.2	Passing Data to Threads	45
13.5.3	Thread Recycling with Threadpools	45
13.5.4	AsyncDelegates	46
14	Parallel Programming	46
14.1	Partitioning Strategies	46
14.2	Overview	46
14.3	Parallel Considerations	47
14.4	Task Parallel Library	47
15	Code Contracts	48
15.1	Contract Principles	48
15.2	Precondition Guidelines	48
15.3	Object invariants	49
15.4	Quantifeiers	49
15.5	Contract inheritance	49
15.6	Contracts and Interface	49
15.7	Static analysis	49

1 Essenzielle Sachen

1.1 Feature Vergleich

Wie in Java	Wie in C++
Interfaces	Structs
Exceptions	Operator Overloading
Threading	Pointer arithmetic in unsicherer Code
Namespaces	Manche syntaktische Details
Starke Typisierung	Generics (<i>Templates</i>)
Garbage Collection	
Reflektion	
Dynamisches Laden von Code	
Enums	
Generics	
Attributen	
Anonyme Typen	
foreach schleife	

1.2 Echte Unterschiede

Wirkliche Unterschiede	Syntaktische Zucker
Call by Reference Parameter (Stern in c++)	Properties
Stack allozierte Objekten (structs werden auf Stack alloziert)	Events
Block Matrizen	Delegates
Uniform Typ System	Indexers (Indexoperator Überladung sozusagen)
goto Anweisung	Boxing / Unboxing (Referenz in Value Typ und umgekehrt)
Versionierung	Statische Klassen
Anonyme Methoden	Partielle Klassen (über mehrere Dateien zerlegt)
Funktionale Programmierung	
Native interoperabilität (Aufrufe von Methoden in anderen Sprachen (CLR))	
Dynamische Programmierung	
LINQ	

1.3 Typ Baum

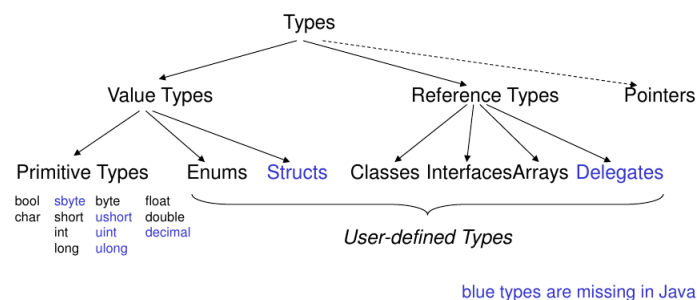


Abbildung 1: Typ Baum von c hash

1.4 Boxing and Unboxing

Listing 1: Boxing und Unboxing

```
1 // boxing - Werttyp in Referenztyp umwandeln - Wert 3 ist jetzt in einem
    Heapobjekt geboxed
    object obj = 3;
    // Unboxing Referenztyp in Werttyp konvertieren durch casting.
    int x = (int) obj;
```

1.5 Klassen

Listing 2: Deklaration von eine Klasse

```
1 class Hello {
    private String name;
    private void Greet() {...}
    public static void Main(String[] args) {...}
}
```

Statische Konstruktor 1 mal per Typ ausgeführt, bevor alle instanzen oder andere statische mitglieder der Klasse zugegriffen werden können. *Bevor ist nicht klar wegen Intermediate Language Übersetzung*

Statische Felder werden bevor statische Konstruktor initialisiert.

1.6 Properties

Properties sind syntaktische Zucker für Getter und Setter methoden. Die sind auch ein bisschen angenehmer. Beispiel :

Das Property eliminiert die Methoden getFileName und setFileName. Stattdessen kann man das gleiche tun durch normalen zugriff mittels der Punkt :

data.FileName

Listing 3: Properties Beispiel

```
//Deklaration
class Data {
    string f;
    public string FileName{
5      set { f = value; }
      get { return f; }
    }
}
//Nutzung
10 Data d = new Data();
    d.FileName = "myFile.txt";
    string s = d.FileName;

// Teil 2 Automatische Properties
15 class Data {
    public string FileName { get; set;}
}
```

1.6.1 Automatische Properties

Hier wird die private Variable vom System generiert im hintergrund (string f von oben), sowie auch der Inhalt von get und set

Es funktioniert genau gleich wie im ersten Teil des Beispiels

1.7 C# modifiers

Fields :

Zugriff Modifiers	public,internal,private,protected
Statisch	static
Vererbung	new Selten benutzt.
Unsafe code	(pointers usw) unsafe
Read-only (final)	readonly
Threading	volatile

Methoden :

Zugriff Modifiers	public,internal,private,protected
Statisch	static
Vererbung	new,virtual,override,abstract,sealed
Unsafe unmanaged code	(pointers usw) unsafe,extern

1.8 Indexers

Listing 4: Indexers

```
public class Portfolio
2 {
    Stock[] stocks;
    ...
    public Stock this[int index] // indexer impl{
        get { return this.stocks[index];}
7     set { this.stocks[index] = value; }
    }

    Console.WriteLine(portfolio[i].Symbol);
```

1.9 Abstract Classes

Listing 5: Abstrakte Klassen

```
abstract class Stream {
    public abstract void Write(char ch);
    public void WriteString(string s) { foreach (char ch in s) Write(s); }
4 }
class File : Stream {
    public override void Write(char ch) {... write ch to disk ...}
}
```

1. Abstrakte Methoden sind nicht implementiert.
2. Abstrakte Methoden sind implizit virtual.

Definiton von Virtual :

A virtual method can be redefined. The virtual keyword designates a method that is overridden in derived classes. We can add derived types without modifying the rest of the program. The runtime type of objects thus determines behavior.

3. Falls eine Klasse abstrakte Methoden hat, die deklariert oder vererbt sind, muss die Klasse selbst abstrakt sein.
4. Abstrakte klassen können nicht instanziiert werden.

1.10 Interfaces

Listing 6: Interface Beispiel

```
public interface IList : ICollection, IEnumerable {
2  int Add (object value);
    // methods
    bool Contains (object value);
    ...
    bool IsReadOnly { get; } // property
7  ...
    object this [int index] { get; set; }
    //indexer
}
```

1. Interface ist definiert so dass es in eine gewisse Art und Weise ähnlichkeiten mit C++ hat.
2. Ein Interface ist eine rein abstrakte Klasse mit nur Methodensignaturen und keine implementationen.
3. Es ist jedoch möglich dass ein Interface Methoden, Properties, sowie auch Indexers und Events beinhalten kann.
4. Interface Mitglieder sind implizit public abstract (virtual).
5. Interface Mitglieder müssen nicht statisch sein.
6. Klassen sowohl als auch **Structs** können mehrere Interfaces implementieren.
7. Interfaces können andere Interfaces erweitern.

Klassen, Structs und Interfaces

1. Eine Klasse vererbt von einer einzigen Stamm/BaseKlasse. Es kann aber mehrere Interfaces implementieren.
2. Ein Struct kann nicht von eine Stamm/Baseklasse (Typ) vererben. Es kann doch mehrere Interfaces implementieren.
3. Alle Interface Methoden müssen implementiert werden.
4. *Interface methoden müssen keine override haben*
5. Interface Methoden können als virtual oder abstrakt implementiert werden.
6. Subclass override von Methoden = explizite virtual Modifier.

1.11 Standard Collections

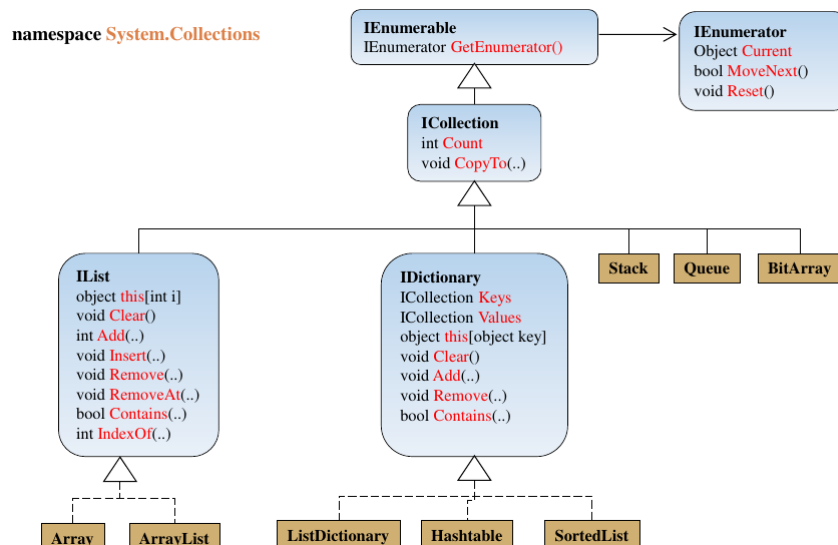
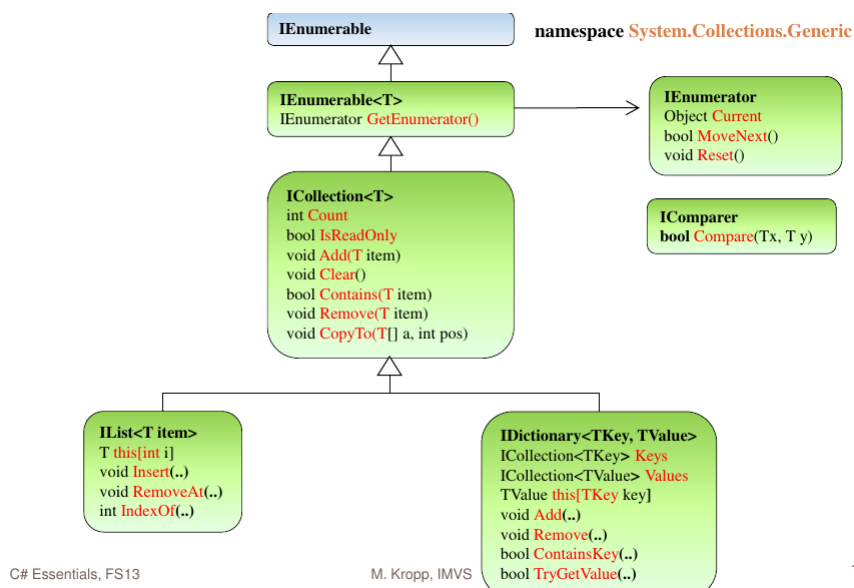


Abbildung 2: Standard Collections



C# Essentials, FS13

M. Kropp, IMVS

18

Abbildung 3: Generic Collections

1.12 Switch Anweisungen

Listing 7: Switch Anweisung

```

switch (country) {
    case "England": case "USA":
3       language = "English";
        break;
    case "Germany": case "Austria": case "Switzerland":
        language = "German";
        break;
8       case null:
        Console.WriteLine("no country specified");
  
```



```

        break;
    default:
        Console.WriteLine("don't know the language of " + country);
        break;
    }

    //Bemerkung keine Fall through - alles muss mit break return goto oder throw
    //terminiert werden.
    // Typen : int,char,enum,oder string, oder null

```

1.13 Control Statements

Listing 8: foreach Beispiel

```

foreach (char c in "beers"){
    Console.WriteLine(c);
}

```

break Endet Schliefe oder switch.


continue Macht nächste Iteration der Schliefe.

goto ist des Teufels

1.14 Arrays

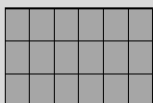
One-dimensional arrays

```
int[] a = new int[6];
int[] c = {3, 4, 5};
String[] d = new String[10]; // array of references
```



Multidimensional arrays (rectangular)

```
int[,] a = new int[4, 6]; // block matrix
int[,] b = {{1, 2, 3}, {4, 5, 6}}; // can be initialized
int[,][] c = new int[2, 4, 2];
```



Multidimensional arrays (jagged)

```
int[][] a = new int[2][]; // array of other arrays
a[0] = new int[] {1, 2, 3}; // cannot be initialized
a[1] = new int[] {4, 5, 6}; // directly
```

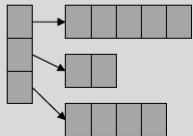


Abbildung 4: Arrays

1.15 Variablen - Stack + Heap

Stack + Heap Valuetypen sind immer auf dem Stack.

Referenztypen sind auf dem Heap (Objekten new operator usw.) Garbage collection hier.

Definite Assignment Definitive Zuweisung - Unmöglich uninitialisierte Hauptspeicher zuzugreifen.

Lokale Variablen : 1) Init 2) Zugriff.

Felder + Arrays = Automatische Init.

1.15.1 Dynamische Variablen

Listing 9: Dynamische Variablen

```
var x = 5; // is equivalent to int x = 5; implicit type assignment, compile
           time checking
2 dynamic d = 5; // really dynamic, runtime checking
  d =      hello    ;
```

1.16 Parameter

Listing 10: Parameter Beispiel

```
1 //Modifiers
  void Parameters(int i, ref int ref_i, out int out_i)
  //Variable Anzahl parameter müssen letzte Parameter sein
  void Parameters(int i, params int[] ints)
```

1.17 Partial Classes

Eine Klasse über mehrere Dateien. VS.Net Trennung - Hand von generierte Code. Nicht nötig.

1.18 Typechecking und Casting

□ Run-time type checks – the “is” operator

```
a = new C();
if (a is C) ... // true, if the dynamic type of a is C or a subclass; otherwise
               false
a = null;
if (a is C) ... // false: if a == null, a is T always returns false
```

□ Cast: type cast with runtime exception

```
A a = new C();
B b = (B) a; // if (a is b) = true, type(a) is B in this expression; else
             // exception
C c = (C) a;

a = null;
c = (C) a; // ok → null can be casted to any reference type
```

□ as: similar to (T)v but no runtime exception

```
A a = new B();
B b = a as B; // (if (a is B) == true) b = (B)a; else b = null;
C c = a as C; // c == null, because a is not of type C: (a is C) == false

a = null;
c = a as C; // c == null
```

Abbildung 5: Typcheck + Casting

1.19 Vererbung

Listing 11: Vererbungsbeispiel

```
1 class B : A {
  // subclass (inherits from A, extends A)
  int b;
  public B() {...}
  public void G() {...}
6 }
```

- Konstruktoren nicht vererbt.
- Vererbte methoden können überschrieben werden.
- **Single Inheritance** nur von einer einzigen Klasse ableitbar.
- Klasse kann nur von einer Klasse vererben.
- Klasse ohne Vererbung vererbt automatisch von Object

1.20 Override /Methodenüberschreibung

Listing 12: Override Beispiel

```

class A {
    public void F() {...} // cannot be overridden
    public virtual void G() {...} // can be overridden in a subclass
4 }

class B : A {
    // warning: hides inherited F(), default is new
    public void F() {...}
9 // warning: hides inherited G(), default is new
    public void G() {...}
    public override void G() { // ok: overrides inherited G
        ...
    }
14 }

```

- Methodensignaturen müssen identisch sein.
- Properties und Indexers auch überschreibbar (virtual/override)
- **Statische Methoden können nicht überschreiben werden**

1.21 Methoden verstecken

New Keyword muss benutzt werden.

Listing 13: Versteckung Beispiel

```

class A {
    public int x;
    public void F() {...}
    public virtual void G(){...}
5 }

class B : A
{
    public
10 new int x;
    public
    new void F() {...}
    public
    new void G() {...}
15 }

B b = new B();
b.x = ...;
20 b.F(); ... b.G();
    // accesses B.x
    // calls B.F and B.G
    ((A)b).x = ...;
    // accesses A.x!

```

```

25 ((A)b).F(); ... ((A)b).G();
    // calls A.F and A.G!

```

1.21.1 Base Konstruktor aufrufen

Listing 14: Basiskonstruktor implizit aufrufen

```

class A {
...
3 }
class B : A {
    public B(int x) {...}
}

8 // A() dann B(int x)
class A {
    public A() {...}
}
class B : A {
13     public B(int x) {...}
    }
    //das gleiche

class A {
18     public A(int x) {...}
    }
class B : A {
    public B(int x) {...}
    }
23 B b = new B(3);

//Error : A hat kein Defaultkonstruktor mehr.

```

Listing 15: Basiskonstruktor explizit aufrufen

```

class A {
    public A(int x) {...}
}
4 class B : A {
    public B(int x)
    : base(x) {...}
    }
    // A(int x) dann B(int x)

```

1.22 Generics

Listing 16: Generics Beispiel

```

class Buffer <Element, Priority> {
2     private Element[] data;
    private Priority[] prio;
    public void Put(Element x, Priority prio) {...}
    public void Get(out Element x, out Priority prio) {...}
    }
7
Buffer<int, int> a = new Buffer<int, int>();
a.Put(100, 0);
int elem, prio;
a.Get(out elem, out prio);
12 Buffer<Rectangle, double> b = new Buffer<Rectangle, double>();
    b.Put(new Rectangle(), 0.5);
    Rectangle r; double prio;
    b.Get(out r, out prio);

```

```

17 //GENERICIS MIT CONSTRAINTS

    class OrderedBuffer <Element, Priority> where Priority: IComparable {
        Element[] data;
        Priority[] prio;
22 int lastElem;
        interface or base class
        ...
        // sorts x according to its priority into buffer
        public void Put(Element x, Priority p) {
27 int i = lastElement;
        while (i >= 0 && p.CompareTo(prio[i]) > 0) {
            data[i+1] = data[i]; prio[i+1] = prio[i]; i--;
            data[i+1] = x; prio[i+1] = p;
        }
32 }

        OrderedBuffer<int, int> a = new OrderedBuffer<int, int>();
        a.Put(100, 3);

37 //Ordered buffer implementiert IComparable

```

1.22.1 Generics und Vererbung

Listing 17: Generics und Vererbung

```

    class Buffer <Element>: List<Element> {
        ...
3 public void Put(Element x) {
    this.Add(x); // Add is inherited from List
}

}

8
// from a non-generic class
class T<X>: B {...}
//T from an concrete generic class
class T<X>: B<int> {...}
13 //from a generic class
//with the same placeholder
class T<X>: B<X> {...}

//CONSTRAINTS
18
public class BaseClass<T> where T : ISomeInterface
{...}
    public class SubClass<T> : BaseClass<T> where T: ISomeInterface
{...}

```

1.22.2 Generic Methods

Listing 18: Generische Methoden

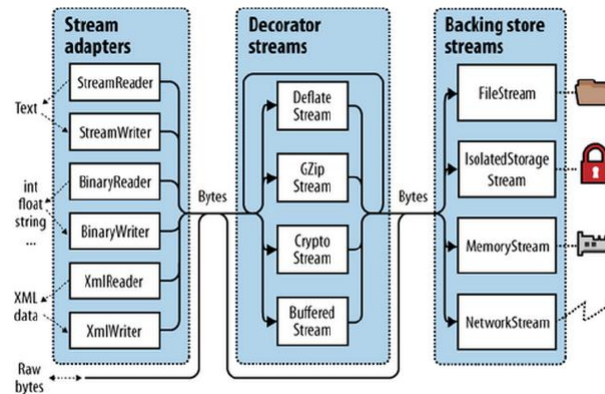
```

1 static void Sort<T> (T[] a) where T: IComparable {
    for (int i = 0; i < a.Length-1; i++) {
        for (int j = i+1; j < a.Length; j++) {
            if (a[j].CompareTo(a[i]) < 0) {
                T x = a[i]; a[i] = a[j]; a[j] = x;
6 }
        }
    }

    int[] a = {3, 7, 2, 5, 3};
11 ...
    Sort<int>(a);
    //Typ inferenz hier, man muss nicht <int> angeben.

```

1.23 FILE/IO



From J. and B. Albahari: *C# 4.0 in a Nutshell*, 4th Edition

Abbildung 6: IO Architektur

Listing 19: Stream Beispiele

```
Stream stream = new FileStream("test.txt", FileMode.Create);
2 Console.WriteLine(stream.CanRead); // true
  Console.WriteLine(stream.CanWrite); // true
  Console.WriteLine(stream.CanSeek); // true
  stream.WriteByte(201);
  stream.WriteByte(210);
7 stream.Position = 0;
  Console.WriteLine(stream.ReadByte());
  stream.Close();

string filename = "Text.txt";
12 TextWriter writer= new StreamWriter(filename);
  writer.WriteLine("First line.");
  writer.WriteLine("Last line.");
  writer.Close();
  TextReader reader = new StreamReader(filename);
17 Console.WriteLine(reader.ReadLine());
  Console.WriteLine(reader.ReadLine());
  reader.Close()
```

2 Delegates

2.1 Java vs c# Delegate

```
class MyComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
4 return (o1>o2 ? -1 : (o1==o2 ? 0 : 1));
    }
    ...

    Collections.sort ( list , new MyComparator() ) ;
9
c#
// public delegate int Comparison<in T>(T x, T y);
```

```

    Comparison<int> myComparer = delegate(int i1, int i2)
    {
14  return i1.CompareTo(i2);
    };
    list.Sort(myComparer);

    The Sort-Method Interface
19 List<T>.Sort(Comparison<T> comparison);
    In Fact: You can use any method following the
    required delegate interface

```

2.2 Nutzung von Delegates

Ein Delegate ist ein Referenztyp dass einen Methodensignatur definiert.

Listing 20: Delegate Intro

```

//muster
delegate returnType DelegateName([DelArguments]);
3
//beispiel
// method signature with keyword delegate
delegate void Notifier(string sender);
Notifier greetings;
8 void SayHello(string sender) { // just provide same interface
  Console.WriteLine("Hello from " + sender);
}

greetings = new Notifier(SayHello) // full version, or
13 greetings = SayHello; // simplified assignment: delegate type
// is inferred from the type of the
// left-hand side

greetings("John");
18 // invokes SayHello("John")

//invoke another method

void SayGoodBye(string sender) { // just another method
23 Console.WriteLine("Good bye " + sender);
}
greetings = SayGoodBye;
// assign the method
greetings("John"); // SayGoodBye("John") => "Good bye John"

```

Bemerkungen

- Delegate Variable kann null sein.
- Exception bei aufruf von diese Nullvariable.
- Delegateobjekten sind Objekten der ersten Klasse : können als Parameter benutzt weden oder in Datenstruktur gespeichert werden.

2.3 Delegate Characteristics

- Delegates sind objektorientierte , typsichere und sicher, Funktionszeiger.
 Delegateinstanz beinhaltet mehrere Methoden die statisch oder nicht statisch sein können.
- Delegates erlauben dass man Methoden als Parameter übergeben kann.
- Benutzt um "callback" Methoden zu fehinieren.
- Chaining : Mehrere Methoden können bei einem Event aufgerufen werden.

- Ref Parameter sind durch alle Methoden weitergeleitet.

Listing 21: KeyHandler Beispiel

```
1 public delegate void KeyEventHandler (object sender, KeyEventArgs e);
  public class KeyEventArgs : EventArgs {
    public virtual bool Alt { get {...} }
    // true if Alt key was pressed
    public virtual bool Shift { get {...} } // true if Shift key was pressed
6 public
    bool Control { get {...} } // true if Ctrl key was pressed
    public
    bool Handled { get {...} set {...} } // indicates if event
    // was already handled
11 public
    int
    KeyValue { get {...} } // the typed keyboard code
    ... }
    class MyKeyEventArgs {
16 public event KeyEventHandler KeyDown;
    public KeyPressed() {
      KeyDown(this, new KeyEventArgs(...));
    }
    }
21 class MyKeyListener {
    public MyKeyListener(...) { keySource.KeyDown += HandleKey;}
    void HandleKey (object sender, KeyEventArgs e) {...}
    }
```

2.5 Predicate Delegates

Delegate der einen Boolean zurückgibt.

Listing 22: Predicate Beispiel

```
    class List<T> {
      ...
      List<T> FindAll(Predicate<T> match);
4 T Find(Predicate<T> match);
    int FindIndex(Predicate<T> match);
    ...}

    public delegate bool Predicate<T> ( T obj )
9

    bool ProductGT10(Point p)
    {
      if (p.X * p.Y > 100000){
14 return true;
      }
      return false;
    }

19 //Anonymous

    List<Point> FindWithAnonymousDelegate(List<Point> list)
    {
24 return list.FindAll(delegate(Point p)
    { // here starts the anonymous method
      if (p.X * p.Y > 100) { return true; }
      return false;
    });
29 }
```

2.6 Func + Action Delegates

Listing 23: Func Delegate

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

5 public static void FuncDelegateSample() {
    Func<string, string> convertMethod = UppercaseString;
    Console.WriteLine(convertMethod("Dakota"));
}
    private static string UppercaseString(string inputString) {
10 return inputString.ToUpper();
    }
```

Listing 24: Action Delegate

```
    delegate void Action ();
2    delegate void Action<in T> (T arg);
    delegate void Action<in T1, in T2> (T1 arg1, T2 arg2);

    private static void ActionDelegateSample()
    {
7    Action<string> act = ShowMessage;
    act("C# language");
    }
    private static void ShowMessage(string message)
    {
12 Console.WriteLine(message);
    }
```

2.7 Exceptions

Listing 25: Exception Beispiel

```
1    FileStream s = null;
    try {
        s = new FileStream(curName, FileMode.Open);
        ...
    } catch (FileNotFoundException e) {
6    Console.WriteLine("file {0} not found", e.FileName);
    } catch (IOException) {
        Console.WriteLine("some IO exception occurred");
    } catch {
        Console.WriteLine("some unknown error occurred");
11 } finally {
        if (s != null) s.Close();
    }
```

2.7.1 Exception Characteristics

Catch Klausel in sequenziellen Reihenfolge abgearbeitet.

Finally wird immer ausgeführt.

Exception Parametername kann weggelassen werden im Catchklausel.

Exceptions immer von System.Exception abgeleitet.

Catchklauselbehandlung Kette wird Rückwärts travasiert bis Methode mit passende Catchklausel gefunden ist. Falls nicht dann Stack trace.

Fangen von Exceptions Müssen nicht sein. Keine Unterschiede zwischen Checked und Unchecked Exceptions.

3 Operator Overloading

Listing 26: Operator Overloading Beispiel

```
1  public struct Rational {
    ...
    public static Rational operator+ (Rational lhs, Rational rhs)
    {
        // here comes the implementation
6  return new Rational(...);
    }
}

    Rational r3 = r1 + r2;
11 r3 += r2; // !!! Provided for FREE
    public override bool Equals(object o)

    public static bool operator== (Rational lhs, Rational rhs)
    public static bool operator!= (Rational lhs, Rational rhs)
16 if ( r1.Equals(r2) ) { ... }
    if ( !r1.Equals(r2) ) { ... }
    if ( r1 == r2 ) { ... }
    if ( r1 != r2 )
    { ... }
21
    public struct Rational {
    ...
    public static implicit operator Rational(int i)
    {
26 return new Rational(i,1);
    }
}
    From int to Rational
    Rational r = 2;
31
    public struct Rational {
    ...
    public static explicit operator double(Rational r)
    {
36 return (double) r.Numerator / r.Denominator;
    }
}

    From Rational to double
41 Rational r = new Rational(2,3);
    double d = (double) r;
```

4 Extension Methods

Listing 27: Extension Methods Beispiel

```
    public static String Concat(this String[] arr, String sep) {
2  StringBuilder sb = new StringBuilder();
    sb.Append(arr[0]);
    for (int i=1; i<arr.Length; i++)
        sb.Append(sep).Append(arr[i]);
    }
7  return sb.ToString();
    Just call the new method, as it would be a regular string method
    String[] stringArray = { "www", fhnw, "ch" };
    Console.WriteLine(stringArray.Concat("."));

12
namespace MyExtensions {
    public static class MyExtensionsClass {
        public static void MyMethod(this My my) {...}
```

```

    }
17 }
    Use namespaces and using to control scope
    namespace N1 {
        using MyExtensions;
        // My.MyMethod usable here
22 }
    namespace N2 {
        // My.MyMethod not usable here
    }

```

4.1 Objekt erweitern (vorsicht)

Listing 28: Objekt Extension Method

```

    public static class ObjectExtensions {
    public static String ToString(this Object receiver,
    params Object[] args) {
    return "ToString, idiot!"; } }

```

- Vorsicht mit Methoden im Scope von Ausländischer Namespace.
- Vorsicht welche Klassen haben Erweiterungen.
- Diese Methode kann auf irgendein Objekt benutzt werden.

5 Yield

Listing 29: yield Beispiel

```

    public static IEnumerable<int> DoTest(int num) {
    for (int i = 0; i < num; i++) {
    yield return i;
    }
5 yield break;
    }
    static void Main(string[] args) {
    foreach (int num in DoTest(8)) {
    Console.WriteLine(num);
10 }
    }

    public
    yield static return IEnumerable<int> and yield To(this break
15 int first, int last)
    {
        Must for (var return i = IEnumerator first; i <= last; or i++)
        IEnumerable
        {
20 yield return i;
        }
        }
        foreach (int i in 3.To(10))
        {
25 j = i*i;
        }
    }

```

Yield packt alles automatisch in return typ ein im Hintergrund.

5.1 Yield Einschränkungen

1. Gibt immer IEnumerator oder IEnumerable zurück.

2. Nur innerhalb Iterator Block.
3. Unsafe Blocks nicht erlaubt.
4. Parameter fpr Methode, Operator oder Accessor können nicht ref oder out sein.
5. Nicht in anonymen Methoden erlaubt.

6 Sicher und Unsicheren code

Spezieller Modus : unsafe. Erlaubt pointers. Es ist aber nicht verifiziertbar und admin Rechte sind nötig.
Plnvoke mechanismus? GC + Pinning.

Man kann :

1. Pointers deklarieren und benutzen.
2. Konvertierungen zwischen Pointers und Basistypen machen.
3. Nehme Adressen von variablen und weitere c++ sachen.

Listing 30: Unsafe Code Example

```

class TestPointer
2   {
    public unsafe static void Main()
    {
        int[] list = {10, 100, 200};
        fixed(int *ptr = list)
7
        /* let us have array address in pointer */
        for ( int i = 0; i < 3; i++)
        {
            Console.WriteLine("Address of list[{0}]=<1>",i,(int)(ptr + i));
12           Console.WriteLine("Value of list[{0}]=<1>", i, *(ptr + i));
        }
        Console.ReadKey();
    }
17 }

```

6.1 Pointer Operations

Pointer Element access	<code>char *p</code>
Adress-of Operator	<code>unsafe int i; &l</code>
Pointer Arithmetik	<code>char *p; p++;</code>
sizeof Operator	<code>sizeof(byte)</code>
stackalloc	<code>stackalloc int [10]</code>
Fixed statement	<code>fixed</code>

6.2 Pinning Objects

Listing 31: Pinning Objects

```

public void FixVar2()
{
    string strName = "Fixing Variables";

```

```

4 unsafe
{
    fixed( char* pStr = strName )
    {
        for ( int i = 0; pStr[i] != '\0'; i++)
9 System.Diagnostics.Trace.Write( pStr[i] );
    }
}

14 //The fixed statement prevents relocation of a variable by the garbage collector
    in
    //unsafe contexts only.

```

7 Reflection

7.1 Reflection Grundkonzepte

Metadata

Einzige Ort für Typ Info und Code

Code ist innerhalb Typinfo beinhaltet.

Jede Objekt kann für seinen Typ abgefragt werden.

Metadata von Typen kann mit Reflection entdeckt werden.

Dynamic Type System

Sehr dynamisch und Spracheunabhängig.

Typen können erweitert werden und gebaut werden zur Laufzeit.

“On the fly” erzeugung von Assemblies (dlls).

7.2 Type Info zur Laufzeit

7.3 Laden von Assemblies

Listing 32: Laden von Assemblies

```

    Assembly a = Assembly.LoadFrom("HelloWorld.exe");
    Type[] types = a.GetTypes();
    foreach (Type t in types)
4 Console.WriteLine(t.FullName);

    Type hw = a.GetType("Hello.HelloWorld");
    MethodInfo[] methods = hw.GetMethods();
    foreach (MethodInfo m in methods)
9 Console.WriteLine(m.Name);

```

System.Type

Zugriff auf Metadata für alle net Types.

Rückgabewert von Object . GetType()

zugriff auf Methoden Konstruktoren , Parametern, Feldern, Properties,Argumente,Atributen,Events,Delegates,Nam

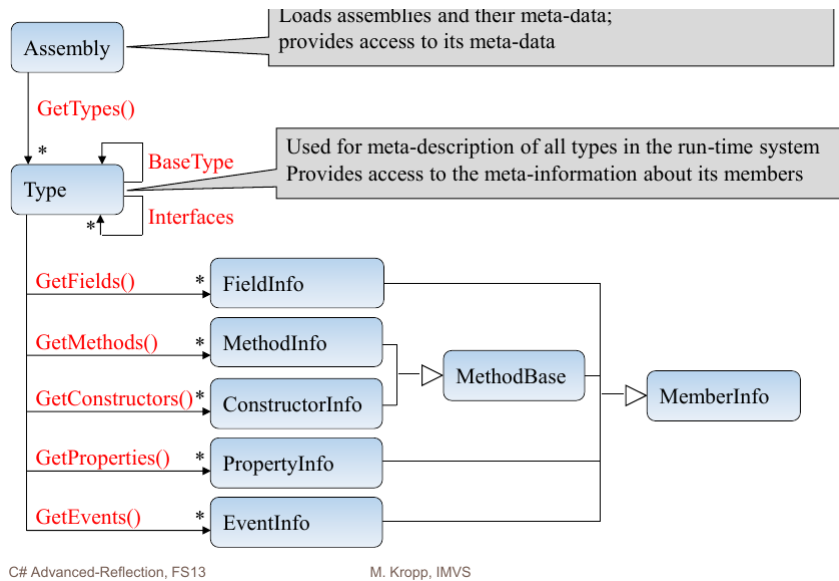


Abbildung 9: Typ Info zur Laufzeit

7.4 Innerhalb Typinfo

Members Entdecken [MemberInfo] GetMembers(), FindMembers()

Felder + Properties Entdecken [MemberInfo] GetFields(), PropertyInfo : GetProperties()

Konstruktoren Entdecken GetConstructors(), GetMethods(), GetEvents()

7.4.1 MemberInfo

- Base class for all "member" element descriptions
 - ▣ Fields, Properties, Methods, etc.
- Provides member kind, name and declaring class

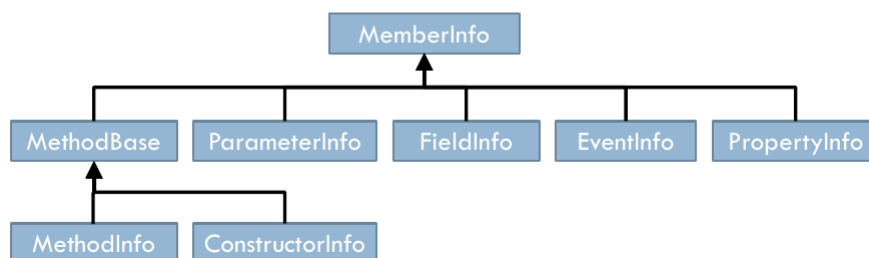


Abbildung 10: Memberinfo Diagramm

- Types know their Module, Modules know their types
- Modules know their Assembly and vice versa
- Code can browse and search its entire context

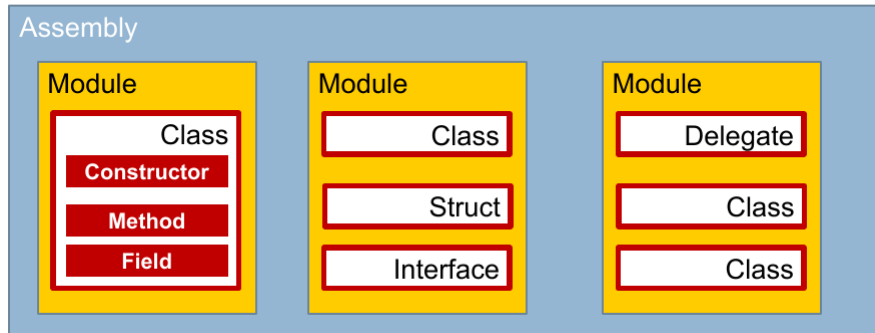


Abbildung 11: Macht von Reflexion

7.5 Bigger Picture

8 Attributes

- Benutzerdefinierte MetaInfo über Programmelemente.
- Kann an alles geklebt werden.
- Kann zur Laufzeit abgefragt werden
- Erweitert vorherdefinierte Attributen wie public sealed oder abstract.
- Benutzt von CLR Servixes (serialization , COM usw)

8.1 Benutzer definierte Attributen

Listing 33: Benutzerdefinierte Attributen

```

    /// causes compiler to bring message, that a class C is
    /// obsolete
    public class ObsoleteAttribute : Attribute {
    public string Message { get {...} }
5  public bool IsError { get {...} set {...} }
    public ObsoleteAttribute() {...}
    public ObsoleteAttribute(string msg) {...}
    public ObsoleteAttribute(string msg, bool error) {...}
    }

10  //Positional Parameter : of attr constructor

    [Obsolete( Message Use class C1 instead", true)]
    public class C {... }

15  //Name Parameter : public property of attr class

    [Obsolete(IsError=true, Message= Use class C1 instead")]
    public class C {... }

20  //Mixed
    [Obsolete( Use class C1 instead , IsError=true)]
    public class C {... }

```


Listing 34: Querying Attributes

```
class CommentAttribute : Attribute {
    string text, author;
    public string Text { get {return text;} }
    public string Author { get {return author;} set {author = value;} }
5 public Comment (string text) { this.text = text; author = "HM"; }
}
[Comment("This is a demo class for Attributes", Author="XX")]
class C { ... }
class Test {
10 search should
    static void Main() {
        also be continued
        Type t = typeof(C);
        in subclasses
15 object[] a = t.GetCustomAttributes(typeof(Comment), true);
        Comment ca = (Comment)a[0];
        Console.WriteLine(ca.Text + ", " + ca.Author);
    }
}
```

8.2 AttributeUsage

```
Describes how user-defined attributes are to be used
public class AttributeUsageAttribute : Attribute {
    public AttributeUsageAttribute (AttributeTargets validOn) {...}
5 public bool AllowMultiple { get; set; }
    // default: false
    public bool Inherited { get; set; }
    // default: true
    public AttributeTargets ValidOn { get; set; } // default: All
10 public virtual Object TypeId { get; }
}
validOn to which program elements is the attribute applicable?
AllowMultiple can it be applied to the same program element multiple times?
Inherited is it inherited by subclasses?
15 TypeId when implemented, gets unique identifier for derived attribute classes

Usage
[AttributeUsage(AttributeTargets.Class |
20 AttributeTargets.Interface, AllowMultiple=false)]
public class MyAttribute : Attribute {...}
```

9 LINQ + Lambda

9.1 Lambda

Drawback of anonymous methods :

- Method must be implemented
- Anonymous methods are hard to read.
- Restricted usage

Listing 35: some basic lambda examples

```
(parameters) => expression-or-statement-block
3
List<Point> list = list.FindAll((p) =>
```

```

{
    return (p.X * p.Y > 100000);
8 });

    delegate int Transformer(int i);
    We could assign and invoke the following
    lambda expression
13 Transformer square = x => x*x; // Lambda expression

    x => { return x * x; };

    Lambdas are mostly used with FunAction delegate
18 Func <int, int> sqr = p => p * p;
    Action showValue = i => Console.WriteLine(i);

```

The compiler converts these expressions into anonymous delegate methods or an expression tree.

Listing 36: lambdas and functions

```

Func <string, string, int> totalLength =
(s1, s2) => s1.Length + s2.Length;
//Explicitly specified parameter types
4 Func <int, int> sqr = (int p) => p * p;
//Otherwise the compiler uses type inference to determine that x is an int
Func <int, int> sqr = p => p * p;

```

9.1.1 Lambda Execution Context

- Lambdas execute in the context of their appearance.
- Therefore, they can use the values of the variables that are defined in that context.
- A lambda expression can reference the local variables and parameters of the method in which it is defined (**outer variables**).

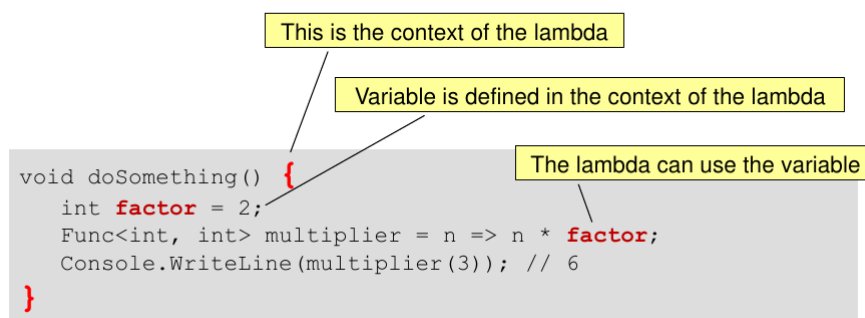


Abbildung 12: Lambda Execution Context

9.2 LINQ

9.3 Advantages and bla bla

- Type Safe,
- SQL Similarities,
- Query Set and Transform Operations for .net-

- The outer variable is **captured** by the lambda expression
- Deferred Execution
 - ▢ Lambda's are executed when they are invoked
- Captured variables are evaluated when the lambda is **invoked**

```
void doSomething() {
    int factor = 2;

    // define the lambda expression
    Func<int, int> multiplier = n => n * factor;
    factor = 10;

    // call the lambda expression
    Console.WriteLine(multiplier(3)); // what's the result?
}
```

Deferred execution

Abbildung 13: Figure

- Works with all types of data (xml,objects,db)
- Integrated part of .net Languages , enables language-based querying.

Listing 37: basic LINQ Examples

```
//Given a collection
names = new List<String> { Tom ", Dick ", Harry ",
3 Mary ", Jay }
//Search all names in list containing a
var results = from n in names
where n.Contains( a ")
select n;
8 // Sort list
ar results = from n in names
orderby n
select n;
```

9.3.1 LINQ General Syntax

- Query begins with from clause, and ends with select or group clause.
- Clause Examples : Where , Orderby join, let , from , into

9.3.2 LINQ Key Features

1. Deferred Execution. Retrieve, specific value, Iterate through a collection perform a specific operation.
2. Compile time syntax and schema checking.
3. LINQ is abstracted from underlying data. Consistent across various data sources.

9.3.3 Query Chaining

```
var query = from name in names
where name.Contains a
3 orderby name.Length
select name.ToUpper();
//filter->sorter->projector
```

9.3.4 Formulating LINQ Expressions

Method or Fluent Syntax	var usCustomers =
<pre> names .Where(n=>n.Contains(a));</pre>	
<hr/>	
<pre>2 where n.Contains(a) select n;</pre>	<pre>var usCustomers = from name in names</pre>
<hr/>	
Query Comprehension Syntax	Hybrid Syntax
<pre> from c in Comedians select c).Count();</pre>	<pre>var results = (</pre>

9.3.5 LINQ and objects

If an object supports the IEnumerable interface it is compatible with LINQ to Objects provider.

9.3.6 Local vs Interpreted Query

Local IEnumerable + Delegates

```
public static IEnumerable<TSource> Where<TSource>(  
    this IEnumerable<TSource> source,  
3 Func<TSource, bool> predicate)
```

Interpreted Queryable + Expression Trees

```
1 public static IQueryable<TSource> Where<TSource>(  
    this IQueryable<TSource> source,  
    Expression<Func<TSource, bool>> predicate)
```

9.3.7 Feature Summary

10 Dynamic Programming

10.1 Dynamic vs Static Languages

Dynamic 1. Simple and succinct.

2. Implicit Typing
3. Meta programming
4. No compilation.

Static

- Robust
- Performing
- Intelligent tools,
- Scalable

LINQ Language Features Summary

Hochschule für Technik

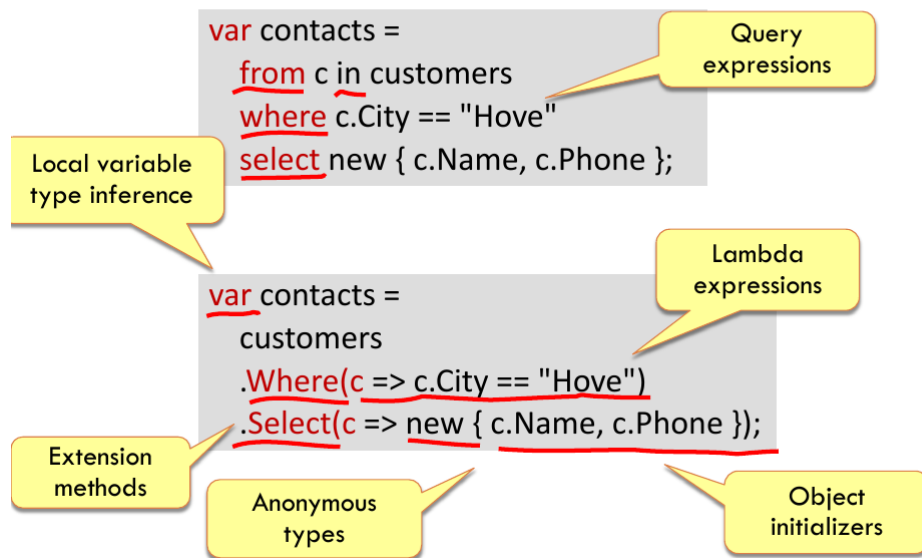


Abbildung 14: Figure



Abbildung 15: Figure

10.2 Dynamic Language Architecture

10.3 DLR Concepts

Expression Trees Represent language semantics; enables creation of new dynamic languages. Includes control flow, assignment, other language-modeling nodes.

Call-Site Caching Optimization for faster dynamic binding.

Dynamic Object Interop Dynamic binding and dispatching objects and method calls

DLR runs on top of CLR as additional layer

10.4 Using Dynamic

Listing 38: using dynamic

```
1 //Static
  ScriptObject calc = GetCalculator();
  object res = calc.Invoke("Add", 10, 20);
  int sum = Convert.ToInt32(res);
  //Dynamic
6 dynamic calc = GetCalculator();
  int sum = calc.Add(10, 20);
  dynamic x = 1;
  dynamic y = "Hello";
  dynamic z = new List<int> { 1, 2, 3 };
```

Member selection deferred to runtime, dynamic substituted at runtime. Static type is “dynamic”

10.5 Implementing Dynamic

Listing 39: dynamic object example

```
    class Duck : DynamicObject{
    public override bool TryInvokeMember(
    InvokeMemberBinder binder, object[] args,
4 out object result)
    {
    Console.WriteLine(binder.Name + " method was called");
    }
    }
9 return true;
    static void Main(string[] args){
    dynamic duck = new Duck();
    duck.Quack();
    duck.Waddle();
14 }
```

10.5.1 API

Call Dynamic Methods

```
public virtual bool TryInvokeMember(InvokeBinder
binder, Object[] args, out Object result)
```

```
public virtual bool TryGetMember(GetMemberBinder
2 binder, out Object result)
```

Get Properties

```
public virtual bool TryInvokeMember(InvokeBinder
2 binder, Object[] args, out Object result)
```

10.6 Dynamic Language Integration - Python

```
    static object Calculate(string expression)
2 {
    ScriptEngine engine = Python.CreateEngine();
    return engine.Execute(expression);
    }
    static void Main(string[] args)
7 {
    dynamic result = Calculate("2 * 3");
```

```
Console.WriteLine("result: {0}", result);
}
```

10.6.1 Data Passing

in ScriptScope.SetVariable.

out ScriptEngine.GetVariable.

10.7 COM Interop

Listing 40: COM Interop Example

```
using Word = Microsoft.Office.Interop.Word;
var word = new Word.Application();
word.Documents.Add();
4 Word._Document doc = word.ActiveDocument;
word.Selection.Text = "Demo test";
word.Selection.Paste();
doc.SaveAs(@"test");
doc.Close();
9 word = null;
```

Features

- Optional and Named Params.
- Indexed Properties
- Optional Ref modifiers.
- Interop Type Embedding.

10.8 Native Code

Can be done in one of two ways : Explicit P/Invoke (DLL Import) or Implicit P/Invoke (C++ Interop).

Listing 41: Explicit PInvoke example

```
// Assume that we have the following Win32
function (in C)
int MessageBox (HWND hWnd, LPTSTR lpText, LPTSTR lpCaption, UINT
uType);
5 //We can invoke it from C#
using System;
System.Runtime.InteropServices;
class Test {
[DllImport("user32.dll")]
10 static extern int MessageBox(uint hWnd, string text, string
caption, uint type);
// no body
static void Main() {
int res = MessageBox(0, "Isn't that cool?", "", 1);
15 Console.WriteLine("res = " + res);
}
}
```

```
[DllImport ( dll.name
    [, EntryPoint=function-name]
    [, CharSet=charset-enum]           // for string marshaling
    [, ExactSpelling= (true|false)]    // should name be modified acc.
                                        // to CharSet?
    [, SetLastError= (true|false)]    // true: returns Win32 error
                                        // info
    [, CallingConvention=callconv-enum]
)]
```

charset-enum: CharSet.Auto (default), CharSet.Ansi, CharSet.Unicode, ...

callconv-enum: CallingConvention.StdCall (default),
CallingConvention.FastCall, ...

Abbildung 16: DLL Import Structure

10.8.1 DLL Import Structure

10.8.2 Parameter Marshaling

- Primitive types of C# (int, float, ...) are automatically mapped to Win32 types.
- Standard mapping can be modified as follows:

```
using System.Runtime.InteropServices; Assume that we want to pass a class Time and a
structure S to a native method
...
[DllImport("...")]
static extern void Foo (
    [MarshalAs(UnmanagedType.LPStr)] string s, int x
);
```

ANSI string of bytes

- LPWStr Unicode string
- LPTStr default: Windows XP/NT/2000 => Unicode,
Windows 98 => ANSI

Abbildung 17: Parameter Marshalling

10.8.3 Passing Structs

- Assume that we want to pass a class Time

```
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Sequential, Pack=4)] // members are
// allocated sequentially
class Time {
    public ushort year; // and are aligned at 4 byte boundaries
    public ushort month;
    ...
}
```

- And the call

```
[DllImport("...")] static extern void Foo(Time t);
[DllImport("...")] static extern void
    Foo([MarshalAs(UnmanagedType.LPStruct)] Time t);
Foo(new Time());
```

Abbildung 18: Passing Structs

10.9 Implicit P Invoke C++

Problems : Custom types Byte alignment Memory Management Pointers Passing managed allocated memory.

Listing 42: Implicit PInvoke

- Assume that we want to pass a struct S

```
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Explicit)] // members are allocated at
// specified offsets
struct S {
    FieldOffset(0) ushort x;
    FieldOffset(2) int y;
}
```

- And the call

```
[DllImport("...")] static extern void Bar(S s);
Bar(new S());
```

Abbildung 19: Figure

```
[StructLayout ( layout-enum // Sequential | Explicit | Auto
[, Pack=packing-size] // 0, 1, 2, 4, 8, 16, ...
[, CharSet=charset-enum] // Ansi | Unicode | Auto
)]
```

Abbildung 20: Figure

```
// vcncppv2_impl_dllimp.cpp
2 // compile with: /clr:pure user32.lib
using namespace System::Runtime::InteropServices;
// Implicit DllImport specifying calling convention
extern "C" int __stdcall MessageBeep(int);
```

11 Diagnostics and Garbage Collection

11.1 General

1. Allocate the object and forget about it - Automatic Memory management.
2. If the heap does not have enough space for the new object, garbage collection occurs.
3. Always call Dispose() on any object you directly create if the objects supports IDisposable.

11.2 Object Creation

```
Car c = new Car( Viper , 200, 100);
IL
IL_000c: newobj instance void CilNew.Car::.ctor (string, int32, int32)
```

Tasks for CIL new object instruction are : Calculating total memory, examining the managed heap to ensure that there is enough room. Return the referece to the caller, and then advance the "next object pointer" to point at the next available object on the heap.

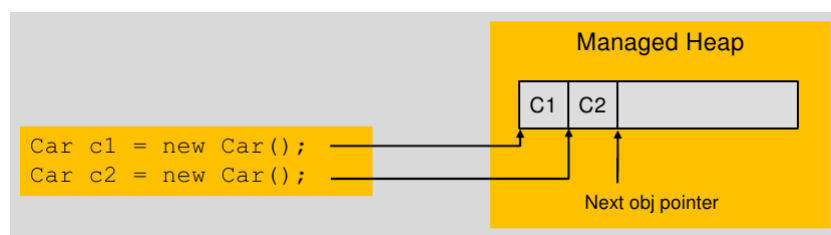


Abbildung 21: Garbage Collection with managed Heap

11.3 GC in .net

Tracing garbage collector. Determines reachable objects and then discards all the unreachable ones. Does not interfere with object access, but is a thread that wakes up intermittently and calculates the object graph - reachability.

11.3.1 Reachability

Reachable Objects reachable from roots via references, otherwise garbage.

Roots Distinguished set of objects always reachable. Examples are static and global variables, locals and arguments on the stack, CPU Registers. Otherwise known as **Strong references**

Determining Reachability

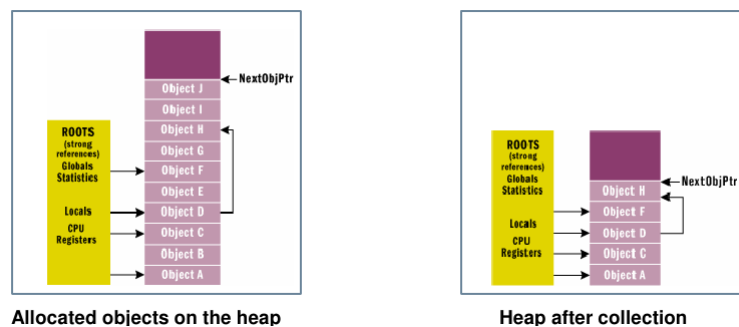


Abbildung 22: Figure

11.3.2 Cleaning up objects

Listing 43: c# finalise

```
1 protected override void Finalize()
{
    try { /* code */ }
    finally { base.Finalize(); }
}
```

- Prior to an object being released its finalizer is called.
- Overriding finalise can perform any memory cleanup for the type.
- Finalise is called in the following cases **this call is non deterministic** :
 - natural garbage collection
 - GC.collect method.
 - Application Domain is being unloaded from memory.
 - CLR being shutdown

11.3.3 The collection process

11.3.4 Finalisation guidelines

1. Override in case of unmanaged resources (c++ pinvoke)

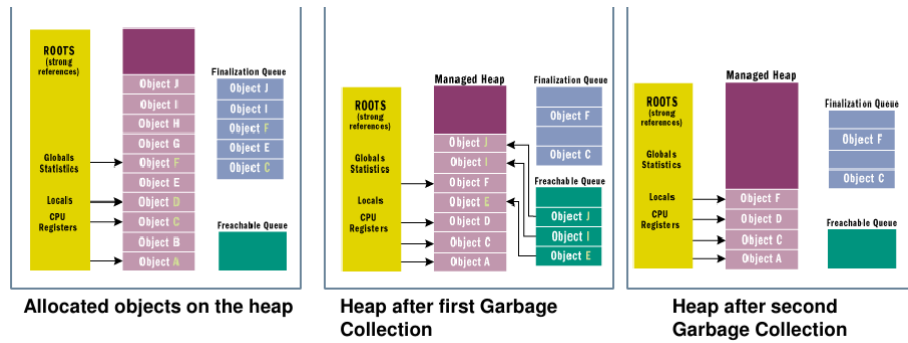


Abbildung 23: Garbage collection Process

2. Dont use unnecessarily.
3. make finalised objects as small as possible
4. dont use referenced fields in finalised objects.
5. dont access above field in finalised method.

11.4 Explicit Resource Management

Some objects require explicit tear down. IDisposable interface is made for this purpose. A programmer must exclusively call dispose method of object to perform cleanup. Structures and classes support IDisposable, finalise is only overrideable on classes. Can also be used along with finalise.

Listing 44: IDisposable example

```
// Implementing IDisposable
public class MyResourceWrapper : IDisposable
{
    4 // The object user should call this method
    // when they finished with the object.
    public void Dispose()
    {
        // Clean up unmanaged resources here.
    9 // Dispose other contained disposable objects.
    }
}
```

11.4.1 Semantics

1. Disposed = beyond redemption. No reactivation, call on disposed object throws disposed exception.
2. calls to dispose can be repeated
3. A containing object should call dispose of all its children.

11.4.2 Dispose Pattern

Scope Classes that are not sealed and require resource cleanup.

Purpose The pattern has been designed to ensure reliable, predictable cleanup, to prevent temporary resource leaks, and to provide a standard, unambiguous pattern for disposable classes. It also aids subclasses to correctly release base class resources.

Listing 45: Dispose Pattern Implementation

```
// thread-safe wrapper of some unmanaged handle:
public sealed class OSHandle : IDisposable
3 {
    private bool disposed;
    public OSHandle(IntPtr h) { handle = h; disposed = 0; }
    public void Dispose()
    {
6      Dispose(true);
      GC.SuppressFinalize(this);
    }
    ~OSHandle() {
      Dispose(false);
13 }

    void Dispose(bool disposing)
    {
        if (!disposed){
18 }
        // clean-up:
        if (disposing) {
            /* safe to access reference fields here */
        }
23 disposed = true;
        // dispose unmanaged resources here
    }
    base.Dispose(disposed);
}
```

Using statements automatically call dispose on all objects within brackets of the statement, at the end.

11.4.3 Finalise vs Dispose

Finalise Called by GC , not deterministic, fully auto, Used to free memory,

Dispose Deterministic,explicit call,Free resources - file locks etc.

11.5 System.GC

Interaction with garbage collector. Use when creating types that use unmanaged resource.

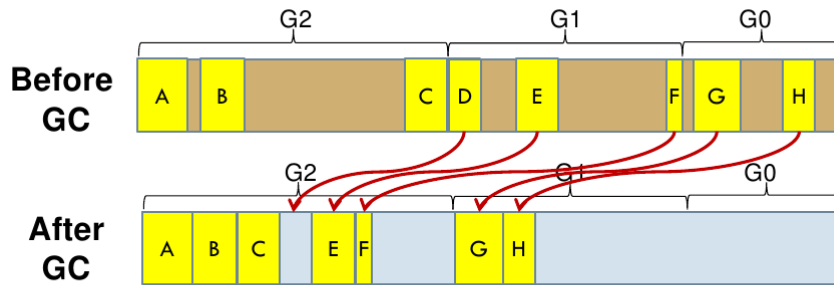
11.6 GC Explained

Divided into two Phases. Detection and reclamation. Detection is when objects are no longer used, reclamation is a release for memory use. Each phase has various techniques and algorithms.

Garbage collection steps

1. Detection : searches for managed objects that are referenced in managed code. (mark)
2. Reclamation : finalise unreachable objects. (sweep)
3. Free unmarked objects and reclaim memory. (sweep)

□ Generational Collection



□ Large Object Heap



Abbildung 24: Garbage collector visualised

12 Assembly and CIL

12.1 Role of Assemblies

1. Reuse of Code - smallest executable unit.
2. Type boundry - types are usually unique.
3. Versionable Units.
4. Contain all type info - self describing.
5. Cofngurable can be stored anywhere.

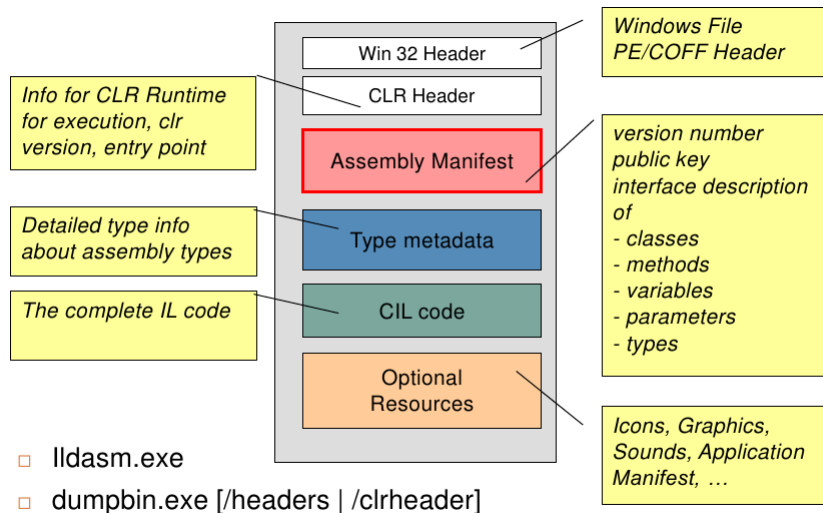


Abbildung 25: Assembly Content

12.2 Tools

Compilers csc.exe or vbc.exe.

IL assembler ilasm.exe

IL disassembler ildasm.executable

Assembly Management gacutil.executable

12.3 Assembly Info

Use assembly class.

Currently executing Assembly GetExecutingAssembly()

Assembly of original entry method GetEntryAssembly()

Assembly that called current function GetCallingAssembly()

Assembly Attributes assemblyinfo.cs or Reflection API

```
GetCustomAttributes
(
    typeof
    (
        AssemblyTitle
    ),
    false)
;
```

Application Manifest Assembly info for OS. Represented with external file or usually embedded.

Satellite Assemblies external for resources.

12.4 Managing Private Assemblies

Usually stored in local bin dir, if not found gives exception. However can be stored anywhere provided it has been configured in App.Config.

```
<configuration>
<runtime>
3 <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <probing privatePath="MyLibraries"/>
</assemblyBinding>
</runtime>
</configuration>
```

12.5 Make assembly available publicly

Location: C:\Windows\assembly (before .Net 4.0)

C:\Windows\Microsoft.NET\assembly (after .Net 4.0)

The directory

C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Accessibility\
2.0.0.0__b03f5f7f11d50a3a

- Install with the GACUtil (gacutil.exe)
 - > gacutil **-i** GlobalLib.dll
- Remove assemblies from the GAC
 - > gacutil **-u** GlobalLib
- Only **signed** assemblies can be installed (shared via GAC)
 - ▣ Assembly has to be signed with a **strong name**

Abbildung 26: Figure

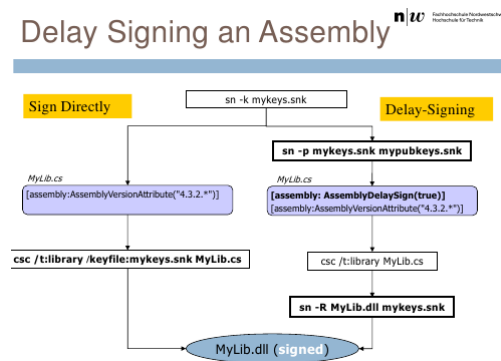


Abbildung 27: Figure

12.5.1 Signing Assembly

12.6 What is CIL

Intermediate language between high level and machine code. Lowest level before machine code/ similar to java byte code. Open standard , machine independant. Advantages : Language interop - CIL code different languages, one program. OS Portability.

12.7 CIL

12.7.1 Hello world example

```

1 .assembly extern mscorlib{...}
2 .assembly FirstGlance{...}
3 .module FirstGlance.dll
4 .class public auto ansi beforefieldinit HelloWorld
5 extends [mscorlib]System.Object
6 {
7 .method public hidebysig instance void
8 SayHello() cil managed
9 {
10 .maxstack 8
11 ldstr "Hello World"
12 call void [mscorlib]System.Console::WriteLine(string)
13 ret
  
```

```
}
}
```

12.7.2 CIL Programming model

All operations are stack not register based. Locals and incoming parameters live on stack. Stack is of arbitrary size.

Names : everything has to be fully qualified.

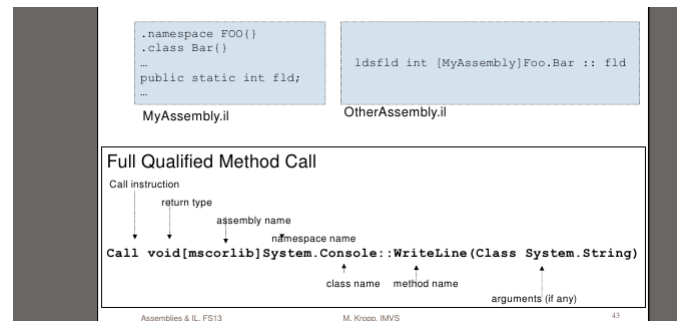


Abbildung 28: IL Types

12.7.3 Execution State

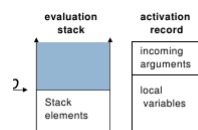


Abbildung 29: CIL Stack

Activation Record holds all per activation data of method. Used to implement method calls. Contains : Arguments and local variables. Activation of method by a "call" instruction causes a new activation record to be created

Evaluation Stack Operations performed on this stack. Stack used to store information just before the execution of a statement. It has 3 operations push, pop and perform

Logical Stack Elements of stack are not of particular size. Depth = number of elements on stack.

Stack δ Instruction in IL to change depth of stack.

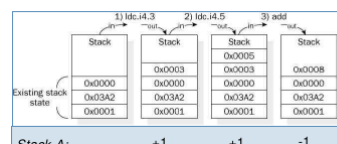


Abbildung 30: Stack Example operation

12.7.4 Base Instruction Set

Load and store Instructions Push pop.

Operate Instructions Arithmetic, logical, type conversion.

Branching and Jumping Boolean vals, labels and jumps.

- Data kind and load instructions in IL

Data kind	Opcode	Comment
static field	ldsfld	Op-arg: Type Class :: field-name
local variable	ldloc	Op-arg: local variable ordinal
method argument	ldarg	Op-arg: formal argument ordinal
array element	ldelem.*	Suffix: specialized for element type

- The numbering of local variables defaults to the order of declaration in the .locals section (we will discuss later)
- "ldloc.0" through to "ldloc.3"

Abbildung 31: Loading and storing

- Data kind and load instructions in IL

Data kind	Opcode	Comment
static field	stsfld	Op-arg: Type Class :: field-name
local variable	stloc	Op-arg: local variable ordinal
method argument	starg	Op-arg: formal argument ordinal
array element	stelem.*	Suffix: specialized for element type

Abbildung 32: Loading and Storing 2

Arithmetic and logical instructions

- Applicability of add instructions

Instruction	int32	int64	Nfloat	nfloat is nativeFP format
add	V	V	V	Unchecked addition
add.ovf	V	V	----	Signed overflow check
add.ovf.un	V	V	----	Unsigned overflow check

- Applicability of div and rem instructions

Instruction	int32	int64	Nfloat	nfloat is nativeFP format
div	V	V	V	signed division
div.un	V	V	----	unsigned division
rem	V	V	V	signed remainder
rem.un	V	V	----	unsigned remainder

Abbildung 33: Operating Instructions

12.7.5 Names and Directives in CIL

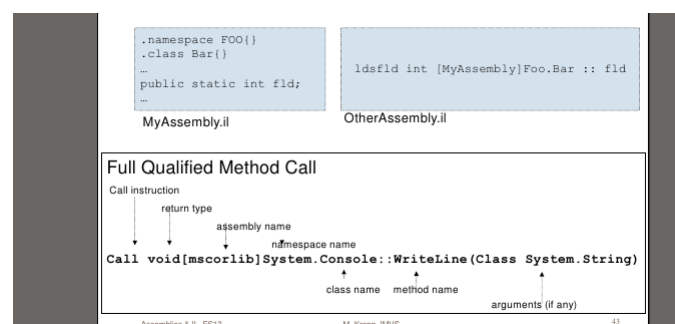


Abbildung 34: Names in CIL

12.7.6 Value Classes /Structs

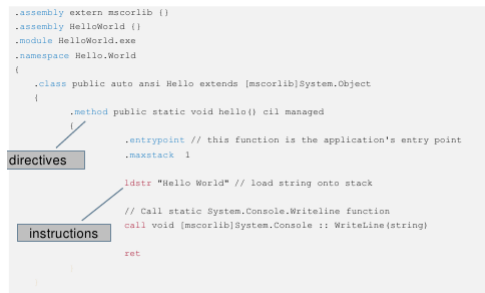


Abbildung 35: Directives and Names

```

//In c#:
public struct ValCls {public int I,j,k}
4
//MSIL:
.class value public auto ansi sealed ValCls
extends [mscorlib]System.ValueTypes
{
9 .field public int32 I;
  .field public int32 j;
  .field public int32 k;
}

```

12.7.7 Class Declarations

```

ClassDecl .class ClassHeader { {MemberDecl} }
2 ClassHeader {ClassAttr} ident [extends TypeRef]
[implements TypeRef { , TypeRef }]
MemberDecl <Member Declaration>
  class members: fields, methods, properties, events
  .class public auto ansi Hello extends [mscorlib]System.Object
7 {
  ...
}

```

Class Declaration Attribs	private	Default - Class and Members not visible outside assembly. Cannot be used with p
	public	Opposite of private.
	abstract	Cannot be instantiated.
	interface	interface definition.
	sealed	cannot be extended.
	ansi	ansi marschalling for strings - default
	autochar	auto for character strings
	unicode	unicode for character strings
	auto	layout of class determined automatically
	explicit	layout of class is explicit
	sequential	sequential layout of class.

12.7.8 Defining Methods

```

// MethodDeclaration
.method MethodHead { {MethodBody} }
// MethodHead
{ MethodAttr } [CallConv] TypeRef DottedName ( [Parameter
5 { , Parameter} ] {ImplAttr}
  .method public static void hello() cil managed
  {
  ...
  }
}

```

12.7.9 Method bodies

- **.locals declaration:**

LocalDecl \rightarrow `.local` '(' LocalSignature { ',' LocalSignature } ')'

Body Item	Description
<code>.entrypoint</code>	This method is the application entry point.
<code>.locals</code>	Declares local variables for this method.
<code>.maxstack</code>	Specifies the maximum stack height.
<code>.line</code>	Specifies a source line number.
<code>instruction</code>	An IL instruction
<code>label</code>	A code label in the IL

Abbildung 36: Method bodies

13 Concurrency

13.1 Basic Concepts

Multithreading Use of multiple Threads

Concurrency Order in which multiple tasks execute is not determined.

Parallelism True simultaneous multicore execution

Types of Asynchrony:

IO Bound Async Usually wait for response Goal is thread efficiency.

Computing Bound async Responsive foreground behaviour during computing intensive tasks.

13.2 .Net Async Concepts

Async methods almost like sync code. Unifies comp, network and IO Async. Need for a more responsive UI.

Threads Low level.

TPL Parallel programing.

13.3 Threads vs Tasks

Task Future / Promise. `Task<T>` returns a T when task is done but not now.

Thread One of many ways to fill promise above. *Not every task needs a thread.* Sometimes the task only registers a callback, fully automated decision on whether thread is needed or not.

Listing 46: task example"

```
// Step 1
private Task<int> ComputeValueAsync(int start, int count)
{
    var random = Task.Run(() =>
        ComputeValue(start, count));
}
return random;

//Step 2
int sum = await ComputeValueAsync(Start, Count);
```

```

        Console.WriteLine("Result (synchronously) : {0}", sum);

//Step 3
15 public async void DoLongRunningTask()
    {
        int sum = await ComputeValueAsync(Start, Count);
        Console.WriteLine("Result (synchronously) : {0}",
            sum);
20 }

```

13.4 Task class API

Task.Run(someDelegate) Static method starts a task, has many overloads.

Task.Wait waits for task to complete.

Task<T> = Task.Run(someDelegate) T is returning type of delegate.

TaskCompletionSource Another class to start and control task execution.

Additionally there is cancellation and progress reporting.

13.4.1 Task Based Async Pattern /TAP

- Returns a running (“hot”) Task or Task<T>
- Has an “Async” suffix
- Is overloaded to accept a cancellation token and/or IProgress<T>, if needed
- Returns quickly to the caller
- Does not tie up a thread if I/O bound

13.4.2 Task Combinators

Task.WhenAll or Task.WhenAny

Listing 47: Combinator Task example

```

    async Task<int> Delay1() { await Task.Delay(1000); return 1;}
    async Task<int> Delay2() { await Task.Delay(2000); return 2; }
    async Task<int> Delay3() { await Task.Delay(3000); return 3; }
4 Task<int> winningTask = await Task.WhenAny(Delay1(), Delay2(),
    Delay3());
    Console.WriteLine("Done");
    Console.WriteLine("Winning task {0}", await winningTask);

```

13.5 Low Level Threading API

Important classes : Thread, ThreadPool, Threadstate, ThreadPriority, Monitor.

Listing 48: Simple Thread Example

```

usingusing System;
2 System.Threading;
class Printer {
    char ch;
    int sleepTime;
    public Printer(char c, int t)

```

```

7 {ch = c; sleepTime = t;}
  public void Print() {
    for (int i = 0; i < 100; i++) {
      Console.Write(ch);
      Thread.Sleep(sleepTime);
12 }
  }
  }
  class Test {
    static void Main() {
17 Printer a = new Printer('.', 10);
    Printer b = new Printer('*', 100);
    new Thread(new ThreadStart(a.Print)).Start();
    new Thread(new ThreadStart(b.Print)).Start();
    }
22 }

```

13.5.1 Background Threads

Two different type of threads , foreground and background. As long as a foreground thread is running, the program will not terminate. *Background threads do not stop the program from terminating*

Listing 49: background thread example

```

    Thread bgThread = new Thread(new ThreadStart(...));
2  bgThread.IsBackground = true;
    bgThread.Start();

```

13.5.2 Passing Data to Threads

Using lambdas or delegates. Beware of captured data.

Listing 50: data passing low level thread

```

1  static void Main() {
    string msg = "Hello ";
    Thread t = new Thread ( () => Print (msg + " from t!") );
    t.Start();
  }
6  static void Print (string message) {
    Console.WriteLine (message);
  }

```

13.5.3 Thread Recycling with Threadpools

```

1  ThreadPool.QueueUserWorkItem(ThreadProc);
  void ThreadProc(object o)
  {
  }

```

□ **Define** Async delegate

- Note: same interface as synchronous method with same return type

```
public delegate string[] AsyncReadFileDelegate(string fileName);
```

□ **Execute** the delegate asynchronously

```
// create the asynchronous delegate
AsyncReadFileDelegate asyncCall = new AsyncReadFileDelegate(ReadFile);

// start the asynchronous execution of ReadFile
IAsyncResult ar = asyncCall.BeginInvoke("fileName", null, null);

...
// do your stuff here in the meantime
...

// wait for and get the results
string[] strReturnValue = asyncCall.EndInvoke(ar);
```

Abbildung 37: Async Delegates

Thread Pooling Automatic Thread Management. Limits max concurrent threads. Default 50. Also supports multicore technology. Async Delegates and Backgroundworker + TaskParallel and PLINQ

13.5.4 AsyncDelegates

14 Parallel Programming

High level mechanisms for parallel programming. Parallel, Task, PLINQ. Handles partitioning, parallel execution, collection of results.

14.1 Partitioning Strategies

Data Parallelism Divide data across multiple processors. Supported in PLINQ

Task Parallelism Smaller tasks that can be executed on different processors. (TPL - Parallel and Task)

14.2 Overview

Components	Partitions work	Collates results
PLINQ	Yes	Yes
Parallel class	Yes	No
Task parallelism	No	No

Abbildung 38: Parallel Overview

PLINQ and the Parallel class are useful whenever you want to execute operations in parallel and then wait for them to complete (structured parallelism). This includes non-CPU-intensive tasks such as

calling a web service. The **task parallelism** constructs are useful when you want to run some operation on a pooled thread, and also to manage a task's workflow through continuations and parent/child tasks.

Listing 51: PLINQ Example

```
IEnumerable<int> numbers = Enumerable.Range(3, 30);
var parallelQuery =
    from n in numbers.AsParallel()
    where Enumerable.Range(2, (int)Math.Sqrt(n)).All(i =>
5 n % i > 0)
    select n;
    return parallelQuery.ToArray();
```

14.3 Parallel Considerations

Not in the same order that they are committed. AsParallel.AsOrdered() forces order
AsUnordered() more efficient.

Enforcing Parallel - Only parallel when auto sees benefits.

.WithExecutionMode(ParallelExecutionMode.ForceParallelism)

14.4 Task Parallel Library

Listing 52: TPL example

```
//parallel for
2 Parallel.For(0, 10, (x) => Console.WriteLine(x));
List<string> capitals = new List<string>()
{ "London", "Paris", "Berlin", "... " };
//parallel for each
Parallel.ForEach(capitals, (x) =>
7 Console.WriteLine(x));

//Or
Parallel.Invoke(
    () => DoSomeWork(),
12 () => DoSomeOtherWork() );
```

Parallel Class provides Implicit task execution. Executes each of the possible tasks in parallel and waits for them to finish. *These are action Delegates that do not return results*

Task Class Explicit Execution. Managing unit of work. or with generic parameter for Task that returns result of specified type. TaskScheduling manages scheduling of Tasks.

Listing 53: Task Example

```
1 Task [] tasks = new Task[] {
    Task.Factory.StartNew(() =>
        Console.WriteLine("Hello from taskA.")),
    Task.Factory.StartNew(() =>
        Console.WriteLine("Hello from taskB.")),
6 }
//Block until all tasks complete.
Task.WaitAll(tasks);

11 Task<double>[] tasks = new Task<double>[]
{
    Task.Factory.StartNew(() => DoComputation1()),
    Task.Factory.StartNew(() => DoComputation2()),
};
16 double[] results = new double[tasks.Length];
```

```
for (int i = 0; i < tasks.Length; i++)
    results[i] = tasks[i].Result;
```

15 Code Contracts

Listing 54: first example

```
1  public int Sqrt(double x)
   //Specify allowed
   {
       Contract.Requires( x >= 0);
       Contract.Ensures( 0 <= Contract.Result<double>());
6  return Math.Sqrt(x);
   //Specify guarantee}
   public string Substring( int startIndex ) {
       Contract.Requires( 0 <= startIndex );
       Contract.Requires( startIndex <= this.Length );
11 ...
   }

   public string Substring( int startIndex ) {
       ...
16 Contract.Ensures( Contract.Result<string>() != null );
   Contract.Ensures( Contract.Result<string>().Length ==
       this.Length      startIndex )
       ...
   }
```

Unit tests can be automatically generated from contracts. **Benefits :**

- Documentation
- Consisensess. (No code duplication)
- Inheritance - works with super and sub classes.
- Contracts used in abstract methods and interfaces.
- Also supports invariants.

15.1 Contract Principles

Preconditions verified when function starts. Also verified before function exists. Object invariants verified after every public function in a class. **All functions must be pure and not alter fields**

Contracts are static method calls.

Conditions are boolean expressions

Preconditions = Requirement on input state

Contract method is conditionally defined - Debug release scenario. **Members accessed in a precondition must be visible to all callers of method**

15.2 Precondition Guidelines

Preconditions should:

1. Be possible for client to easily validate.
2. Rely on data as accessible as method itself.

3. Always indicate a bug if violated.
4. Conditions true at exit of method - ensures.
5. Assertions can be made anywhere in code.

15.3 Object invariants

```

    [ObjectInvariantMethod]
    void ObjectInvariant()
3 {
    Contract.Invariant( x >= 0 );
    Contract.Invariant( y >= 0 );
}

```

Method is called after every public methods in class.

15.4 Quantifiers

```

    public static int Max( int[ ] a ) {
        Contract.Requires( a != null && 0 < a.Length );
3    Contract.Ensures(
        Contract.ForAll( 0, a.Length, i => a[ i ] <=
            Contract.Result<int>() ) );
        Contract.Ensures(
            Contract.Exists( 0, a.Length, i => a[ i ] ==
8        Contract.Result<int>() ));
        int res = Int32.MinValue;
        for( int i = 0; i < a.Length; i++ )
            if( a[ i ] > res )
                res = a[ i ];
13    return res;
    }

```

15.5 Contract inheritance

All contracts are inherited by subtypes. Overriding methods should not declare additional preconditions. Subclasses *may declare additional invariants*.

15.6 Contracts and Interface

```

    [ContractClass(typeof(ContractForICalculator))]
    interface ICalculator {
        double Sqrt(double x);
    }
5 [ContractClassFor(typeof(ICalculator))]
    abstract class ContractForICalculator : ICalculator {
        // must use explicit implementation
        double ICalculator.Sqrt(double x)
        {
10    Contract.Requires(x >= 0);
            return 0;
            // dummy value to satisfy compiler
        }
    }

```

15.7 Static analysis

Move run time errors to compile time. (Project Properties).