

C# i .net

Andrzej Krajniak

7 lutego 2015

```
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");

            // Keep the console window open
            // in debug mode.
            Console.WriteLine("Press any key
                               to exit.");
            Console.ReadKey();
        }
    }
}
```

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów

numerycznych

Konwersje 'explicit'
typów

numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Typy wartościowe

`int , double , bool , struct , enum ...`

Typy referencyjne

`string , tablice, class , delegate ...`

Typy wskaźnikowe

zazwyczaj nie używane (używane tylko w kontekście unchecked)

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów
numerycznychKonwersje 'explicit'
typów
numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

bool	System.Boolean	uint	System.UInt32
byte	System.Byte	long	System.Int64
sbyte	System.SByte	ulong	System.UInt64
char	System.Char	object	System.Object
decimal	System.Decimal	short	System.Int16
double	System.Double	ushort	System.UInt16
float	System.Single	string	System.String
int	System.Int32		

Konwersje 'implicit' typów numerycznych

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów
numerycznychKonwersje 'explicit'
typów
numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Z typu	do
sbyte	short, int, long, float, double, or decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
char	ushort, int, uint, long, ulong, float, double, or decimal
float	double
ulong	float, double, or decimal

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów
numerycznychKonwersje 'explicit'
typów
numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Uwagi:

- uwaga na utratę precyzji przy konwersji na float i double
- typ char nie ma żadnej konwersji automatycznej
- brak konwersji automatycznej pomiędzy float i double a decimal
- stałe numeryczne są automatycznie konwertowane jeśli mieszczą się w zakresie

Konwersje 'explicit' typów numerycznych

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów
numerycznychKonwersje 'explicit'
typów
numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Z typu	do
sbyte	byte, ushort, uint, ulong, or char
byte	sbyte or char
short	sbyte, byte, ushort, uint, ulong, or char
ushort	sbyte, byte, short, or char
int	sbyte, byte, short, ushort, uint, ulong, or char
uint	sbyte, byte, short, ushort, int, or char
long	sbyte, byte, short, ushort, int, uint, ulong, or char
ulong	sbyte, byte, short, ushort, int, uint, long, or char
char	sbyte, byte, or short
float	sbyte, byte, short, ushort, int, uint, long, ulong, char, or decimal
double	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or decimal
decimal	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or double

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów
numerycznychKonwersje 'explicit'
typów
numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Uwagi:

- konwersje explicit mogą powodować utratę precyzji lub rzucenie wyjątku
- przy konwersji z `decimal` na typy całkowite wartość jest zaokrąglana do najbliższej wartości całkowitej, gdy ta wartości nie mieści się w zakresie rzuwany jest *OverflowException*
- przy konwersji z `float` i z `double` do typów całkowitych, część ułamkowa zostaje odcięta, jeżeli wartość nie mieści się w zakresie zostaje rzucony wyjątek *OverflowException* lub wynik działania jest nieznany (w zależności od ustawień kontekstu)
- Przy konwersji z `float` i `double` do `decimal` może nastąpić *OverflowException*

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów

numerycznych

Konwersje 'explicit'
typów

numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

class

- Typ referencyjny.
- Może dziedziczyć po dokładnie jednej klasie.
- Jeśli nie podamy klasy bazowej dziedziczy po object
- Może implementować dowolną liczbę interfejsów

```
class Hello : OtherClassType, IInterface
{
    static void Main()
    {
        Console.WriteLine("Hello World!");
    }
}
```

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów
numerycznychKonwersje 'explicit'
typów
numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

interface

- Typ referencyjny
- Nie instancjonowalny
- Może dziedziczyć po innych interfejsach

```
interface IInterface
{
    int SomeMethod();
    String SomeProperty{ get; set; }
    event EventHandler SomeEvent
    int this[string someIndexerIndex]
}
```

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów

numerycznych

Konwersje 'explicit'
typów

numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

struct

- Typ wartościowy
- Automatycznie dziedziczy po *System.ValueType* które dziedziczy po *object*
- Nie można po nim dziedziczyć (automatyczne sealed)
- Automatycznie definiuje konstruktor bezparametrowy
- Może implementować dowolną liczbę interfejsów

```
struct Hello : IInterface
{
    int _i;
    // Can't initiate instance fields
    // int _j = 1;

    // Can't declare parameterless constructor
    // Hello() { }
    Hello(int i) { }
}
```

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów
numerycznychKonwersje 'explicit'
typów
numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

enum

- Typ wartościowy
- Domyślnie dziedziczy po `int` ale może dziedziczyć po dowolnym typie całkowitym
- Nie można po nim dziedziczyć (automatyczne sealed)
- Nie może definiować innych członków niż wartości tego typu

```
// (int)Days.Sat has value 0
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

// (int)Days.Sun has value 2
enum Days {Sat = 1, Sun, Mon, Tue, Wed, Thu,
           Fri}
```

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów

numerycznych

Konwersje 'explicit'
typów

numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

delegate

- Typ referencyjny
- Działa podobnie jak wskaźnik do funkcji

```
delegate int MyDelegate(int x);  
class Program  
{  
    static int DelImpl(int x) { return x; }  
  
    static void Main(string[] args)  
    {  
        MyDelegate del = new  
            MyDelegate(DelImpl);  
        //Compiler shortcut  
        del = DelImpl;  
        Console.Out.WriteLine( del(1) );  
    }  
}
```

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów

numerycznych

Konwersje 'explicit'
typów

numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

object

- Typ referencyjny
- Dziedziczy po *System.Array*

```
int[] oneDimensionArray = new int[5];
int[] oneDimensionArray2 = new int[] {1, 2,
    3, 4};
// Shortcut
int[] oneDimensionArray3 = { 1, 2, 3, 4 };
// Error
// oneDimensionArray = {1,2,3,4};
int[,] twoDimensionalArray = {{1, 2}, {2, 3}};
int[][] jaggedArray = {new int[] {1, 2}, new
    int[] {2, 3}};
```

Podstawy

Typy

Rodzaje typów

Typy budowane

Konwersje 'implicit'
typów

numerycznych

Konwersje 'explicit'
typów

numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

object

- Typ referencyjny
- Klasa bazowa dla pozostałych typów (nie do końca prawda)

```
class object
{
    public Object();
    public virtual string ToString();
    public virtual bool Equals(object obj);
    public static bool Equals(object objA,
        object objB);
    public static bool ReferenceEquals(object
        objA, object objB);
    public virtual int GetHashCode();
    public Type GetType();
}
```

Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów

numerycznych

Konwersje 'explicit'
typów

numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

dynamic

- Jest to obiekt "niewiadomego" typu
- Można do niego przypisać cokolwiek, można go przypisać do czegokolwiek
- Można wywołać na nim dowolną metodę, użyć dowolnego pola
- Sprawdzenie czy dany członek typu istnieje następuje w czasie uruchomienia nie w czasie kompilacji

```
int i = 1;  
dynamic d = i;  
int j = d;  
  
d.AnyMethod();
```


Podstawy

Typy

Rodzaje typów

Typy wbudowane

Konwersje 'implicit'
typów

numerycznych

Konwersje 'explicit'
typów

numerycznych

Klasy

Interfejsy

Struktury

Enum

Delegaty

Tablice

object

dynamic

Nullable

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Nullable

- Generyczny typ wartościowy.
- Można do niego przypisać implicite wartość typu generycznego lub null

```
Nullable<bool> b = true;  
int? x = null;  
  
// Boxing generic type  
object o = b;  
  
// o == null  
o = x;
```

Podstawy

Konstrukcja
programuInstrukcja
warunkowa

Pętle

Switch

Wyjątki

Lock

Operatory

var

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Przykład:

```
if (i == 1)
{
    return 0;
}
// -----
if (someBoolValue)
{
    DoSomething();
}
else
{
    DoSomethingElse();
}
```

Podstawy

Konstrukcja
programuInstrukcja
warunkowa

Pętle

Switch

Wyjątki

Lock

Operatory

var

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Przykład:

```
while (someBoolValue)
    continue;

// -----
do
{
    break;
} while (someBoolValue);

// -----
foreach (var i in iEnumerableImplementator)
{
    for (int i=0; i<=10; ++i)
        if (i == 2)
            goto Finish
}

Finish:
    return 0;
```

Podstawy

Konstrukcja
programuInstrukcja
warunkowa

Pętle

Switch

Wyjątki

Lock

Operatory

var

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Przykład:

```
switch (str)
{
    case "1":
    case "small":
        break;
    case "2":
    case "medium":
        cost += 25;
        goto case "1";
    case "3":
    case "large":
        cost += 50;
        goto case "1";
    default:
        Console.WriteLine("Select 1, 2, or 3.");
        break;
}
```

Podstawy

Konstrukcja
programuInstrukcja
warunkowa

Pętle

Switch

Wyjątki

Lock

Operatory

var

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

```
try
{
    throw new
        SomeSystemDotExceptionDerivative();
}
catch (SomeSystemDotExceptionDerivative ex)
{
    Console.Out.WriteLine(ex.Message);
}
catch (Exception)
{
    throw;
}
finally
{
    // Cleanup
}
```

Podstawy

Konstrukcja
programuInstrukcja
warunkowa

Pętle

Switch

Wyjątki

LockOperatory
varTworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

```
class Account
{
    decimal balance;
    private Object thisLock = new Object();

    public bool Withdraw(decimal amount)
    {
        lock (thisLock)
        {
            if(balance < amount)
                return false;
            balance -= amount;
            return true
        }
    }
}
```

Podstawy

Konstrukcja
programuInstrukcja
warunkowa

Pętle

Switch

Wyjątki

Lock

Operatory

var

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Podstawowe	x.y x-- unchecked	f(x) new default(T)	a[x] typeof delegate	x++ checked sizeof
Unarne	+x ++x	-x --x	!x (T)x	~ x await
Mnożne	x*y	x/y	x%y	
Addytywne	x+y	x-y		
Przesunięcia	x<<y	x>>y		
Relacyjne	x<y	x>y	x<=y	x>=y is as
Prównawcze	x==y	x!=y		
Logiczne*	x&y	x^y	x y	x&&y x y
	x ?? y			
	?.			
Przypisania	x=y	x+=y	etc...	

Słowa kluczowego `var` można używać zamiast podawania typu zmiennej w momencie gdy kompilator może wywnioskować typ tej zmiennej z kontekstu.

```
class VarFun
{
    // ERROR:
    // var x = 1;

    public int GetVal() { return 2; }
    public void DoSomething()
    {
        var x = 1;
        var s = new object();
        // ERROR
        // var z;
        var y = GetVal();
    }
}
```


public

typ dostępny dla wszystkich typów

internal

typ dostępny dla typów w tym samym assembly

brak modyfikatora

domyślnie ustalany jest **internal**

```
// These delegates are accessible only in  
    this assembly  
delegate int SomeDelegate(object x)  
internal delegate void SomeOtherDelegate()  
  
// This class is accessible in any assembly  
public class SomeClass { }
```

public

członek dostępny dla wszystkich typów

internal

członek dostępny dla typów w tym samym
assembly

protected

członek dostępny dla członków tego typu i typów
dziedziczących

private

członek dostępny dla członków tego typu

protected internal

członek dostępny dla członków tego typu, typów
dziedziczących i typów w tym samym assembly

brak modyfikatora

domyślnie ustalany jest **private** dla **class** i
struct oraz **public** dla **enum** i **interface**

abstract

klasa abstrakcyjna nie może być instancjonowana,
metody klasy oznaczone słowem kluczowym
`abstract` mogą nie posiadać implementacji

sealed

po tej klasie nie można dziedziczyć

static

wszyscy członkowie klasy muszą używać
modyfikatora `static`, klasy nie można
instancjonować

Podstawy

Tworzenie
typów

Rodzaje członków

Pola, właściwości i
zdarzeniaMetody,
konstruktory,
finalizatory,
indekseryCzłonkowie
statyczniPrzeładowanie
operatorówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

```
class FieldsAndProperties
{
    private int _field = 1;
    public string AutoProperty { get; private
        set; }
    private int Property
    {
        get { return _field; }
        set { _field = value; }
    }
    public event EventHandler SimpleEvent;
    private event EventHandler _privateEvent;
    public event EventHandler PropertyEven
    {
        add { _privateEvent += value; }
        remove { _privateEvent -= value; }
    }
}
```

Podstawy

Tworzenie
typów

Rodzaje członków

Pola, właściwości i
zdarzeniaMetody,
konstruktory,
finalizatory,
indekseryCzłonkowie
statyczniPrzeładowanie
operatorówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

```
class Methods
{
    \\ Constructors
    private Methods(){ }
    public Methods(int i) : base() { }
    \\ Finalizer
    ~Methods() { }
    \\ Method
    public void SomeMethod(int param) { }
    \\ Indexer
    public int this [string index]
    {
        get { /*...*/ }
        set { /*...*/ }
    }
}
```

- Nie potrzebują instancji klasy do działania
- Statyczne pola inicjalizowane są przy pierwszym "użyciu klasy"
- Statyczny konstruktor wołany jest zaraz po zainicjowaniu statycznych pól

```
class StaticClass
{
    public static int _field = 1;

    // Static constructor can't have access
    // modifier nor parameters
    static StaticClass()
    {
        Console.WriteLine(_field);
    }
}
```

- Operatory przeładowuje się poprzez implementację odpowiedniej metody statycznej
- Przeładowanie niektórych operatorów wiąże się z koniecznością przeładowania innych
- Listę operatorów które można przeładować:
[https://msdn.microsoft.com/en-us/library/8edha89s\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/8edha89s(v=vs.71).aspx)

```
class PlusOperator
{
    private int x;
    public PlusOperator(int x) { this.x = x; }
    public static PlusOperator operator
        +(PlusOperator p1, PlusOperator p2)
    {
        return new PlusOperator(p1.x+p2.x);
    }
}
```

Podstawy

Tworzenie
typówRelacje między
typamiKonwersje i
rzutowania

Konwersje 'implicit'

Konwersje 'explicit'
i rzutowanieBezpieczne
rzutowanie i
operator asPrzeladowanie
operatorów
rzutowania

Boxing i Unboxing

Funkcje i
metody

Generyki

Kolekcje

Konwersje implicit stosują się wtedy gdy nie istnieje ryzyko błędu.

```
int x = 1337;  
double d = x;  
  
string s = "str";  
object o = s;
```


Podstawy

Tworzenie
typówRelacje między
typamiKonwersje i
rzutowania

Konwersje 'implicit'

Konwersje 'explicit'
i rzutowanieBezpieczne
rzutowanie i
operator asPrzeladowanie
operatorów
rzutowania

Boxing i Unboxing

Funkcje i
metody

Generyki

Kolekcje

Konwersje explicit stosują się wtedy gdy typy są 'kompatybilne' ale ponieważ kompiłator nie może stwierdzić czy rzutowanie jest bezpieczne wymaga od nas jasnej deklaracji. Jeżeli w trakcie wykonywania programu okaże się, że rzeczywiste typy nie były kompatybilne, ruczany jest *InvalidCastException*.

```
double d = 1.7;
int x = (int)d;
// x == 1

object o = "str";
string s = (String)o;
```

Podstawy

Tworzenie
typówRelacje między
typamiKonwersje i
rzutowanie

Konwersje 'implicit'

Konwersje 'explicit'
i rzutowanieBezpieczne
rzutowanie i
operator asPrzeladowanie
operatorów
rzutowania

Boxing i Unboxing

Funkcje i
metody

Generyki

Kolekcje

- Za pomocą operatora `is` może sprawdzić typ instancji w trakcie wykonywania programu.
- Za pomocą operatora `as` możemy dokonać bezpiecznego rzutowania typów referencyjnych.

```
object o = "text";  
if(o is string)  
    string s = (string)o;
```

```
o = new object();  
string s = o as string;  
// s == null
```

```
// Compile error on:  
// int x = o as int;
```

```
class CastingClass
{
    public int X { get; set; }
    public static explicit operator
        CastingClass(int i)
    {
        return new CastingClass { X = i };
    }

    public static implicit operator
        int(CastingClass c)
    {
        return c.X;
    }
}
```

Podstawy

Tworzenie
typówRelacje między
typamiKonwersje i
rzutowania

Konwersje 'implicit'

Konwersje 'explicit'
i rzutowanieBezpieczne
rzutowanie i
operator asPrzeladowanie
operatorów
rzutowania

Boxing i Unboxing

Funkcje i
metody

Generyki

Kolekcje

- Boxing następuje gdy przypisujemy instancję typu wartościowego do zmiennej typu referencyjnego (object lub jakiś interfejs)
- Unboxing to proces odwrotny do Boxingu
- Boxing wykonywany jest implicite a unboxing explicite

```
object o = 1;  
int i = (int) o;
```

- Domyślnie metody i właściwości są niewirtualne.
- Zwirtualizowanie metody następuje poprzez użycie `virtual` w klasie bazowej
- Przesłonięcie w klasie pochodnej następuje poprzez użycie `override`

```
class VirtualFun
{
    public virtual int VirtProp { get; set; }
    public void Meth(){}
}

class VirtualFun2: VirtualFun
{
    public override int VirtProp { get
        {return 1;} }
}
```

Podstawy

Tworzenie
typówRelacje między
typamiDziedziczenie
Metody wirtualneUkrywanie i
modyfikator newPlombowanie
Klasy abstrakcyjneImplementowanie
interfejsówImplementowanie
interfejsów expliciteFunkcje i
metody

Generyki

Kolekcje

```
class VirtualFun
{
    public virtual void Meth(){}
    public virtual void Meth2(){}
    public void Meth3(){}
    public void Meth4(){}
}

class VirtualFun2: VirtualFun
{
    // Overriding of virtual method
    public override void Meth(){}
    // Hiding of virtual method - WARNING
    public void Meth2(){}
    // Hiding of nonvirtual method - WARNING
    public void Meth3(){}
    // Explicit hiding of any method
    public new void Meth4(){}
}
```

Podstawy

Tworzenie
typówRelacje między
typami

Dziedziczenie

Metody wirtualne

Ukrywanie i
modyfikator new**Plombowanie**

Klasy abstrakcyjne

Implementowanie
interfejsówImplementowanie
interfejsów expliciteFunkcje i
metody

Generyki

Kolekcje

```
class VirtualFun
{
    public virtual void Meth(){}
}

class VirtualFun2: VirtualFun
{
    public override sealed void Meth(){}
}

class VirtualFun3: VirtualFun2
{
    public new void Meth() {}
}
```

Podstawy

Tworzenie
typówRelacje między
typami

Dziedziczenie

Metody wirtualne

Ukrywanie i
modyfikator new

Plombowanie

Klasy abstrakcyjne

Implementowanie
interfejsówImplementowanie
interfejsów expliciteFunkcje i
metody

Generyki

Kolekcje

Metody abstrakcyjne są automatycznie wirtualne

```
abstract class AbstractFun
{
    public abstract void Meth();
}

class VirtualFun : AbstractFun
{
    public override void Meth() { }
}

class DevirtualizedFun : AbstractFun
{
    public sealed override void Meth(){ }
}
```


Wszyscy członkowie interfejsu są wirtualni. Ich implementacje są domyślnie dewirtualizowane.

```
interface IFun
{
    void Meth1();
    void Meth2();
}

class FunImpl : IFun
{
    public virtual void Meth1() { }
    // Implicit sealed override
    public void Meth2() { }
}
```

```
interface IFun
{
    void Meth();
}
class FunImpl : IFun
{
    void IFun.Meth() { }

    private static void Main(string[] args)
    {
        FunImpl impl = new FunImpl();
        // ERROR
        // impl.Meth();

        IFun fun = impl;
        fun.Meth();
    }
}
```

Parametry przekazywane przez wartość

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metodyWywoływanie funkcji
i metodParametry
przekazywane przez
wartośćParametry
przekazywane przez
referencjeFunkcje ze zmienną
liczbą argumentów

Domyślne wartości

Generyki

Kolekcje

```
void Method1(int i)
{
    i = 1;
}

void Method2(string s)
{
    s = "2";
}

void Test()
{
    string str = "1";
    Method2(str);
    Debug.Assert(str == "1");
}
```

Parametry przekazywane przez referencje

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metodyWywoływanie funkcji
i metodParametry
przekazywane przez
wartośćParametry
przekazywane przez
referencjeFunkcje ze zmienną
liczbą argumentów

Domyślne wartości

Generyki

Kolekcje

```
void Method1(out int i)
{
    // ERROR
    // if (i == 0)
        i = 1;
}

void Method2(ref string s)
{
    if (s == "1")
        s = "2";
}

void Test()
{
    string str = "1";
    Method2(ref str);
    Debug.Assert(str == "2");
}
```

Funkcje ze zmienną liczbą argumentów

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metodyWywoływanie funkcji
i metodParametry
przekazywane przez
wartośćParametry
przekazywane przez
referencjeFunkcje ze zmienną
liczbą argumentów

Domyślne wartości

Generyki

Kolekcje

```
int Sum(params int[] ints)
{
    return ints.Sum();
}

void Test()
{
    int a = Sum();
    Debug.Assert(a == 0);
    int b = Sum(1);
    Debug.Assert(b == 1);
    int c = Sum(1, 2, 3, 4);
    Debug.Assert(c == 10);
    int d = Sum(new[] {1, 2, 3, 4, 5});
    Debug.Assert(d == 15);
}
```

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metodyWywoływanie funkcji
i metodParametry
przekazywane przez
wartośćParametry
przekazywane przez
referencjeFunkcje ze zmienną
liczbą argumentów

Domyślne wartości

Generyki

Kolekcje

```
void Meth(int i = 0, string str = null)
{
}

void Test()
{
    double d = 0;
    Meth(1, "");
    Meth(1);
    Meth(str: "");
    Meth(str: "", i: 1);
}
```

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Metody anonimowe

Generyki

Kolekcje

```
private delegate void Del(int i);

private static void Main(string[] args)
{
    Del del = delegate(int i) { /* ... */ };
    Del del2 = delegate { /* ... */ };
}
```

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Wyrażenia lambda

Generyki

Kolekcje

```
Func<bool> l1 = () => true;  
  
Func<int, bool> l2 = x => x == 2;  
  
Func<int, int, int> l3 = (x, y) => x + y;  
  
Action<int> l4 = x => Debug.Write(x);  
  
Action l5 = () =>  
{  
    int x = 0;  
    Debug.Write(x);  
};
```



```
void NonGenericSwap(ref object obj1, ref
    object obj2)
{
    object temp = obj1; obj1 = obj2; obj2 =
        temp;
}
void Swap<T>(ref T obj1, ref T obj2)
{
    T temp = obj1; obj1 = obj2; obj2 = temp;
}
void Main(string[] args)
{
    int a = 0, b = 1;
    // ERROR
    // NonGenericSwap(ref a, ref b);
    Swap(ref a, ref b);
    Debug.Assert(a == 1 && b == 0);
}
```

where T: struct

Typ T jest typem wartościowym

where T: class

Typ T jest typem referencyjnym

where T: new()

Typ T ma publiczny konstruktor bezparametrowy,
występuje na końcu

where T: BaseClassName

Typ T dziedziczy po typie BaseClassName

where T: InterfaceName

Typ T implementuje interfejs IInterfaceName

where T: U

Typ T dziedziczy po generycznym typie U

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Klasy generyczne

Kolekcje

```
class NodeItem<T> where T :  
    System.IComparable<T>, new() { }  
  
class SpecialNodeItem<T> : NodeItem<T> where  
    T : System.IComparable<T>, new() { }  
  
class SuperKeyType<K, V, U>  
    where U : System.IComparable<U>  
    where V : new()  
{ }  
  
// ERROR  
// List<BaseType> l = new List<DerivedType>()  
  
// ERROR  
// class SomeClass<T> : T { }
```

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metodyGeneryki
Interfejsy generyczne

Kolekcje

```
interface IMonth<T> { }
```



```
interface IJanuary          : IMonth<int> { }
```



```
    //No error
```



```
interface IFebruary<T>     : IMonth<int> { }
```



```
    //No error
```



```
interface IMarch<T>        : IMonth<T> { }
```



```
    //No error
```



```
//interface IApril<T>      : IMonth<T, U> {}
```



```
    //Error
```

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Metody generyczne

Kolekcje

```
void Swap<T>(ref T lhs, ref T rhs)
{
    T temp = lhs;
    lhs = rhs;
    rhs = temp;
}

void SwapTest()
{
    int a = 1, b = 2;
    Swap<int>(ref a, ref b);
    Swap(ref a, ref b);
}

void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

```
T GetValueOrDefault<T>()  
{  
    return default(T);  
}
```

Operator default zwraca domyślną wartość:

- Dla typów referencyjnych: `null`
- Dla podstawowych typów wartościowych zwraca reprezentację zera dla danego typu: `0`, `0.0`, `0.0d`, `false`
- Dla enumów zwraca `(TypeEnum)0`
- Dla typów strukturalnych zwraca instancje po wywołaniu domyślnego konstruktora

```
// IEnumerable<T> is covariant
IEnumerable<string> covariantInterface = new
    List<string>();
IEnumerable<object> baseInterface =
    covariantInterface();

// Action delegate is contravariant
Action<object> actObject = x=>{ };
Action<string> actString = actObject;
```

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kowariancja i
kontrawariancja

Przykłady

Interfejsy
kowariacyjneInterfejsy
kontrawariacyjne

Kolekcje

```
interface ICovariant<out T>
{
    T GetSomething();
    void ContraAction(Action<T> callback);
    // ERRORS:
    // void SetSomething(T something)
    // void DoOut(out T something)
}

private static void Main(string[] args)
{
    ICovariant<string> obj = new SomeType();
    ICovariant<object> parent = obj;
    parent.ContraAction(
        (x)=>Console.WriteLine(x) );
}
```


Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kowariancja i
kontrawariancja

Przykłady

Interfejsy
kowariacyjneInterfejsy
kontrawariacyjne

Kolekcje

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // ERROR
    // A GetSomething();
}
```

Podstawy

Tworzenie
typówRelacje między
typamiFunkcje i
metody

Generyki

Kolekcje

Lista generycznych
kolekcji

Dictionary<K,V>	Słownik na tablicy haszującej
HashSet<T>	Zbiór na tablicy haszującej
LinkedList<T>	Lista podwójnie wiązana
List<T>	Lista tablicowa
Queue<T>	Kolejka na tablicy cyklicznej
SortedDictionary<K,V>	Słownik na drzewie binarnym
SortedList<K,V>	Posortowana lista tablicowa
Stack<T>	Stos na tablicy