using System; //namespace delcaration. A namespace is a ollection

//of classes. (For knowing the compiler the Console class.)

Class Program

{

Static void Main()

{

Console.WriteLine(„Welcome to C# training!”); //Console is a class. Writeline is a method/function

} // belonging to this class.

}

A namespace is a collection of classes, interfaces, structs, enums and delegates etc.

If we don’t use Using System expr. Then we must type in main() System.Console.Writeline(””);

**Reading and writing to the console**

using System;

Class Program

{

Static void Main()

{

Console.WriteLine(„Enter user’s name”)

string Username= Console.ReadLine();

Console.WriteLine(„Enter last name”)

string Lastname= Console.ReadLine();

Console.WriteLine(„Good Morning”+ Username); //Concatenation of two strings

//Alternative syntax to the above line is:

Console.WriteLine(„Good Morning (0)”, Username);

Console.WriteLine(„Good Morning (0), (1)”, Username, Lastname);

//C# is case sensitive

}

}

using System;

Class Program

{

Static void Main()

{

Bool b= true;

Console.WriteLine(„Min= (0)”, Int.MinValue); //shows the min value of an int type

Console.WriteLine(„Max= (0)”, Int.MaxValue);

//float and double are floating point types

}

}

using System;

Class Program

{

Static void Main()

{

String Name= ”\”Pako\””; //Escape sequence characters. The quot. marks will be also printed.

//due to use of \.

String Name = @”C: [\\Pako\\DotNet\\CSharp](file:///\\Pako\\DotNet\\CSharp)” //All the characters after @ will be printed (verbatim literal).

Console.WriteLine(Name);

}

}

**Operators in C#**

using System;

Class Program

{

Static void Main()

{

int numerator= 10;

int denominator= 2;

int Result= numerator/denominator; // % is for remainder of division

Console.WriteLine(”Result= (0)”, Result); // = is the assignment operator

/ \* Comparison operators: ==, !=, >, >=, <, <=

Conditional operators: &&,||

Ternary operator: ?:

Null coalescing operator: ??

Arithmetic operators: +,-, \*, /, %

Int Number = 15;

bool IsNumber10 = Number==10? True: false; /\*This line is equivalent to all the if conditions below.

bool IsNumber10;

If (Number == 10)

{ IsNumber10 = true;

}

else

{ IsNumber10 = false;

}

Console.WriteLine(”Number = 10 is (0)”, IsNumber);

}

}

**Nullable datatypes-null coalescent operator**

using System;

Class Program

{

Static void Main()

{

bool? AreYouMajor= null; /\* ? makes the bool type nullable

if (AreYouMajor == true)

{

Console.WriteLine(”User is Major”);

}

Else if (AreYouMajor == false)

{

Console.WriteLine(”Not Major”);

}

Else

{ Console.WriteLine(”User did not answer”);

}

}

}

using System;

class Program

{

static void Main()

{

int? TicketsOnSale = null;

int AvailableTickets;

If (TicketsOnSale == null)

{

AvailableTickets = 0;

}

Else

{

AvailableTickets = (int)TicketsOnSale /\*Use of cast. (a nullable int to a non nullable int.

}

Console.WriteLine(”Available tickets = (0)”, AvaibleTickets);

}

}

Alternative equivalent to the above would the following with the null coalescing operator ??

using System;

Class Program

{

Static void Main()

{

Int? TicketsOnSale = null;

Int AvailableTickets = TicketsOnSale ?? 0; /\* it means if TicketsOnSale is null use the value 0 for Availabletickets. Else use the integer value of TicketsOnSale for assignment.

Console.WriteLine(”Available tickets = (0)”, AvaibleTickets);

}

}

**Datatypes conversion**

using System;

class Program

{

static void Main()

{

float f = 1.045;

int i = (int)f; /\*Convert float to int with the type cast operator

/\*Alternativ explicit data type conversion is to use the Convert class with some of its methods

Int i = Convert.ToInt32(f);

Console.WriteLine(i);

}

}

using System;

class Program

{

static void Main()

{

string strNumber = ”100”;

int i = int.Parse(strNumber); /The string will be converted to int

Console.WriteLine(i);

string strNumber = ”100tg”;

int Result= 0;

bool IsConversionSuccessful= int.TryParse(strNumber, out Result);

if (IsConversionSuccessful)

{

Console.WriteLine(Result);

}

else

{

Console.WriteLine(”Please enter a valid number”);

}

/\*Parse() method throws an exception if it cannot parse the value, whereas TryParse() returns a bool indicationg whether it succeeded or failed. So use Parse() if you are sure the value will be valid, otherwise use TryParse().

}

}

Int [] EvenNumbers= new int[3]; /\* Declares the size of the array.

**The switch statement**

using System;

Class Program

{

Static void Main()

{

Switch (UserNumber)

{

Case 10:

Console.WriteLine(”UserNumber = 10”);

break;

Case 20:

Console.WriteLine(”UserNumber = 20”);

break;

Case 30:

Console.WriteLine(”UserNumber = 30”);

Break;

default:

Console.WriteLine(”UserNumber is not 10, 20 or 30”);

break;

case 10:

case 20:

case 30:

Console.WriteLine(”UserNumber = (0)”, UserNumber);

Break;

default:

Console.WriteLine(”UserNumber is not 10, 20 or 30”);

break;

}

}

}

using System;

Class Program

{

Static void Main()

{ Start: /\*start is a label for the code to refer to later using

goto.

Console.WriteLine(”1- small, 2- Medium, 3- Large”);

Int UserChoice = int.Parse(Console.ReadLine());

Switch (UserChoice)

{

Case 1:

Cost += 1;

Break;

Case 2:

Cost += 2;

Break;

Case 3:

Cost += 3;

Break;

Default:

Console.WriteLine(”Invalid entry (0)”, UserChoice);

Goto start;

}

Decide:

Console.WriteLine(”Wollen Sie einen ander Kaffee kaufen?”);

String UserDecision= Console.ReadLine();

Switch (UserDecision)

{

Case ”Ja”:

Goto start;

Case ”Nein”:

Break;

Default:

Console.WriteLine(”Invalid entry (0)”, UserChoice);

}

Console.WriteLine(”Bill amount = (0)”, cost);

} }

**While loop in C#**

using System;

Class Program

{

Static void Main()

{

Console.WriteLine(„Bitte die Nummer eingeben”); //shows the min value of an int type

Int BenutzerNum= int.Parse(Console.ReadLine());

Int Beginn= 0;

While (Beginn<= BenutzerNum)

{ Console.WriteLine(Beginn);

//Console.Write(Beginn + „ ”); will print the numbers on the same line with space between

Beginn= Beginn + 2;

}

} }

**Do while loop**

using system;

class Program {

int main () {

Console.WriteLine(„Bitte die Nummer eingeben”); //shows the min value of an int type

int BenutzerNum= int.Parse(Console.ReadLine());

int Beginn= 0;

while (Beginn <= BenutzerNum){

Console.Write(Beginn + „ ”);

Beginn = Beginn + 2;

}

do {

console.WriteLine(„Wollen Sie fortsetzen?);

BenutzerAntwort = Console.ReadLine().Toupper();

If (BenutzerAntwort != „JA” && BenutzerAntwort != „NEIN”)

{

Console.WriteLine(„Ungultiger Antwort! Bitte antworten ja oder nein.”);

}

} while(BenutzerAntwort != „JA” && BenutzerAntwort != „NEIN”);

} while(BenutzerAntwort == „JA”);

}

}

**For and foreach loops** **Tutorial 15**

**Methods in C#**

using system;

class Program {

public static void main () { //main ist eine static method.

Program p = new Program(); //Schaffen eine Klasse Objekt. To invoke an instance method it must be done by an instance of the class it belongs to. Thus create an instance of the class program p.

p.Nummern();

}

public void Nummern() //Nummern ist eine instance method.

{

Int Beginn=0;

While (Beginn<=20)

{

Console.WriteLine(Beginn);

Beginn = Beginn + 2;

}

}

}

using system;

Class Program {

public static void main () { //main ist eine static method.

Program.Nummern(30); //Since Nummern is static method it is invoked using the class name.

Program P = new Program();

Int Sume =P.Add(10, 20);

Console.WriteLine(„Sum= {0}”, Sum);

}

Public int Add(int EN, int SN)

{

Return EN+SN;

}

Public static void Nummern(int Limit)

{

Int Beginn=0;

While (Beginn<= Limit)

{

Console.WriteLine(Beginn);

Beginn = Beginn + 2;

}

Why have instance and static methods? Multiple instances of a class can be created and each instance can have its own separate method. Instead when a method is static there are no instances of that method, and one can invoke only that one definition of the static method.

**Method parameters in C#**

using system;

Class Program {

public static void main () {

int i = 0;

EinfachMethode(i);

// EinfachMethode(ref i); invoke the method passing param. By reference.

Console.WriteLine(i);

}

public static void EinfachMethode(int j) //Pass parameter by value.

// public static void EinfachMethode(ref int j) pass parameter by reference

{

J= 101;

}

}

When passing parameters to methods as above by value i and j point to different memory locations. Operations on one variable will not affect the value of the other variable.

When passing parameters to methods by reference i and j are pointing to the same memory location. Operations on one variable will affect the value of the other variable.

using System;

class Program

{

Public static void main()

{

Int Total = 0;

Int Product= 0;

Calculate(10,20, out Total, out Product);

Console.WriteLine(„Sum = {0} && Product = {1}”, Total, Product);

}

Public static void Calculate( int EN, int SN, out int Sum, out int Product) /out parameters are used when a method returns more than one value.

{

Sum= EN + SN;

Product= EN \* SN;

}

}

using System;

class Program

{

Public static void main()

{

Int[] Nummern = new int[3];

Nummern[0]= 101;

Nummern[1]= 102;

Nummern[3] = 103;

ParamsMethode();

}

Public static void ParamsMethode(params int [] Nummern) //The params keyword makes optional the invoke of ParamsMethode with int array optional. It can be called without passing int array on the line above.

{

Console.WriteLine(„Es gibt {0} Elemente”, Nummern.Length);

Foreach(int i in Nummern)

{

Console.WriteLIne(i);

}

}

}

The params keyword lets specify a method parameter that takes a variable number of arguments. You can send a comma- separated list of arguments, or an array, or no arguments. Params keyword should be the last one in a method declaration and only one params keyword is permitted in a method declaration.

**Namespaces in C#**

using System;

//Alternatively using ProjectA.TeamA;

//using Project.TeamB;

class Program

{

public static void main()

ProjectA.TeamA.ClassA.Print();

// ClassA.Print();

ProjectA.TeamB.ClassA.Print();

}

{

Namespace ProjectA; //Create namespace

{

Namespace TeamA

{

ClassA

{

Public static void print()

{

Console.WriteLine(„GruppeA Print Methode”);

}

}

}

}

Namespace ProjectA; //Create namespace

{

Namespace TeamB

{

ClassA

{

Public static void print()

{

Console.WriteLine(„GruppeB Print Methode”);

}

}

}

}

Alternative way to call the functions of the two teams

Using System;

Using PATA = ProjectA.TeamA;

Using PATB =ProjectA.TeamB;

class Program

{

public static void main()

PATA.ClassA.Print(); // Resolves the ambiguity whose Team’s ClassA Print function is called. PATA and PATB are namespace aliases directives.

PATA.ClassA.Print();

}

Namespaces can be nested in 2 ways. This is the other one.

Namespace ProjectA. TeamB; //Create namespace

{

ClassA

{

Public static void print()

{

Console.WriteLine(„GruppeB Print Methode”);

}

}

}

**Introduction to classes**

Using System;

Class Customer

{

String \_ersteName;

String \_letzteName; //Constructor with no arguments

Public Customer() : this(„Keine Erste Name”, „Keine Nachname”)

{

}

Public Customer (string ErsteName, string LetzteName) //This is the constructor. It never returns variables and has the same name as the class.

{

This.\_ersteName = ErsteName; //Initialize the variables of the class.

This.\_letzteName = LetzteName;

}

Public void PrintvolleName()

{

Console.WriteLine(„Volle Name = {0}, this.\_ersteName + „ ” + this.\_letzteName);

}

~Customer() //destructors. Optional.

{

//Clean up code.

}

}

Class Program

{

Public static void Main()

{

Customer C1= new Customer(„Jayardana”,”Pura”);

C1.PrintvolleName();

Customer C2= new Customer(„J”,”P”);

C2.PrintvolleName();

}

} //Constructors can be overloaded by the number and type of parameters.

**Static and instance class members**

Using system;

Class Circle{

Float \_PI = 3.141F;

Int \_Radius;

Static Circle()

{

Circle .\_PI=3.141F; //\_PI is initialized here and only declared above (float \_PI;) For a static constructor like this access modifieres need not be used (public, private). Static constr. Are called only once because they initialize members used by all objects and static constr. Are called before instance constructors.

}

Public Circle(int Radius)

{

This.\_Radius = Radius;

}

Public static void Print()

{

/\*this Print method because it is static cannot be accessed by per object reference as C1.Print or C2.Print but it must be accessed by the class’ name. So Circle.Print();

}

Public float Flächeverschätzen()

{

Return this.\_PI\*this.\_Radius\*this.\_Radius;

}

}

Class program{

Public static void main()

{

Circle C1 = new Circle(5);

Float Fläche1 = C1.Flächeverschätzen();

Console.WriteLine(„Fläche = {0}”, Fläche1);

Circle C2 = new Circle(6);

Float Fläche2 = C2.Flächeverschätzen();

Console.WriteLine(„Fläche = {0}”, Fläche2);

\_PI and \_Radius are instance fields. That means whenever an object circle is created like C1 and C2 seperate memory is allocated and \_PI and \_Radius are assigned to different memory Locations for each object. If some method or variable does not change on a per object basis it should be marked as static (like \_PI and Flächeverschätzen method).

Float \_PI = 3.141F;

}

}

**Inheritance in C# Tutorial 20**

Using System;

Public class Arbeitnehmer

{

String VorName;

String NachName;

String Email;

Public void PrintVolleName()

{

Console.Writeline(VorName + „ „ + NachName);

}

}

Public class VollZeitArbeitnehmer : Arbeitnehmer

{

Public Float jährlichesGehalt;

}

Public class TeilZeitArbeitnehmer : Arbeitnehmer

{

Public Float UhrGehalt;

}

Public class Program

{

Public static void Main()

{

VollZeitArbeitnehmer VZA = new VollZeitArbeitnehmer();

VZA.VorName = „Jayardana”;

VZA.NachName = „Pura”;

VZA.PrintVolleName()

VZA.JährlichesGehalt();

TeilZeitArbeitnehmer TZA = new TeilZeitArbeitnehmer();

TZA.VorName = „Jayardana”;

TZA.NachName = „Pura”;

TZA.PrintVolleName()

TZA.UhrGehalt();

}

}

}

Pillars of OOP 1) Inheritance 2) Encapsulation 3) Abstraction 4) Polymorphism

Public class A: TeilZeitArbeitnehmer

{

//Class A inherits all the features from TeilZeitArbeitnehmer

}

C# supports only single class inheritance.

C# supports multiple interface inheritance.

Child class is a specialization of base class.

Base classes are automatically instantiated before child class constructor.

Using system;

Public class ElternKlasse

{

Public ElternKlasse()

{

Console.WriteLine(„ElternKlasse Kontruktor geruft”);

}

Public ElternKlasse(string Bericht)

{

Console.WriteLine(Bericht);

}

Public class KinderKlasse: ElternKlasse

{

Public KInderKlasse(): base(„Text”) //Choose which of the two constructors is called of the ElternKlasse.

{

Console.WriteLine(„KinderKlasse Kontruktor geruft”);

}

}

Public class program

{

Public static void Main()

{

KinderKlasse KK = new KinderKlasse();

}

}

**Method hiding in C#**

Using system;

Public class Arbeitnehmer

{

String VorName;

String NachName;

String Email;

Public void PrintVolleName()

{

Console.Writeline(VorName + „ „ + NachName);

}

}

Public class TeilZeitArbeitnehmer: Arbeitnehmer

{

Public new void PrintVollName() //With the new keyword one hides the base class method i.e. the printVollName of the base class Arbeitnehmer.\*/

{ base.PrintVollName(); /\*This calls the base class method PrintVollName

Console.Writeline(VorName + „ „ + NachName + „-Contractor.”);

}

}

Public class VollZeitArbeitnehmer: Arbeitnehmer

{

}

Public class program {

Public static void main() {

VollZeitArbeitnehmer VZA = new VollZeitArbeitnehmer();

VZA.ErsteName= „VollName”;

VZA.NachName = „Arbeitnehmer”;

VZA.PrintVollName();

((Arbeitnehmer)VZA).PrintVollName(); /\*Alternative method to call the base class PrintVollName().\*/

TeilZeitArbeitnehmer TZA = new TeilZeitArbeitnehmer();

VZA.ErsteName= „TeilZeit”;

VZA.NachName = „Arbeitnehmer”;

VZA.PrintVollName();

Arbeitnehmer TZA = new TeilZeitArbeitnehmer(); /\*This is a third way to call a base class method. The

TZA.ErsteName= „TeilZeit”; child object TZA can fulfil all the operations of its

TZA.NachName = „Arbeitnehmer”; parent class Arbeitnehmer. Hence the

TZA.PrintVollName(); function here is the base class method PrintVollName of Arbeitnehmer class.\*/

}

}

Different ways to invoke a hidden base class member from derived class

1. Use the base keyword
2. Cast child type to parent type and invoke the hidden member
3. ParentClass PC = new ChildClass() PC.HiddenMethod()

**Polymorphism in C#**

Using system;

Public class Arbeitnehmer

{

Public String VorName =”ErN”;

String NachName = „NachN”;

String Email;

Public virtual void PrintVollName()

{

Console.Writeline(VorName + „ „ + NachName);

}

}

Public class TeilZeitArbeitnehmer: Arbeitnehmer

{

Public override void PrintVollName()

{

Console.Writeline(VorName + „ „ + NachName + „-Teil Zeit”);

}

}

Public class VollZeitArbeitnehmer: Arbeitnehmer

{

Public override void PrintVollName()

{

Console.Writeline(VorName + „ „ + NachName + „-Voll Zeit”);

}

}

Public class VorübergehendZeitArbeitnehmer: Arbeitnehmer

{

Public override void PrintVollName()

{

Console.Writeline(VorName + „ „ + NachName + „Vorübergehend”);

}

}

Public class program

{

Public static void main()

{

Arbeitnehmer[] Viele\_Arbeitnehmer = new Arbeitnehmer[4];

Viele\_Arbeitnehmer[0] = new Arbeitnehmer();

Viele\_Arbeitnehmer[1] = new VollZeitArbeitnehmer();

Viele\_Arbeitnehmer[2] = new TeilZeitArbeitnehmer();

Viele\_Arbeitnehmer[4] = new VorübergehendArbeitnehmer();

Foreach(Arbeitnehmer Arb in Viele\_Arbeitnehmer)

{

Arb.PrintVollName();

}

}

}

Polymporphism is the capability to invoke derived class methods using a base class reference variable at run time. In the base class the method is declared virtual, and in the derived class we override the same method. The virtual keyword indicates, the method can be overridden in any derived class.

**Method overriding and method hiding**

Public class BaseClass

{

Public virtual void Print()

{

Console.WriteLine(„Base Class Print Method”);

}

}

Public class DerivedClass : BaseClass

{

Public override void Print()

{

Console.WriteLine(„Derived Class Print Method”);

}

}

Public class Program

{

Public static void Main()

{

BaseClass B = new DerivedClass();

B.print();

}

} /\*In method overriding a base class reference variable pointing to a child class object, will invoke the overridden method in the Child class.\*/

Public class BaseClass

{

Public virtual void Print()

{

Console.WriteLine(„Base Class Print Method”);

}

}

Public class DerivedClass : BaseClass

{

Public new void Print()

{

Console.WriteLine(„Derived Class Print Method”);

}

}

Public class Program

{

Public static void Main()

{

BaseClass B = new DerivedClass();

B.print();

}

} /\*In method hiding a base class reference variable pointing to a child class object, will invoke the hidden method in the Base class.\*/

**Method overloading Tutorial 25**

Using system;

Public class Program

{

Public static void Main()

{

Public static void Zufugen(int NUM\_1, int NUM\_2)

{

Console.WriteLine(„Zufugung= {0}”, NUM\_1 +NUM\_2);

}

Public static void Zufugen(int NUM\_1, float NUM\_2, float NUM\_3)

{

Console.WriteLine(„Zufugung= {0}”, NUM\_1 +NUM\_2);

}

Public static void Zufugen(int NUM\_1, int NUM\_2, int NUM\_3, out int NUM\_4)

{

Console.WriteLine(„Zufugung= {0}”, NUM\_1 +NUM\_2);

NUM\_4 = NUM\_1 + NUM\_2;

}

}

} /\* Function or method overloading are equivalent terms. Method overloading allows a class to have multiple methods with the same name, but with a different signature. So in C# functions can be overloaded based on the number, type (int, float etc.) and kind (value, Ref or out) of parameters. The signature of a method consists of the name of the method and the type, kind (value, reference, or output) and the number of its formal parameters. The signature of a method does not include the return type and the params modifier. So it is not possible to overload a function based on the return type or params modifier. \*/

**Why properties in C#**

Using system;

Public class Student

{

Private int \_id;

Private string \_Name;

Private int\_passMark= 35;

Public void SetName(string Name)

{

If (string.IsNullOrEmpty(Name))

{

Throw new Exception(„Name cannot be null or empty”);

}

This.\_Name = Name;

}

Public string GetName()

{

If (string.IsNullOrEmpty(this.\_name))

{

return „No Name”;

}

Else

Return this.\_Name;

/\* This is equivalent to return string.IsNullOrEmpty(this.\_Name) ? „No Name” : this.\_Name;

}

Public void SetId(int Id)

{

If (Id <= 0)

{

Throw new Exception(„Student Id cannot be negative”);

}

This.\_id =Id;

}

Public int GetId()

{

Return this.\_id;

}

Public class Program

{

Public static void Main()

{

Student C1 = new Student();

C1.SetId( -101);

Console.WriteLine(„Student Id = {0}”, C1.GetId());

}

}

**Properties in C#**

Using system;

Public class Student

{

Private int \_id;

Private string \_Name;

Private int\_passMark= 35;

Public string City {get; set;} /\*This is auto-implemented properties. The compiler creates a private

Anonymous field that can only be accessed through the property’s get and

set accessors. Reduces excessive code.\*/

Public int PassMark

{

Get

{

Return this.\_PassMark;

}

}

Public string Name

{

Set

{

If (string.IsNullOrEmpty(value))

{

throw new Exception(„Name cannot be null or empty”);

}

this.\_Name = value;

}

Get

{

Return string.IsNullOrEmpty(this.\_Name) ? „No Name” : this.\_Name;

}

}

Public int Id

{

Set

{

If (value <= 0)

{

Throw new Exception(„Student Id cannot be negative”);

}

This.\_id = value;

}

get

{

return this.\_id;

}

}

}

Public class Program

{

Public static void Main()

{

Student C1 = new Student();

C1.Id = 100;

C1.Name = „Mark”;

C1.PassMark = 34;

Console.WriteLine(„Student Id = {0}”, C1.Id);

Console.WriteLine(„Student Name = {0}”, C1.Name);

Console.WriteLine(„PassMark = {0}”, C1.PassMark);

}

} /\*In C# to encapsulate and protect fields one uses properties. Get and set accessors to implement properties. A property with get and set accessor is a Read/Write property. A property with only a get accessor is a Read only property. A property with only a set accessor is a Write only property.\*

**Structs in C#**

public struct {

private int \_id;

private string \_name;

public string Name

{

get { return this.\_name; }

set {\_name = value; }

}

Public int ID {

get { return this.\_id; }

set{this.\_id = value; }

}

}

public Kunde (int Id, string Name) {

this.\_id = Id;

this.\_name= Name;

}

Public void Print\_Details() {

Console.WriteLine(„Id= {0} && Name= {1}, this.\_id, this.\_name);

}

public class Project {

public static void Main() {

Kunde K1 = new Kunde(101, „Max”);

K1.PrintDetails();

Kunde K2 = new Kunde();

K2.PrintDetails();

K2.ID = 103;

K2.Name = „ Johan”;

K2.PrintDetails();

Kunde K3 = new Kunde

{

ID = 104,

Name = „Reichard”

};

}

}

}

**Classes vs Structs**

* A struct is a value type where a class is a reference type.
* Structs are stored on stack, where as classes are stored on the heap.
* Value types hold their value in memory where they are declared. However reference types hold a reference to an object in memory.
* Value types are destroyed immediately after the scope ends. Whereas for reference types only the reference variable is destroyed after the scope ends. The object is later destroyed by the garbage collector.
* When on copies a struct into another struct, a new copy of that struct gets created and modifications on one struct will not affect the values contained by the other struct.
* When a class is copied into another class, only a copy of the reference variable is created. Both of the reference variables point to the same object on the heap. So operations on one variable will affect the values pointed to by the other reference variable.
* Structs can’t have destructors, but classes can have.
* Structs cannot have explicit parameter less constructor where as a class can.
* Struct can’t inherit from another class where as a class can. Both structs and classes can inherit from an interface.
* A class or a struct cannot inherit from another struct. Struct are sealed types. [Use sealed keyword to seal classes.

**Interfaces Tutorial 30**

Interface keyword. Like classes they contain properties, methods, delegates or events but only the declarations not implementations of those.

Interface members are public by default and they don’t allow explicit access modifiers. Interfaces cannot contain fields.

Whenever a class or a struct inherits from an interface, it must provide implementation for all interface members. Otherwise a compile time error occurs.

A class or a struct can inherit from more than one interface at the same time yet a class cannot inherit from more than one class at once.

Interfaces can inherit form other interfaces. A class that inherits one interface must provide implemementation for all interface members in the entire interface inheritance chain.

One cannot create an instance of an interface but an interface reference variable can point to a derived class object.

using System;

Interface I\_Kunde

{

void Ausdruck();

}

Interface I2

{

void Ausdruck2();

}

Class Kunde : I\_Kunde, I2

{

public void Ausdruck()

{

Console.WriteLine(„Interface Ausdruck Methode”);

}

public void I2\_Ausdruck()

{

Console.WriteLine(„I2\_Ausdruck”);

}

}

/\* Interface I2: I\_Kunde

{

void Ausdruck2();

}

Class Kunde : I2

{

public void Ausdruck()

{

Console.WriteLine(„Interface Ausdruck Methode”);

}

public void I2\_Ausdruck()

{

Console.WriteLine(„I2\_Ausdruck”);

}

} \*/

public class Projet

{

public static void Main()

{

Kunde K1 = new Kunde();

K1.Ausdruck();

I\_Kunde1 Vem = new Kunde(); //Upcasting.

Vem.Ausdruck2();

}

}

**Explicit interfaces implementation**

Using System;

interface I1

{

void Metodo();

}

interface I2

{

void Metodo();

}

public class Proje : I1, I2

{

void I1.Metodo()

{

Console.WriteLine(„I1 Metodo”);

}

void I2.Metodo() // Explicit interface member implementation

{

Console.WriteLine(„I2 Metodo”);

}

public static void Main()

{

Proje P = new Proje();

((I1)P).Metodo(); // Alternative syntax I1 i1 = new Proje();

((I2)P).Metodo(); //... I2 i2 = new Proje();

...................... i1.Metodo();

i2.Metodo();

When a class explicitly implements an interface member, the interface member ca no longer be accessed through class reference variable, but only through the interface reference variable.

Access modifiers are not allowed on explicitly implemented interface members.

Default implementation of an interface member.

public class Proje : I1, I2

{

public void Metodo() //**Default implementation** of interface method.

{

Console.WriteLine(„I1 Metodo”);

}

void I2.Metodo() // **Explicit interface member implementation**

{

Console.WriteLine(„I2 Metodo”);

}

public static void Main()

{

Proje P = new Proje();

P.Metodo(); // The default method implementation is accessible through

((I2)P).Metodo(); a class instance.

**Abstract classes in C#**

The abstract keyword is used to create abstract classes. An abstract class cannot be instantiated, it cannot be sealed. It may contain abstract members (methods, properties, indexers or events). A non-abstract class derived from an abstract class must provide implementation for all inherited abstract members.

If a class inherits an abstract class: 1) it must provide implementation for all the abstract members inherited from the base abstract class.

2) If it does not provide the implementation for the base abstract class it must be marked an abstract class as well.

An abstract class may contain itself implementation for its members!

using System;

abstract class Kunde

{

public abstract void Ausdruck();

}

public /\* abstract\*/ class Projet : Kunde

{

Public override void Ausdruck() //Must use the override keyword.

{

Console.WriteLine(„ ”);

}

public static void Main()

{

Kunde K = new Projet();

K.Ausdruck();

**Differences between abstract classes and interfaces**

Interfaces cannot have fields whereas an abstract class can have fields. An interface can inherit from another interface only and cannot inherit from an abstract class, whereas an abstract class can inherit from another abstract class or another interface. A class can inherit from multiple interfaces at the same time, whereas it cannot inherit from multiple classes at the same time. Abstract class members can have access modifiers whereas interface members cannot have access modifiers.

**Problems of multiple class inheritance**

The diamond problem.

Base class is A. Class B and class C inherit from class A.

Class D inherits from B and C. If a method in D calls a method defined in A and does not override that method, while B and C have overriden that method differently then from which class does D inherit? B or C?

class A {

public virtual void Metodo() {//... }

}

class B: A {

public override void Metodo() { //...}

}

class C: A {

public override void Metodo() { //...}

}

class D: B, C { } // ?

**Multiple class inheritance using interfaces Tutorial 35**

using System;

interface IA

{

void Amethode();

}

class A : IA

{

public void AMethode()

{

Console.WriteLine(„A”);

}

}

interface IB

{

Void Bmethode();

}

class B : IB

{

public void BMethode()

{

Console.WriteLine(„B”);

}

}

class AB : IA, IB

{

A a = new A();

B b = new B();

public void Amethode()

{

a.AMethode();

}

public void BMethode()

{

b.BMethode();

}

}

class Program

{

public static void Main()

{

AB ab = new AB();

ab.Amethode();

ab.Bmethode();

}

}

**Delegates in C#**

Recall structures in C# are value types whereas classes, delegates and interfaces are reference types.

A delegate is a type safe function pointer.

using System;

public delegate void GruesseDelegate(string Bericht); /\*Similar to a method signature. It can point to any function

with the same signature. \*/

class Programme

{

public static void Main()

{

GruesseDelegate del = new GruesseDelegate(Gruesse);

Del(„Gruesse von Delegate!”);

}

public static void Gruesse(string strBericht)

{

Console.WriteLine(strBericht);

}

}

The signature of the delegate must match the signature of the function, the delegate points to, a compile error occurs. This is the reason delegates are called type safe function pointers.

A delegate is similar to a class object. One may create an instance of it, pass in the function name as a parameter to the delegate constructor, and it is to this function the delegate will point to. Delegates’ syntax is similar to function’s syntax with the delegate keyword added.

**Delegates usage in C# I**

using System;

using Syste.Collections.Generic;

class Program

{

public static void Main()

{

List<Arbeitsnehmer> arbneh = new List<Arbeitsnehmer>();

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Warnecke”, Gehalt = 1800, Arbeitserfahrung = 5});

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Gerhard”, Gehalt = 4000, Arbeitserfahrung = 4});

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Harold”, Gehalt = 6000, Arbeitserfahrung = 6});

Arbeitsnehmer.Promote(arbneh);

}

class Arbeinehmer

{

public int ID { get; set; }

public int Name { get; set; }

public int Gehalt { get; set; }

public int Arbeitserfahrung { get; set; }

public static void Promote(List<Arbeitnehmer> ArbeitnehmerList)

{

foreach(Arbeitnehmer arbeitnehmer in ArbeitnehmerList)

{

If(arbeitnehmer. Arbeitserfahrung >= 5)

{

Console.WriteLine(arbeitnehmer.Name + „promoted”);

}

}

}

}

**Delegates usage in C# II**

using System;

using Syste.Collections.Generic;

class Program

{

public static void Main()

{

List<Arbeitsnehmer> arbneh = new List<Arbeitsnehmer>();

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Warnecke”, Gehalt = 1800, Arbeitserfahrung = 5});

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Gerhard”, Gehalt = 4000, Arbeitserfahrung = 4});

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Harold”, Gehalt = 6000, Arbeitserfahrung = 6});

IsPromotable isPromo = new IsPromotable(Promote);

Arbeitsnehmer.Promote(arbneh, isPromo);

}

public static bool Promote(Arbeitnehmer arb)

{

if(arb.Arbeitserfahrung >= 5)

{

return true;

}

else

{

Return false;

}

}

}

delegate bool IsPromotable(Arbeitsnehmer arbeiter);

class Arbeinehmer

{

public int ID { get; set; }

public int Name { get; set; }

public int Gehalt { get; set; }

public int Arbeitserfahrung { get; set; }

public static void Promote(List<Arbeitnehmer> ArbeitnehmerList, IsPromotable IsEligibleToPromote)

{

foreach(Arbeitnehmer arbeitnehmer in ArbeitnehmerList)

{

If(IsEligibleToPromote(Arbeitnehmer))

{

Console.WriteLine(arbeitnehmer.Name + „promoted”);

}

}

}

}

Alternatively with a lambda expression

public static void Main()

{

List<Arbeitsnehmer> arbneh = new List<Arbeitsnehmer>();

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Warnecke”, Gehalt = 1800, Arbeitserfahrung = 5});

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Gerard”, Gehalt = 4000, Arbeitserfahrung = 4});

Arbneh.Add(new Arbeitsnehmer() { ID = 101, Name =”Harold”, Gehalt = 6000, Arbeitserfahrung = 6});

Arbeitsnehmer.Promote(arbneh, arb=> arb.Arbeitserfahrung >= 5 );

}

**Multicast delegates in C#**

A multicast delegate is a delegate that has references to more than one function. When you invoke a multicast delegate, all the functions the delegate is pointing to are invoked.

There are 2 approaches to create a multicast delegate.

+ or += to register a method with the delegate

- or -= to unregister a method with a delegate

A multicast delegate, invokes the methods in the invocation list, in the same order in which they are added.

If the delegate has a return type other than void and if the delegate is a multicast delegate, only the value of the last invoked method will be returned. Along the same lines if the delegate has an out parameter, the value of the output parameter will be the value assigned by the last method.

Interview question: Where does one use multicast delegates?

Multicast delegate makes implementation of the observer design pattern very simple. Observer pattern is also called publish or subscribe pattern.

using System;

public delegate void SamDele();

class Program

{

public static void Main()

{ //Multicast delegate

/\*1st way to create multicast delegate\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SamDele del1, del2, del3, del4;

del1 = new SamDele(SameMet1);

del2 = new SamDele(SameMet2);

del3 = new SamDele(SameMet3);

del4 = del1 + del2 + del3;

del4(); //Multicast delegate pointing to three functions.

//SamDele del = new SamDele(SamMet1); This points to one function.

//del();

\*/\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//Alternative multicast delegate assignment syntax

SamDele del = new SamDele(SamMet1);

del += SamMet2;

del += SamMet3; // Registers method 3 with the delegate.

del(); // Will invoke the three methods.

}

public static void SamMet1()

{

Console.WriteLine(„SamMeth1 invok”);

}

public static void SamMet2()

{

Console.WriteLine(„SamMeth2 invok”);

}

public static void SamMet3()

{

Console.WriteLine(„SamMeth3 invok”);

}

}

public delegate void SamDele(out int Integer);

class Program

{

public static void Main()

{

SamDele del = new SamDele(SamMet1);

del += SamMet2;

int DelOutPara= -1;

del(out DeleOutPara);

Console.WriteLine(„DeleOutPara = {0}”, DeleOutPara); /\*The value of the output parameter

will be the value assigned by the last method.

}

public static void SamMet1(out int Numero)

{

Numero = 1;

}

public static void SamMet2(out int Numero)

{

Numero = 2;

}

}

**Exception handling in C# Tutorial 40**

using System;

using System.IO;

class Exepzion

{

public static void Main()

{

try

{

StreamReader streRea = new StreamReader(„C:\Sample Files\Data.txt”);

Console.WriteLine(streRea.ReadToEnd());

streRea.Close(); //Closes the streamreader and realeases any system resources.

}

catch(Exception e) //Or catch(FileNotFoundException e) inherits indirectly from Exception class

{

Console.WriteLine(e.Message);

Console.WriteLine();

Console.WriteLine(e.StackTrace);

//Or Console.WriteLine(„Check if the file {0} exists, e.FileName);

}

catch(Exception e)

{

Console.WriteLine(e.Message);

}

finally

{

If (streRea != null)

streRea.Close();

}

}

}

Exception Handling

An exception is an unforeseen error that occurs when a program is running.

Examples:

Trying to read from a file that does not exist, throws FileNotFoundException.

Trying to read from a database table that does not exist, throws an SqlException.

Showing actual unhandled exception to the end user is bad for two reasons

1. Users will be annoyed as they are cryptic and does not make much sense to the end users.
2. Exceptions contain information, that can be used for hacking into one’ s application.

An exception is actually a class that derives from System.Exception class. The System.Exception class has several useful properties, that provide valuable information about the exception.

Message: Gets a message that describes the current exception.

Stack Trace: Provides the call stack to the line number in the method where the exception occured.

Releasing System Resources

Try- the code that can possibly cause an exception will be in the try block.

Catch-handles the exception.

Finally – Clean and free resources that the class was holding onto during the program execution. Finally block is optional. It is good practice to always release resources in the finally block, because finally block is guaranteed to execute, irrespectively whether there is an exception or not.

Specific exception will be caught before the base general exception, so specific exception blocks should always be on top of the base exception block. Otherwise a compile error occurs.

**Inner Exceptions in C#**

The InnerException property returns the Exception instance that caused the current exception.

To retain the original exception pass it as a parameter to the constructor, of the current exception.

Always check if inner exception is not null before accessing any property of the inner exception object, else, you may get Null Reference Exception.

To get the type of the InnerException use GetType() method.

using System;

using System.IO;

class InnerException

{

public static void Main()

{ try {

try {

Console.WriteLine(„Gebe die erst Nummer ein:”);

int FN = Convert.ToInt32(Console.ReadLine());

Console.WriteLine(„Gebe die zweite Nummer ein:”);

int SN = Convert.ToInt32(Console.ReadLine());

int Resultat = FN / SN;

Console.WriteLine(„Resultat = {0}”, Resultat);

}

catch (Exception e) {

string filePath = „C:\Sample Files\Log.txt”;

if(File.Exists(filePath)) {

StreamWriter sw = newStreamWriter(filePath);

sw(e.GetType().Name);

sw.Close();

Console.WriteLine(„Proben Sie später”);

}

else {

throw new FileNotFoundException(filePath + „ ist nicht anwesend”,e);

}

}

}

catch (Exception e) {

Console.WriteLine(„Current exception = {0} ”, exception.GetType().Name);

If (exception.InnerException != null)

{

Console.WriteLine(„Inner exception = {0}”, exception.InnerException.GetType().Name);

}

}

}

}

**Custom Exceptions in C#**

Example

There isan asp.net web application. The application should allow the user to have only one logged in session. If the user is already logged in, and if he opens another browser window and tries to login again, the application should throw an error stating he is already logged in another browser window. Assume that none of the .NET framework exceptions can handle this case properly.

using System;

using System.IO;

using System.Runtime.Serialization;

public class CustomException {

public static void Main() {

try {

throw new NutzerSchonAngemeldedExce();

}

catch(NutzerSchonAngemeldedExce e) {

Console.WriteLine(e.Bericht);

}

}

}

/\* First create a class inheriting from the Exception class

[Serializable]

public class NutzerSchonAngemeldedExce : Exception {

public NutzerSchonAngemeldedExce(string Bericht)

: base(Bericht) {

}

public NutzerSchonAngemeldedExce()

: base() {

}

public NutzerSchonAngemeldedExce(string Bericht, Exception innerException)

: base(Bericht, innerException) {

}

public NutzerSchonAngemeldedExce(SerializationInfo info, StreamingContext context )

: base(info, constext) {

}

}

**Custom Exception Steps**

Create a class that derives from System.Exception class. As a convention, end the class name with Exception suffix. All .NET exceptions end with exception suffix.

Provide a public constructor, that takes in a string parameter. This constructor simply passes the string parameter, to the base exception class constructor.

Using InnerExceptions, one can also track back the original exception. If it is desired to provide this capability for the custom exception class, then the constructor must be overloaded accordingly.

If one wishes the Exception class object to work across application domains, then the object must be serializable. To make the exception class serializable mark it with Serializable attribut and provide a constructor that invokes the base Exception class constructor that takes in SerializationInfo and StreamingContext objects as parameters. It is also possible to provide one’ s own serialization.

**Exception Handling Abuse**

Writing many catch blocks for a try block that might throw different types of exceptions can be bad design. The programmer should try to correct the error from the lines that generate it using flow control statements and taking correcting actions. Thus many exception types handled by the catch blocks can be removed.

**Enums Ex.**

Public static string GetGender(Geschlecht gender)

{

switch (gender)

{

case Geschlecht.unbekennt:

Return „unbekkent”;

case Geschlecht.männlich:

Return „männlich”;

case Gender.weiblich:

Return „weiblich”;

}

}

Public enum Geschlecht

{

Unbekkent,

Männlich,

Weiblich

}

**Enums in C#**

Using System;

Public class Enums

{

Geschlecht geschlecht = 3; /\*Compile time error! Enums are strongly typed constants.

Geschlecht geschlecht = (Geschlecht)3; //Fixes the problem.

Public static void Main()

{

Int [] Werte = (int[])Enum.GetValues(typeof(Geschlecht));

Foreach (int Wert in Werte)

{

Console.WriteLine(Wert);

}

string [] Namen = Enum.GetNames(typeof(Geschlecht));

Foreach (string Name in Namen)

{

Console.WriteLine(Name);

}

}

}

Public enum Geschlecht

{

Unbekannt,

Mannlich,

Weiblich

}

//Customizing enum values.

Public enum Geschlecht

{

Unbekannt = 1,

Mannlich = 5,

Weiblich = 23

}

1. Enums are enumerations.
2. Enums are strongly typed constants. Hence, an explicit cast is needed to convert from enum type to an integral type and vice versa. Also, an enum of one type cannot be implicitly assigned to an enum of another type even though the underlying value of their members are the same.
3. The default underlying type of an enum is int.
4. The default value for first element is zero and is incremented by 1.
5. It is possible to customize the underlying type and values.
6. Enums are value types.
7. enum keyword is used to create enumerations, whereas Enum class, contains static GetValues() and GetNames() methods which can be used to list Enum underlying type values and names.

**Types and type members in C#**

Customer is the type and fields, Properties and methods are type members.

In general classes, structs, enums, interfaces, delegates are called as types and fields, properties, constructors, methods, etc., that normally reside in a type are called as type members.

In C# there are 5 different access modifiers:

1. Private 2. Protected 3. Internal 4. Protected Internal 5. Public

Type members can have all the access modifiers, whereas types can have only two (internal, public) of the five access modifiers.

Using regions one can expand and collapse sections of your code either manually, or using visual studio Edit->Outlining->Toggle All Outlining

Public class Kunde

{

#region Fields

Private int \_id;

Private string \_Vorname;

Private string \_Nachname;

#endregion

#region Properties

Public int Id

{

Get { return \_id; }

Set { \_id = value; }

}

.

.

.

#endregion

**Access modifiers in C#**

Public class Kunde

{

Protected int ID;

}

Public class CorporateCustomer : Kunde

{

Public void PrintID()

{ //ID member from the base class can be accessed through a derived

Class object.

CorporateCustomer CC = new CorporateCustomer();

CC.ID = 101;

//Or through the base keyword

base.ID = 101;

//Or through the this keyword

this.ID

Protected members are available, within the containing type and to the types that derive from the containing type.

**Internal and protected internal access modifiers in C# Tutorial 50**

Internal members are accessible anywhere within the containing assembly

Protected internal member are accessible anywhere within the containing assembly, and from within a derived class in any other assembly.

Using system;

Namespace AssemblyOne

{

Public class AssemblyOneClassI

{

internal int ID = 101;

}

Public class AssemblyOneClassII

{

Public void SampleMethod()

{

AssemblyOneClassI A1 = new AssemblyOneClassI();

Console.WriteLine(A1.ID);

}

}

}

Using System;

Using AssemblyOne;

Namespace AssemblyTwo

{

Public class AssemblyTwoClassI

{

Public void Print()

{

AssemblyOneClassI A1 = new AssemblyOneClassI();

A1.ID // compile error. Access is disallowed.

}

}

}

Let: public class AssemblyOneClassI

{

Protected internal int ID;

Then:

Namespace AssemblyTwo

{

Public class AssemblyTwoClassII : AssemblyOneClassI

{

Public void Print()

{

AssemblyOneClassI A1 = new AssemblyOneClassI();

Base.ID = 101;

AssemblyTwoClassI A2 = new AssemblyOneClassI();

A2.ID = 102;

}

}

}

**Access modifiers for types**

C# has namespaces. Marking a class with the internal keyword access modifier makes this class accessible only from the assembly-namespace it is declared. Without specifying an access modifier the default is internal.

Namespace

{

Class AssemblyOneClass

{

Public void print() {}

}

}

**52. Attributes in C#**

Attributes allow to add declarative information to programs. This information can be queried at runtime using reflection.

Public class Calculator {

[ObsoleteAttribut( „Use Add(List<int> Nummer) Method”)] // This is the attribute.

Public static int Add( int ErsteNummer, int ZweiteNummer)

{ return ErsteNummer + ZweiteNummer;

}

Public static int Add( List<int> Nummer)

{ int Sum = 0;

Foreach(int Num in Nummer)

Sum = Sum + Num;

Return Sum;

}

}

There are several predefined attributes provided by .NET.

A few of them are: obsolete- marks types and type members which are outdated.

WebMethod-to expose a method as an XML Web service method.

Serializable- indicate that a class can be serialized.

It is possible to customize the attribute using parameters. An attribute is a class that inherits from System.Attribute base class.

**53. Reflection in C#**

using System;

using System.Reflection;

namespace Werk

{

public class MainClass

{

private static void Main()

{

Type T = Type.GetType("Werk.Custo");

// Or Type T = typeof(Custo);

Custo C1 = new Custo();

//Or Type T = C1.GetType();

Console.WriteLine("Full Name = {0}", T.FullName);

Console.WriteLine("Name = {0}", T.Name);

PropertyInfo[] properties = T.GetProperties();

foreach(PropertyInfo prope in properties)

{

Console.WriteLine(prope.PropertyType.Name + " " + prope.Name);

}

Console.WriteLine("Methods in Custo class");

MethodInfo[] methods = T.GetMethods();

foreach(MethodInfo method in methods)

{

Console.WriteLine(method.ReturnType.Name + " " + method.Name);

}

Console.WriteLine("Constructors in Custo class");

ConstructorInfo[] constructors = T.GetConstructors();

foreach(ConstructorInfo constru in constructors)

{

Console.WriteLine(constru.Name);

}

}

}

public class Custo {

public int Id { get; set; }

public string Name { get; set; }

public Custo(int ID, string Name)

{

this.Id = ID;

this.Name = Name;

}

public Custo()

{

this.Id = -1;

this.Name = string.Empty;

}

public void Print()

{

Console.WriteLine("ID = {0}", this.Id);

}

public void PrintName()

{

Console.WriteLine("Name = {0}", this.Name);

}

}

}//Other methods include methods.GetParameters, GetFields, etc.

Reflection is the ability of inspecting an assembly’s metadata at runtime. It is used to find all type in an assembly and/or dynamically invoke methods in an assembly.

Uses of reflection:

1. When dragging and dropping a button on a win forms or an asp.net application. The properties window uses reflection to show all the properties of the Button class. So reflection is extensively used by IDE or UI designers.
2. Late binding can be achieved by reflection. Dynamically create an instance of a type, about which we don’ t have any information at compile time. So reflection enables one to use code that is not available at compile time.
3. Consider an example where we have two alternate implementations of an interface. The user should pick one or the other using a config file. With reflection, one can simply read the name of the class whose implementation wants to use from the config file, and instantiate an instance of that class. This is another example for late binding using reflection.

**Reflection Example**

Using System;

Using System.Reflection;

Namespace ReflectionDemo {

Public partial class Reflection : Form {

Public Reflection() {

InitializeComponent();

}

Private void btnDiscover TypeInformation\_Click(object sender, EventArgs e)

{

String TypeName = txtTypeName.Text;

Type T = Type.GetType(TypeName);

lstMethods.Items.Clear();

lstConstructors.Items.Clear();

MethodInfo[] methods = T.GetMethods();

Foreach(MethodInfo method in methods) {

lstMethods.Items.Add(method.ReturnType.Name + „ ” + Method.Name);

ConstructorInfo[] constructors = T.GetConstructors();

Foreach(ConstructorInfo constru in constructors) {

lstConstructors.Items.Add(constru.ToString());

}

}

}

}

**Late binding using reflection Tutorial 55**

Namespace Pragim {

Public class MainClass {

Private static void Main() {

//Here starts the late binding.

Assembly executingAssembly = Assembly.GetExecutingAssembly();

Type custoType = executingAssembly.GetType(„Pragim.Custo”);

object custoInstance = Activator.CreateInstance(custoType);

MethodInfo getFullNameMethod = custoType.GetMethod(„GetFullName”);

String[] parames = new string[2];

Parames[0] = „Pura”;

Parames[1] =”Tech”;

String fullName = (string)getFullNameMethod.Invoke(custoInstance, parames);

Console.WriteLine(„Full Name = {0}”, fullName);

Custo C1 = new Custo();

String FullName = C1.VollName(„Pura”, „Tech”);

}

}

Public class Custo {

Public string GetFullName(string Vorname, string Nachname) {

Return Vorname + Nachname;

}

}

1. Early binding can flag errors at compile time. With late binding there is a risk of run time exceptions.

2. Early binding is much better for performance and should always be preferred over late binding. Use late binding only when working with an objects that are not available at compile time.

**Generics in C#**

Generics have been introduced in C# 2.0. Generics allow us to design classes and methods decoupled from the data types.

Generic classes are extensively used by collection classes available in System.Collections.Generic.namespace.

One way of making AreEqual() method reusable, is to use object type parameters. Since, every type in .NET directly or indirectly inherit from System.Object type, AreEqual() method works with any data type, but the problem is performance degradation due to boxing and unboxing happening. Also AreEqual() method is no longer type safe. It is now possible to pass integer for the first parameter, and a string for the second parameter. It doesn’ t really make sense to compare strings with integers.

So the problem with using System.Object type is that:

1. AreEqual() method is not type safe
2. Performance degradation due to boxing and unboxing.

Namespace Pura {

Public class MainClass {

Private static void Main() {

bool Equal = Calcula.AreEqual<int>(1, 2);

}

}

Public class Calcula {

Public static AreEqual<T>( T Val1, T Val2) {

Return Val1.Equals(Val2);

}

}

Public class Calcula<T> //Makes the class generic

// Then in Main() bool Equal = Calcula<int>.AreEqual(1, 2);

**Overriding ToString() method**

The System.object class has four methods: GetType(), ToString(), Equals(), GetHashCode()

Due to inheritance they are available to every type.

namespace Ravichandra {

using System;

class Heidi {

static void Main() {

Kunde K1 = new Kunde();

K1.Name = "Simon";

Console.WriteLine(K1.ToString());

Console.WriteLine(Convert.ToString(K1));

}

}

public class Kunde {

public string Name {get; set; }

public string Id { get; set; }

public override string ToString() {

return this.Name + „ „ + this.Id;

//return base.ToString();

}

}

}

**Overriding Equals() method**

Namespace Ravi {

Public class Program {

Private static void Main() {

Kunde K1 = new Kunde();

K1.VorN = „Simo”;

K1.NachN = „San”;

Kunde K2 = new Kunde();

K2.VorN = „Simo”;

K2.NachN=”San”;

Console.WriteLine(K1 == K2);

Console.WriteLine(K1.Equals(K2));

}

}

Public class Kunde {

Public string VorN { get; set; }

Public string NachN { get; set; }

Public override bool Equals(object obj) {

If(obj == null) {

Return false;

}

If (!(obj is Kunde)) {

Return false;

}

Return this.VorN ==((Kunde)obj).VorN &&

This.NachN == ((Kunde)obj).NachN;

}

Public override int GetHashCode() {

return this.VorN.GetHashCode()^this.NachN.GetHashCode();

}

}

**Difference between Convert.ToString() and ToString()**

Namespace Ravi {

Public class Program {

Private static void Main() {

Kunde K1 = null;

String str = ConvertToString(K1);

//String str = K1.ToString(); //Will generate a null reference exception

Console.WriteLine(str);

Convert.ToString() handles null whereas ToString() throws a NULL Reference exception.

}

}

Public class Kunde {

}

}

**Difference between String and StringBuilder Tutorial 60**

System.String is immutable. StringBuilder is mutable. As StringBuilder objects are mutable, thye offer better performance than string objects of type System.String when heavy string manipulation is involved.

using System;

using System.Text; //To enable the StringBuilder class.

class Programme {

static void Main() {

//string Benutzer = "C#";

StringBuilder Benutzer = new StringBuilder("C#");

Benutzer.Append(" Video");

Benutzer.Append(" Vortrag");

Console.WriteLine(Benutzer.ToString());

/\* Benutzer += " Video";

Console.WriteLine(Benutzer);

Benutzer += " Vortrag";

Benutzer += " Lektura";\*/

Console.WriteLine(Benutzer);

}

}

**Partial Classes in C#**

using System.Linq;

using System.Web;

using System.Collections.Generic;

using System;

namespace Teil {

public class Program {

static void Main() {

Console.WriteLine(" ");

}

}

public partial class Kunde {

private string \_Vorname;

private string \_Nachname;

public string Vorname {

get {

return \_Vorname;

}

set {

\_Vorname = value;

}

}

public string Nachname {

get {

return \_Nachname;

}

set {

\_Nachname = value;

}

}

} //Closes the class’ body

} //Closes the namespace

namespace Teil {

public partial class teillKunde{

public string HolVollName() {

return \_Vorname + " " + \_Nachname;

}

}

}

Partial classes allow to divide a class into two or more files. All these parts are then combined into a single class, when the application is compiled. The partial keyword can also be used to split a struct or an interface over two or more files.

Advantages: 1) Visual Studio uses partial classes to separate, automatically generated system code from the developer’s code. E.g. when adding a webform, two .CS files are generated: a) WebForm1.aspx.cs contains the developer’s code, b) WebForm1.aspx.designer.cs contains the system generated code. E.g. declarations for the controls that one drags and drops on the webform.

2) When working on large projects, spreading a class over separate files allows multiple programmers to work on it simultaneously. Though, Microsoft claims this as an advantage, I haven’t seen anywhere so far, people using partial classes to work on them simultaneously.

**Creating partial classes in c#**

1. All the parts spread across different files, must use the partial keyword.
2. All the parts spread across different files, must have the same access modifiers.
3. If any of the parts are declared abstract, then the entire type is considered abstract.
4. If any of the parts are declared sealed, then the entire type is considered sealed.
5. If any of the parts inherits a class, then the entire type inherits that class.
6. C# does not support multiple class inheritance. Different parts of the partial class, must not specify different base classes.
7. Different parts of the partial class can specify different base interfaces, and the final type implements all of the interfaces listed by all of the partial declarations.
8. Any members that are declared in a partial definition are available to all of the other parts of the partial class.

**Partial methods c#**

1. A partial class or a struct can contain partial methods.
2. A partial method is created using the partial keyword.
3. A partial method declaration consists of two parts. I) The definition-only the method signature

II) the implementation. These may be in separate parts of a partial class, or in the same part.

1. The implementation for a partial method is optional. If we don’t provide the implementation, the compiler removes the signature and all calls to the method.
2. Partial methods are private by default and it is a compile time error to include any access modifiers including private.
3. It is a compile time error, to include declaration and implementation at the same time for a partial method.
4. A partial method return type must be void. Including any other return type is a compile time error.
5. Signature of the partial method declaration must match with the signature of the implementation.
6. A partial mehtod must be declared within a partial class or partial struct. A non partial class or struct cannot include partial methods.
7. A partial method can be implemented only once. Trying to implement a partial method more than once, raises a compile time error.

//Separate file

using System.Linq;

using System.Collections.Generic;

using System;

using System.Text;

namespace PartialMethodDemo {

public partial class PartialDemo {

partial void PartialMethod();

public void PublicMethod() {

Console.WriteLine("Public Method Invoked");

PartialMethod();

}

}

}

//Separate file

namespace PartialMethodDemo {

public partial class PartialDemo {

partial void PartialMethod(){

Console.WriteLine("PartialMethod Invoked");

}

}

}

//Separate file

namespace PartialMethodDemo {

class Program {

static void Main() {

PartialDemo pd = new PartialDemo();

pd.PublicMethod();

PartialMethod();

}

}

}

**How and where indexers are used in .NET**

In the metadata of HttpSessionState class there is an integer and string indexer defined. Use the „this” keyword to create indexers.

Right clicking on SqlDataReader class one can view its metadata. There is an integer and string indexer defined.Indexers allow instances of a class to be indexed like array’s members.

using System.Data;

using System.Data.SqlClient;

using System;

using System.Configuration;

namespace Demo {

public partial class Webform1 : System.Web.UI.Page {

protected void Page\_Load(object sender, EventArgs e) {

Session["Session 1"] = "Session 1 Data";

Session["Session 2"] = "Session 2 Data";

Response.Write("Session 1 Data = " + Session[0].ToString());

Response.Write("<br/>");

Response.Write("Session 2 Data = " + Session["Session 2"].ToString());

}

}

}

**Indexers Tutorial 65**

//Firma.cs file

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace Demo {

public class Arbeitnehmer {

public int ArbeitnehmerId { get; set; }

public string Name { get; set; }

public string Geschlect { get; set; }

}

public class Firma {

private List<Arbeitnehmer> listArbeitnehmer;

public Firma() {

listArbeitnehmer = new List<Arbeitnehmer>();

listArbeitnehmer.Add(new Arbeitnehmer() {ArbeitnehmerId = 1, Name = "Miko", Geschlect ="Male"});

listArbeitnehmer.Add(new Arbeitnehmer() {ArbeitnehmerId = 6, Name = "Helga", Geschlect ="Female"});

listArbeitnehmer.Add(new Arbeitnehmer() {ArbeitnehmerId = 4, Name = "Goghar", Geschlect ="Male"});

listArbeitnehmer.Add(new Arbeitnehmer() {ArbeitnehmerId = 7, Name = "Hust", Geschlect ="Male"});

}

public string this[int arbeiterId] {

get {

return listArbeitnehmer.FirstOrDefault( arb => arb.ArbeitnehmerId== arbeiterId).Name;

}

set {

listArbeitnehmer.FirstOrDefault(arb=>arb.ArbeitnehmerId== arbeiterId).Name = value;

}

}

}

//WebForm1.aspx.cs file

namespace Demo {

public partial class WebForm1 : System.Web.UI.Page {

protected void Page\_Load(object sender, EventArgs e) {

Firma Fir1 = new Firma();

Response.Write("Name des Arbeitnehmer mit Id = 1" + Fir1[1]);

Response.Write("<br/>");

Response.Write("Name des Arbeitnehmer mit Id = 6" + Fir1[6]);

Response.Write("<br/>");

Response.Write("Name des Arbeitnehmer mit Id = 7" + Fir1[7]);

Response.Write("<br/>");

Response.Write("Veränderung des Arbeitnehmersname mit Id = 1" + Fir1[1]);

Response.Write("<br/>");

Fir1[1]="1 Arbeitnehmersname verändert";

Fir1[6]="6 Arbeitnehmersname verändert";

Response.Write("Name des Arbeitnehmer mit Id = 1" + Fir1[1]);

Response.Write("<br/>");

Response.Write("Name des Arbeitnehmer mit Id = 6" + Fir1[6]);

Response.Write("<br/>");

}

}

}

1 Use the „this” keyword to create an indexer

2 Like properties indexers have get and set accessors

3 Indexers can also be overloaded

**66. Overloading indexers in C#**

Indexers are overloaded based on the number and type of parameters.

public string this[int arbeiterId] {

get {

return listArbeitnehmer.FirstOrDefault( arb => arb.ArbeitnehmerId== arbeiterId).Name;

}

set {

listArbeitnehmer.FirstOrDefault(arb=>arb.ArbeitnehmerId== arbeiterId).Name = value;

}

}

public string this[int arbeiterId, int Age] { **//Overloaded indexer**

get {

return listArbeitnehmer.FirstOrDefault( arb => arb.ArbeitnehmerId== arbeiterId).Name;

}

set {

listArbeitnehmer.FirstOrDefault(arb=>arb.ArbeitnehmerId== arbeiterId).Name = value;

}

}

public string this[string Geschlecht] { **//Overloaded indexer used in WebForm1.aspx.cs file**

get {

return listArbeitnehmer.Count( arb => arb.Geschlecht== Geschlecht).ToString();

}

set {

foreach(Arbeitnehmer arbeiter in listArbeitnehmer) {

if(arbeiter.Geschlecht == Geschlecht)

arbeiter.Geschlecht = value;

}

}

//WebForm1.aspx.cs file

namespace Demo {

public partial class WebForm1 : System.Web.UI.Page {

protected void Page\_Load(object sender, EventArgs e) {

Firma Fir1 = new Firma();

Response.Write("Bevor Veränderung");

Response.Write("<br/>");

Response.Write("Gesamt Nummer der männlichen Arbeitnehmer= " + Fir1["Male"]);

Response.Write("Gesamt Nummer der weiblichen Arbeitnehmer= " + Fir1["Female"]);

Response.Write("<br/>");

Response.Write("<br/>");

Fir1["Male"] = "Female";

Response.Write("Nach Veränderung");

Response.Write("<br/>");

Response.Write("Gesamt Nummer der männlichen Arbeitnehmer= " + Fir1["Male"]);

Response.Write("Gesamt Nummer der weiblichen Arbeitnehmer= " + Fir1["Female"]);

}

}

}

**Optional parameters**

Four ways to make method parameters optional

1 use parameter arrays

2 method overloading

3 specify parameter defaults

4 use OptionalAttribute that is present in System.Runtime.InteropServices namespace

using System;

namespace Demo {

class Program {

static void Main() {

NummerZufuegen(10, 20, new object[] {-239, -167});

NummerZufuegen(10, 20, -63, -876);

}

public static void NummerZufuegen(int ersteNummer, int zweiteNummer, params object[] Rest) {

int Resultat = ersteNummer + zweiteNummer;

if(Rest != null) {

foreach(int i in Rest) {

Resultat += i;

}

}

Console.WriteLine("Sum = " + Resultat);

}

}

}

A parameter array must be the last parameter in a formal parameter list.

**Making method parameters optional using method overloading**

/\*This is the over loaded version of NummerZufuegen. \*/

public static void NummerZufuegen(int ersteNummer, int zweiteNummer) {

NummerZufuegen(ersteNummer, zweiteNummer, null);

}

public static void NummerZufuegen(int ersteNummer, int zweiteNummer, params object[] Rest) {

int Resultat = ersteNummer + zweiteNummer;

if(Rest != null) {

foreach(int i in Rest) {

Resultat += i;

}

}

Console.WriteLine("Sum = " + Resultat);

}

**Making method parameters optional by specifying parameter defaults**

The optional parameters must appear after all mandatory parameters.

public static void NummerZufuegen(int ersteNummer, int zweiteNummer, **int[] Rest = null**) {

int Resultat = ersteNummer + zweiteNummer;

if(Rest != null) {

foreach(int i in Rest) {

Resultat += i;

}

}

Console.WriteLine("Sum = " + Resultat);

}

Named parameters

namespace Demo {

class Program {

static void Main() {

Test(1, c:2); //c is named parameter. b has value of 20, the default.

}

public static void Test(int a, int b= 20, int c = 30) {

Console.WriteLine("a = " + a);

Console.WriteLine("b = " + b);

Console.WriteLine("c = " + c);

}

}

}

Test(1, c:2); Here c is the named parameter and for b the default value is used.

**Making method parameters optional by using OptionalAttribute Tutorial 70**

using System;

**using System.Runtime.InteropServices;**

namespace Demo {

class Program {

static void Main() {

NummerZufuegen(3,-873);

}

public static void NummerZufuegen(int ersteNummer, int zweiteNummer, **[Optional]** int[] Rest) {

**Code snippets in Visual Studio**

Types:

Expansion: These snippets allow the code snippet to be inserted at the cursor.

SurroundsWith: These snippets allow the code snippet to be placed around a selected piece of code.

Refactoring: These snippets are used during code refactoring

**Dictionaries**

1 A dictionary is a collection of key-value pairs.

2 The Dictionary class is present in System.Collections.Generic namespace.

3 When creating a dictionary the type for the key and the value must be specified.

4 The Dictionary provides fast lookups for values using keys.

5 Keys in the dictionary must be unique.

using System;

using System.Collections.Generic;

class Program {

static void Main() {

Kunde K1 = new Kunde() {

Id = 110,

Name = "Georg",

Gehalt = 4000

};

Kunde K2 = new Kunde() {

Id = 30,

Name = "Heiko",

Gehalt = 12000

};

Kunde K3 = new Kunde() {

Id = 317,

Name = "Freicher",

Gehalt = 57000

};

Dictionary<int,Kunde> directo1 = new Dictionary<int, Kunde>();

directo1.Add(K1.Id,K1);

directo1.Add(K2.Id,K2);

directo1.Add(K3.Id,K3);

Console.WriteLine(directo1.ContainsKey(K3.Id));

// Dictionary K317 = directo1[317];

foreach(int key in directo1.Keys) {

Console.WriteLine(key);

//foreach(var kevalpa in directo1){

// foreach(KeyValuePair<int, Kunde> kevalpa in directo1){

/\* Console.WriteLine("Id = {0}", kevalpa.Key);

Kunde kun = kevalpa.Value;

Console.WriteLine("Id = {0}, Name = {1}, Gehalt= {2}", kun.Id, kun.Name, kun.Gehalt); \*/

}

foreach(Kunde kun in directo1.Values) {

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Console.WriteLine("Id = {0}, Name = {1}, Gehalt= {2}", kun.Id, kun.Name, kun.Gehalt);

}

}

}

public class Kunde{

public int Id { get; set; }

public string Name {get; set; }

public int Gehalt {get; set; }

}

**73 Dictionary in C#**

Dictionary<int,Kunde> directo1 = new Dictionary<int, Kunde>();

directo1.Add(K1.Id,K1);

directo1.Add(K2.Id,K2);

directo1.Add(K3.Id,K3);

Kunde kun;

if(directo1.TryGetValue(187, out kun)) //The TryGetValue method.

Console.WriteLine("Id = {0}, Name = {1}", kun.Id, kun.Name);

else

Console.WriteLine("Schüssel nicht gefunden");

//The Count() method.

Console.WriteLine("Insgesamt = {0}", directo1.Count());

Console.WriteLine("Insgesamt = {0}", directo1.Count(kvp=> kvp.Value.Gehalt > 4000));

//The Remove() method.

directo1.Remove(399);

//To remove all the elements from the dictionary use clear.

Directo1.ClearI();

//ToDictionary() method.

Dictionary<int, Kunde> dicto = kuni.ToDictionary(kus=> kus.Id, kus => kus);

foreach(KeyValuePair<int, Kunde> kvp in dicto) {

Console.WriteLine("Key = {0}", kvp.Key);

Kunde kuni = kvp.Value;

Console.WriteLine("Id = {0}, Gehalt = {1}", kuni.Id, kuni.Gehalt);

}

**74 The List collection class**

List is one of the generic collection classes in System.Collections.Generic namespace. Other classes

1 Dictionary

2 List

3 Stack

4 Queue etc.

A List class can be used to create a collection of any type such as Integers, Strings or abstract types. Lists can grow automatically. Relevant methods to search, sort,...

Kunde[] kusto = new Kunde[3];

// kusto[0] = K1;

// kusto[1] = K2;

// kusto[2] = K3;

List<Kunde> Kunden = new List<Kunde>(2);

Kunden.Add(K1);

Kunden.Add(K2);

Kunden.Insert(0, K2);

Console.WriteLine(Kunden.IndexOf(K2));

Console.WriteLine(Kunden.IndexOf(K2, 1, 2));//Overloaded version of IndexOf()

foreach(Kunde k in Kunden) {

Console.WriteLine("Id = {0}, Gehalt = {1}", k.Id, k.Gehalt);

}

//Or with a for loop

for (int i=0; i<k.Count; i++){ //Count is a property equivalent to Count()

Kunde k = Kunden[i];

Console.WriteLine("Id = {0}, Gehalt = {1}", k.Id, k.Gehalt);

}

**List collection classs in C# method Tutorial 75**

Contains(). Checks if an item exists in the list. This method returns true if the items exist.

Exists() checks if an item exists in the list based on a condition. It returns true if the item exists.

Find() searches for an element that matches the conditions defined by the specified lambda expression and returns the first matching item from the list.

FindLast() searches for an element that matches the conditions defined by the specified lambda expression and returns the Last matching item from the list.

FindAll() returns all the items from the list that match the conditions specified by the lambda expression.

FindIndex() returns the index of the first item that matches the condition specified by the lambda expression. There are two other overloads of this method which allow to specify the range of elements to search in the list.

FindLastIndex() returns the index of the last item that matches the condition specified by the lambda expression. There are two other overloaded versions which allow to specify the range of elements to search in the list.

Convert an array to a List-ToList()

Convert a list to an array-ToArray()

Convert a List to a Dictionary-ToDictionary()

List<Kunde> Kunden = new List<Kunde>(2);

Kunden.Add(K1);

Kunden.Add(K2);

Kunden.Insert(2, K3);

// if(Kunden.**Exists**(kusto => kusto.Name.StartsWith("P"))) {

if(Kunden.**Contains**(K3)) {

}

else {

Console.WriteLine("Object does not exist in the list");

}

int index = Kunden.FindIndex(kusto => kusto.Gehalt > 5000);

// Console.WriteLine(index);

Kunden.**Find**(kusto => kusto.Gehalt > 5000);

Kunden.**FindLast**(kusto => kusto.Gehalt > 5000);

Kunde[] kusto = new Kunde[3];

kusto[0] = K1;

kusto[1] = K2;

kusto[2] = K3;

List<Kunde> kus = **kusto.ToList();**

List<Kunde> Kunden = new List<Kunde>(2);

Kunden.Add(K1);

Kunden.Add(K2);

Kunden.Insert(2, K3);

**Kunden.ToArray();**

//ToDictionary()

List<Kunde> Kunden = new List<Kunde>(2);

Kunden.Add(K1);

Kunden.Add(K2);

Kunden.Insert(2, K3);

Dictionary<int, Kunde> dicto = Kunden.ToDictionary(x=> x.Id);

foreach(KeVal<int, Kunde> kvp in dicto) {

Console.WriteLine("Key =", kvp.Key);

Kunde ka= kvp.Value;

Console.WriteLine("Id = {0}, Gehalt = {1}", ka.Id, ka.Gehalt);

}

**76 Generic list class and ranges in C#**

AddRange() allows to add another list of items to the end of the list

GetRange() using an index item we can retrieve only one item at a time from the list, if one wants to get a list of items from the list use GeRange(). This function takes two arguments, the start index in the list and the number of elements to return.

InsertRange() inserts another list of items to a list at a specified index.

Remove() removes only the first matching item from the list. RemoveAt() removes the item at the specified index in the list. RemoveAll() removes all the items that match the specified condition. RemoveRange() removes a range of elements from the list. This method expects two parameters, the start index in the list and the number of elements to remove. To remove all the elements from the list without specifying any condition use the Clear() function.

using System;

using System.Collections.Generic;

using System.Linq;

class Program {

static void Main() {

Kunde K1 = new Kunde() {

Id = 110,

Name = "Georg",

Gehalt = 4000,

Typ = "RetaiKunde"

};

Kunde K2 = new Kunde() {

Id = 30,

Name = "Heiko",

Gehalt = 12000,

Typ = "RetaiKunde"

};

Kunde K3 = new Kunde() {

Id = 317,

Name = "Freicher",

Gehalt = 57000,

Typ = "RetaiKunde"

};

Kunde K4 = new Kunde() {

Id = 730,

Name = "Gunther",

Gehalt = 5400,

Typ = "CorpoKunde"

};

List<Kunde> RetaiKun = new List<Kunde>();

RetaiKun.Add(K1);

RetaiKun.Add(K2);

RetaiKun.Add(K3);

List<Kunde> CorpoKunde = new List<Kunde>();

CorpoKunde.Add(K4);

**//Will add K4 Kunde to the list RetaiKun**

**// RetaiKun.AddRange(CorpoKunde);**

**Console.WriteLine(RetaiKun.GetRange(1,3));**

**RetaiKun.InsertRange(0, CorpoKunde);**

**//RetaiKun.RemoveAll(x=> x.Typ =="CorpoKunde");**

**RetaiKun.RemoveRange(3,1);**

}

}

public class Kunde{

public int Id { get; set; }

public string Name { get; set; }

public int Gehalt { get; set; }

public string Typ { get; set; }

}

**Sort a list of primitive types in C#**

Sorting a list of primitives is simple. Invoke the Sort() method on the list and the data will be automatically sorted in ascending order. To sort the data in descending order invoke the Reverse() method on the list instance. To sort abstract types we must implement the IComparable interface. The primitive types have implemented that interface.

List<int> listanume = new List<int>() {1, 8, 7, 5, 2, -12};

listanume.Sort();

foreach(int numero in listanume) {

Console.WriteLine(numero);

}

listanume.Reverse();

**78 Sort a list of abstract types**

To sort a list of complex types without using Linq the abstract type must implement IComparable interface and provide implementation for CompareTo() method. CompareTo() method returns an integer. Alternatively invoke CompareTo() method. Gehalt property of the Kunde object is int. CompareTo() is already implemented on integer type.

If do not want to use the Sort() method provided by the class then you may implement IComparer interface.

Pass an instance of the class that implements IComparer interface as an argument to the Sort().

List<Kunde> RetaiKun = new List<Kunde>();

RetaiKun.Add(K1);

RetaiKun.Add(K2);

RetaiKun.Add(K3);

RetaiKun.Sort();

RetaiKun.Reverse();

foreach(Kunde k in RetaiKun) {

Console.WriteLine(k.Gehalt);

}

public class Kunde : IComparable<Kunde> {

public int Id { get; set; }

public string Name { get; set; }

public int Gehalt { get; set; }

public string Typ { get; set; }

public int CompareTo(Kunde ewa) {

/\* if(this.Gehalt > ewa.Gehalt)

return 1;

else if(this.Gehalt < ewa.Gehalt)

return -1;

else return 0;\*/

//Alternatively

return this.Gehalt.CompareTo(ewa.Gehalt);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In main() {

RetaiKun.Sort(new SortdurchName());

foreach(Kunde k in RetaiKun) {

Console.WriteLine(k.Name);

}

public class SortdurchName:IComparer<Kunde> {

public int Compare(Kunde a, Kunde b){

return a.Name.CompareTo(b.Name);

}

}

public class Kunde : IComparable<Kunde> {

public int CompareTo(Kunde ewa) {

//...

return this.Gehalt.CompareTo(ewa.Gehalt);

}

**Sorting abstract types by comparison delegate**

One of the overloads of the Sort() method in List class expects Comparison delegate to be passed as an argument.

1 Create a function whose signature matches the signature of System.Comparison delegate. This is the method where we need to write the logic to compare two Kunde objects.

2 Create an instance of Syste.Comparison delegate and pass the name of the function in step 1 as the argument. So Comparison delegate is pointing to this function that contains the code to compare two Kunde objects.

3 Pass the delegate as an argument to Sort() function.

Second method: RetaiKun.Sort(delegate(Kunde k1,Kunde k2){ return k1.Id.CompareTo(k2.Id);});

Third method: using lambda expression

**RetaiKun.Sort((k1,k2)=>k1.Id.CompareTo(k2.Id));**

List<Kunde> RetaiKun = new List<Kunde>();

RetaiKun.Add(K1);

RetaiKun.Add(K2);

RetaiKun.Add(K3);

Comparison<Kunde> kungleich = new Comparison<Kunde>(KundeVergleichung);

RetaiKun.Sort(kungleich);

foreach(Kunde k in RetaiKun) {

Console.WriteLine(k.Id);

}

}

//This is outside main but inside class Program

private static int KundeVergleichung(Kunde x, Kunde w) {

return x.Id.CompareTo(w.Id);

}

//second method without the private method and the Comparison Delegate instantiation.

RetaiKun.Sort(delegate(Kunde k1,Kunde k2){ return k1.Id.CompareTo(k2.Id);});

**More Methods of the List class Tutorial 80**

TrueForAll() returns true or false depending on whether every element in the list matches the conditions defined by the specified predicate.

List<Kunde> RetaiKun = new List<Kunde>(4);

RetaiKun.Add(K1);

RetaiKun.Add(K2);

RetaiKun.Add(K3);

RetaiKun.TrueForAll(x=>x.Gehalt>5000);

AsReadOnly() returns a read-only wrapper for the current collection. Use this method if you don’ t want the client code to modify the collection-adding/removing elements. The ReadOnlyCollection will not have methods to add or remove items from the collection. One can only read items from this collection.

ReadOnlyCollection<KUnde> roku = RetaiKun.AsReadOnly();

TrimExcess() sets the capacity to the actual number of elements in the List, if that number is less than a threshold value.

According to MSDN:

This method can be used to minize a collection’s memory overhead if no new elements will be added to the collection. The cost of reallocating and copying a large List<T> can be considerable. So the TrimExcess method does nothing if the list is at more than 90% of capacity. This avoids incurring a large reallocation cost for a relatively small gain. The current threshold is 90% but this could change in the future.

List<Kunde> RetaiKun = new List<Kunde>(511);

RetaiKun.Add(K1);

RetaiKun.Add(K2);

RetaiKun.Add(K3);

Console.WriteLine(RetaiKun.Capacity);

RetaiKun.TrimExcess();

Console.WriteLine(RetaiKun.Capacity);

**81 Use of dictionary over list**

The Find() method of the List class loops through each object in the list until a match is found. So if you want to look up a value using a key dictionary is better for performance over list. Use a dictionary when you know the collection will be primarily used for lookups.

List implementation:

public static void Main() {

Land Deutschland = new Land() {Name = "Deutschland", Kode = "DE", Hauptstadt = "Berlin" };

Land Denmark = new Land() {Name = "Denmark", Kode = "DK", Hauptstadt = "Kopenhavn" };

Land Finland = new Land() { Name = "Suoma", Kode = "S", Hauptstadt = "Helsinki" };

Land Holland = new Land() {Name = "Holland", Kode = "NL", Hauptstadt = "Amsterdam" };

List<Land> Landliste = new List<Land>();

Landliste.Add(Deutschland);

Landliste.Add(Denmark);

Landliste.Add(Holland);

Landliste.Add(Finland);

string strkod = string.Empty;

string NutzerWahl = string.Empty;

do {

Console.WriteLine("Bitte, gebe eine Landskode ein ");

strkod = Console.ReadLine().ToUpper();

Land Resul = Landliste.Find(Land => Land.Kode == strKod);

if (Resul != null)

Console.WriteLine("Name = {0}, Hauptstadt = {0}",Resul.Name, Resul.Kode );

else

Console.WriteLine("Landskode ungueltig");

do {

Console.WriteLine("Magst du fortsetzen?");

NutzerWahl = Console.ReadLine().ToUpper();

} while(NutzerWahl !="No" && NutzerWahl != "yes");

}

while(NutzerWahl == "yes");

}

}

public class Land {

public string Name { get; set; }

public string Kode { get; set; }

public string Hauptstadt { get; set; }

}

**82 Generic queue collection class**

Queue is a generic FIFO collection class in System.Collections.Generic namespace. To add items to the end of the queue use Enqueue() method.

To remove an item that is present at the beginning of the queue use Dequeue() method.

A foreach loop iterates through the items in the queue but will not remove them from the queue.

To check if an item, exists in the queue use Contains() method.

Dequeue() method removes and returns the item at the beginning of the queue whereas Peek() returns the item at the beginning of the queue, without removing it.

Queue<Kunde> kunde = new Queue<Kunde>();

kunde.Enqueue(K1);

kunde.Enqueue(K2);

kunde.Enqueue(K3);

kunde.Enqueue(K4);

Kunde k1 = kunde.Dequeue(); //Removes item from the queue.

k1 = kunde.Peek(); /\*Returns the first

item without removing it from the queue. \*/

Console.WriteLine(kunde.Contains(K1));

Console.WriteLine(kunde.Count);

**83 Generic stack collection class**

Stack is a generic LIFO (Last in First Out) collection class that is present in System.Collections.Generic namespace. The last item added to the stack will be the first item to be removed from the stack. To insert an item at the top of the stack use the Push() method. To remove and return the item that is present at the top of the stack use the Pop() method. To check if an item exists in the stack use the Contains() method. Pop() removes and returns the item from the top of the stack whereas Peek() returns the item from the top of the stack without removing it.

Stack<Kunde> staKun = new Stack<Kunde>();

staKun.Push(new Kunde());

staKun.Push(new Kunde());

staKun.Push(new Kunde());

// Kunde k1 = staKun.Pop(); //Removes the item from the stack.

Kunde k1 = staKun.Peek(); //Returns without removing the item from the stack.

Console.WriteLine(k1);

Console.WriteLine("Kundennummer: " + staKun.Count);

**Example of a queue collection class**

It must be linked with an aspx file. Below is the aspx.cs file.

namespace {

public partial class WebForm1:Syste.Web.UI.Page {

protected void Page\_Load(object sender, EventArgs e) {

if(Session["TokenQueue"]== null) {

Queue<int> queueTokens = new Queue<int>();

Session["TokenQueue"] = queueTokens;

}

}

Console.WriteLine(k1);

protected void butToken(object sender, EventArgs e){

Queue<int> tokenQueue = (Queue<int>)Session["TokenQueue"];

lblStatus.Text = "Er sind " + tokenQueue.Count.ToString() +

"Kunden vor Sie";

if(Session["LetzteTokennummerAusgegeben"]==null) {

Session["LetzteTokennummerAusgegeben"] = 0;

}

int naexteTokenNummer = (int)Session["LetzteTokennummerAusgegeben"] + 1;

Session["LetzteTokennummerAusgegeben"]= naexteTokenNummer;

tokenQueue.Enqueue(naexteTokenNummer) ;

listTokens.Items.Clear();

foreach(int token in tokenQueue){

listTokens.Items.Add(token.ToString());

}

}

private void BedienenNaexteKunde(TextBox textbox, int counterNummer) {

Queue<int> tokenQueue = (Queue<int>)Session["TokenQueue"];

if(tokenQueue.Count==0) {

textbox.Text = "Keine Kunden in der Queue";

}

else {

int tokenNummerZuBedienen= tokenQueue.Dequeue();

textbox.Text = tokenNummerZuBedienen.ToString();

txtBild.Text = "Tokennummer: " + tokenNummerZuBedienen.ToString() +

" Bitte gehen zu Kounter " + counterNummer;

AddTokensToListBox(tokenQueue);

}

}

protected void butCounter1(object sender, EventArgs e){

BedienenNaexteKunde(txtCounter1,1);

}

protected void butCounter1(object sender, EventArgs e){

BedienenNaexteKunde(txtCounter2,2);

}

protected void butCounter1(object sender, EventArgs e){

BedienenNaexteKunde(txtCounter3,3);

}

}

}

**Stack Collection Class Tutorial 85**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace WebFormsDemo {

public class BasePage : System.Web.UI.Page {

protected override void OnLoad(EventArgs e) {

if(Session[„URLStack”] == null) {

Stack<string> urlStack = new Stack<string>();

Session[„URLStack”] = urlStack;

}

If(Request.UrlReferrer != null && !this.Page.IsPostBack && Session[„BackButtonClicked”] == null) {

Stack<string> urlStack = (Stack<string>Session[„URLStack”];

urlStack.Push(Request.urlReferrer.AbsoluteUri);

}

if(Session[„BackButtonClicked”] != null) {

Session[„BackButtonClicked”] = null;

}

}

}

}

// This is the MasterPage CS file.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace WebFormsDemo {

public class BasePage : System.Web.UI.MasterPage {

protected void btnBack\_Click(object sender, EventArgs e) {

Session[„BackButtonClicked”] = „Yes”)

If (Session[„URLStack”] != null) {

Stack<string> urlStack = (Stack<string>Session[„URLStack”];

If (urlStack.Count > 0) {

String url = urlStack.Pop();

Response.Redirect(url);

}

Else {

IblMessage.Text = „Es gibt keine Seite im Chronik”;

}

}

}

}

}

**Multithreading**

Process is the supply of resources by the operating system to execute a program. Each process has a unique process id. The process within which a program is being executed can be viewed with the windows task manager.

Thread is a light weight process. A process has at least one thread which is commonly called as main thread which actually executes the application code. A single process can have multiple threads. All the threading related classes are present in System.Threading namespace.

Namespace MultithreadingExample {

Public partial class Form1 : Form {

Public Form1() {

InitializeComponent();

}

Private void btnTieConsumingWork\_Click(object sender, EventArgs e) {

btnTimeConsumingWork.Enabled = false;

btnPrintNumbers.Enabled = false;

Thread workerThread = new Thread(DoTimeConsumingWork);

workerThread.Start();

//DoTimeConsumingWork();

btnTimeConsumingWork.Enabled = true;

btnPrintNumbers.Enabled = true;

}

Private void DoTimeConsumingWork() {

Thread.Sleep(6000);

}

Private void benPrintNumbers\_Click(object sender, EventArgs e) {

For(int i = 1; i<= 10; i++)

listBoxNumbers.Items.Add(i);

}

}

}

**Advantages and Disadvantages of Multithreading**

1. To maintain a responsive user interface.

2. To make efficient use of processor time while waiting for I/O operations to complete.

3. To split large, CPU-bound tasks to be processed simultaneously on a machine that has multiple processors/cores.

Disadvantages of multithreading

1. On a single processor/core machine threading can affect performance negatively as there is overhead involved with context-switching.
2. Have to write more lines of code to accomplish the same task.
3. Multithreaded applications are difficult to write, understand, debug and maintain.

Use multithreading only when the advantages outweigh the disadvantages.

**88 ThreadStart Delegate**

The purpose of creating a Thread is to execute a function. A delegate is a type safe function pointer, meaning that it points to a function that the thread has to execute. All threads require an entry point to start execution. Any thread created will need an explicitly defined entry point i.e. a pointer to the function where they should begin execution. So threads always require a delegate. The thread function need not be a static function.

using System.Threading;

namespace threadStartDeleg {

class Program {

static void Main() {

// Thread T1 = new Thread(new ThreadStart(Nummer.PrintNummer));

// Thread T1 = new Thread(delegate() {Nummer.PrintNummer();});

//Or even

Thread T1 = new Thread(() =>Nummer.PrintNummer());

T1.Start();

}

}

class Nummer {

public static void PrintNummer() {

for (int i= 1; i<=10; i++) {

Console.WriteLine(i);

}

}

}

}

**89 Parameterized ThreadStart Delegate**

using System.Threading;

using System;

namespace threadStartDeleg {

class Program {

static void Main() {

Console.WriteLine("Geben Sie das Grenzenummer!");

object limit = Console.ReadLine();

Nummer num = new Nummer();

// ParameterizedThreadStart paramThreadSta = new ParameterizedThreadStart(num.PrintNummer);

// Thread T1 = new Thread(paramThreadSta); or else

Thread T1 = new Thread(num.PrintNummer); /\***The compiler implicitly converts**

**new Thread(num.PrintNummer); to new Thread(new ParameterizedThreadStart(Nummer.PrintNummer));** \*/

T1.Start(limit);

}

}

class Nummer {

public void PrintNummer(object limit) {

int num = 0;

if(int.TryParse(limit.ToString(), out num)){

for (int i= 1; i<num; i++) {

Console.WriteLine(i);

}

}

}

}

}

Use ParameterizedThreadStart delegate if you have to pass some data to the Thread function, else use the ThreadStart delegate.

Using ParameterizedThreadStart delegate and Thread.Start(Object) method to pass data to the Thread function is not type safe as they operate on object datatype and any type of data can be passed.

If one tries to change the data type of the target parameter of PrintNummer() function from object to int, a compiler error will be raised as the signature of PrintNummer() method does not match with the signature of the ParameterizedThreadStart delegate.

**Passing data to the Thread function in a type safe manner Tutorial 90**

To pass data to the Thread function in a type safe manner, encapsulate the thread function and the data it needs in a helper class and use the ThreadStart delegate to execute the thread function.

using System.Threading;

using System;

namespace threadStartDeleg {

class Program {

static void Main() {

Console.WriteLine("Geben Sie das Grenzenummer!");

int limit = Convert.ToInt32(Console.ReadLine());

Nummer num = new Nummer(limit);

//Thread T1 = new Thread(new ThreadStart(num.PrintNummer)); or else

Thread T1 = new Thread(num.PrintNummer); //The compiler does it automatically

T1.Start();

}

}

class Nummer {

private int \_limit;

public Nummer(int lmt) {

this.\_limit = lmt;

}

public void PrintNummer() {

for (int i= 1; i<\_limit; i++) {

Console.WriteLine(i);

}

}

}

}

**The Singleton pattern**

The singleton controls concurrent access to the resource.

It ensures there is only one object available across the application in a controlled state.

Ensure that only one instance of the class exists.

Provide global access to that instance by

Declaring all constructors of the class to be private.

Providing static method- or get property- that returns a reference to the instance.

The instance is stored as a private static variable.

Disallow inheritance using the sealed keyword for the class.

public sealed class Bob {

private static int coun = 0;

private static Bob Tania = null;

private Bob() { coun = coun + 1;

Console.WriteLine(coun.ToString());

}

public static Bob f1 {

get

{ if(Tania == null)

Tania = new Bob();

return Tania;

}

}

}

**Thread safety**

Add in the class’ private fields:

Private static readonly object obj = new object();

And in the public method add:

public static Bob f1 {

get

{ if(Tania == null) {

lock(obj) {

if(Tania == null)

Tania = new Bob();

}

}

return Tania;

}

}

**Lazy Eager loading in Singleton**

public sealed class Bob {

private static int coun = 0;

private static Bob Tania = null;

private Bob() { coun = coun + 1;

Console.WriteLine(coun.ToString());

}

private static readonly Lazy<Bob> Tania = new Lazy<Bob> (()=>new Bob());

public static Bob f1 {

get {

return Tania;

}

}

}

**Factory Design Pattern Intro**

Guidelines

The Object needs to be extended to subclasses.

The Class doesn’ t know what exact subclasses has to create.

The Product implementation tend to change over time and the Client remains unchanged.