Programming

Lectures 30 hours

Labs 30 hours

Exam

Lecturer: **Prof. Mariana Goranova, PhD**
Department of Programming and Computer Technologies
Technical University of Sofia
Room: 2304
E-mail: mgor@tu-sofia.bg
URL: http://pct.tu-sofia.bg/moodle001/
Username: student Password: pktt

Bibliography

I. Main

1. **Jeffrey Richter, CLR via C#, Microsoft Press, 2010.**
2. **Tom Archer, Andrew Whitechapel, Inside C#, Second Edition, Microsoft Press, 2002.**
3. **John Sharp, Jon Jagger, Microsoft Visual C# .NET Step by Step, Microsoft Press, 2002.**
4. **Джефри Рихтер, Microsoft .NET Framework – приложно програмиране, СофтПрес ООД, 2002.**
5. **Jesse Liberty, Programming C#, Second Edition, O'Reilly, 2002.**
6. **Charles Petzold, Programming Microsoft Windows with C#, Microsoft Press, 2002.**
7. **Светлин Наков и колектив, Програмиране за .NET Framework, Българска асоциация на разработчиците на софтуер, Фабер, 2004.**
8. **Goranova M., V. Dimitrova, Advanced Software Technologies (C#), Technical University Publishing Complex, Sofia, 2009.**

II. Additional

1. **Developing Microsoft .NET Applications for Windows (Visual C# .NET), MSDN Training, Microsoft Corporation, 2002.**
2. **Damien Watkins, Mark Hammond, Brad Abrams, Programming in the .NET Environment, Microsoft Corporation, 2003.**
3. **Judith Bishop, Nigel Horspool, C# Concisely, Pearson Education Limited, 2004.**

III. Manual

1. **М. Горанова, В. Димитрова, Д. Гоцева, Ръководство по програмиране на C#, ТУ – София, 2006.**

Introducing the .NET Framework

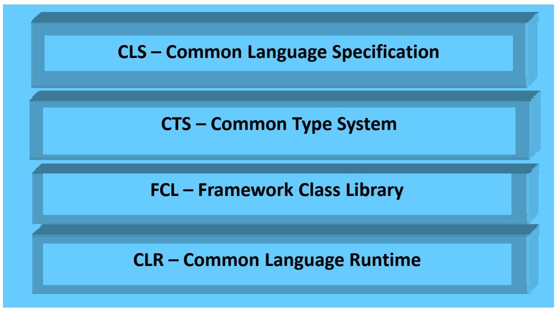
.NET is Microsoft's strategy for developing large distributed software systems.

.NET Framework is a component model for the Internet that allows separate software components written in different languages to be combined to form a functioning system.

.NET Framework could be contrasted with:

1. **Microsoft's Component Object Model (COM) – a component model for the desktop (but not for the large distributed systems).**
2. **Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) – programming model for the Internet, that provides an object-oriented architecture for distributed systems (but does not have a component architecture). CORBA3 extends the model in the component architecture.**
3. **.NET Framework could be compared with Java – a programming language for the Internet with same features that COM, CORBA and .NET, except for a single programming language.**

Architecture of .NET Framework



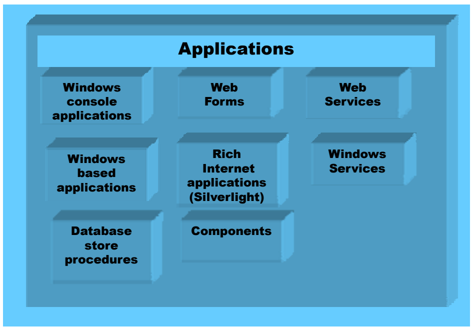
CLS – Common Language Specification

CTS – Common Type System

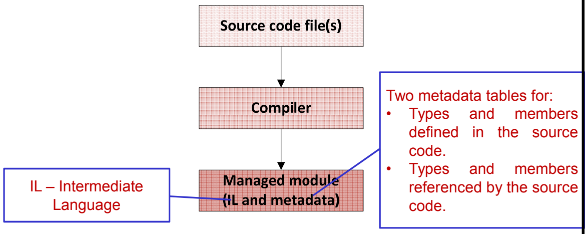
FCL – Framework Class Library

CLR – Common Language Runtime

Architecture of .NET Framework


Applications
Windows console applications
Web Forms
Web Services
Windows based applications
Rich Internet applications (Silverlight)
Windows Services
Database store procedures
Components

CLR – The Main Component of .NET Framework

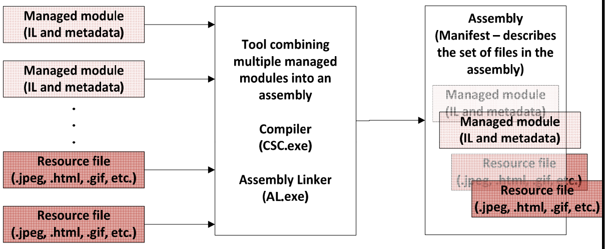


Two metadata tables for:

- Types and members defined in the source code.
- Types and members referenced by the source code.

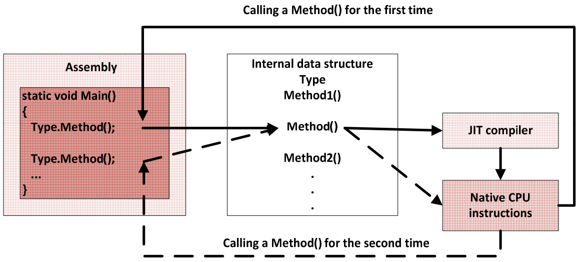
Compiling source code into managed modules

CLR


Managed module (IL and metadata)
Managed module (IL and metadata)
.
.
.
Resource file (.jpeg, .html, .gif, etc.)
Resource file (.jpeg, .html, .gif, etc.)
Tool combining multiple managed modules into an assembly
Compiler (CSC.exe)
Assembly Linker (AL.exe)
Assembly (Manifest – describes the set of files in the assembly)
Managed module (IL and metadata)
Managed module (IL and metadata)
Resource file (.jpeg, .html, .gif, etc.)
Resource file (.jpeg, .html, .gif, etc.)

Combining managed modules into assemblies

CLR



Executing assembly's code

Fundamental Benefits of .NET Framework

1. **Concepts and services remain consistent across all applications.**
   Example: The classes that provide access to a database are the same for all types of applications.
2. **The possibility for substantial reuse exists.**
   Example: A well constructed database access component can be used from many different types of applications without modification (or recompiling).
3. **Support for multiple programming languages – any particular language does not tie you into language-specific libraries and functionality.**

Programming Languages in .NET Framework

1. **Visual Basic .NET**
2. **Visual C++ .NET**
3. **C#**
4. **Python for .NET**
5. **Perl for .NET**
6. **Component Pascal for CLR**
7. **HotDog: Compiling Scheme to Object-Oriented Virtual Machine**
8. **Mondrian (Functional Language)**
9. **Active Oberon for .NET**
10. **J#**
11. **F#**

Introducing the C# Programming Language

Anders Hejlsberg
(Delphi, Java Foundation classes)

Scott Wiltamuth (Microsoft)
Peter Golde (Microsoft)

C# – standard, December 2001
ECMA (European Computer Manufacturer's Association)
ISO 2003

C# Design Goals

1. Designed for .NET Framework
2. Be comfortable for C++ programmers
3. Fit clearly into the .NET CLR (Common Language Runtime)
4. Simplify the C++ model
   - Getting rid of the separate header file and preprocessor
   - Getting rid of memory management issues, by using a reference-based system instead of a pointer-based system, along with the runtime garbage collector
5. Provide the right amount of flexibility
6. Support component-centric development

Characteristics

Very closed to Java

70% Java, 10% C++, 5% Visual Basic, 15% new

As in Java:	As in C++:
Object-oriented (does not support multiple inheritance)	Operator overloading
Interfaces	Pointer arithmetic in unsafe code
Exceptions	Some syntax details
Threads	
Namespaces (as packages)	
Strongly typed	
Garbage collection	
Dynamic code loading	

New characteristics: (towards Java)

Reference and output parameters	Component-oriented development • Properties • Events
Objects, allocated in the stack (struct)	• Delegates • Indexers • Operator overloading
Jagged arrays	Statement foreach
Enumerated data type (enum)	Boxing/unboxing
Common type system	Attributes
Statement goto	
Versioning	

Skeleton Code for C# Application

```
using <namespace>;
namespace <optional_user_namespace>
{
   class <user_class_name>
   {
         public static void Main ()
         {
              // body
         }
   }
}
```

namespace – convenient means of semantically grouping elements

Fully qualify class name:
<namespace>.<class_name>

using – **directive:**

- search path for the listed namespaces (not for classes)
- create aliases for classes

using <alias> = <class_name>

```
using System;                     OK! Using directive with a namespace!
using System.Console;             Error! Using directive with a class!
using output = System.Console;    OK! Use output instead of System.Console!
```

Example:

Write the specified string and the current date and time.

```
using System;
class Welcome
{
    public static void Main ()
    {
        Console.WriteLine ("Welcome!");
        Console.WriteLine ("Today is " +DateTime.Now);
    }
}
```

Class System.Console **– presents the standard streams for console applications.**

Method Console.WriteLine **– writes the specified string followed by a line terminator to the standard output device.**

Output Formatting:

Console.WriteLine("{N[,M][:S]}", argument$_0$, …, argument$_N$);

N **– the position of the argument in the list of values (position numbers start from** 0**)**

M **– (optional) width and justification with added spaces:**

M<0 **or absent – left-justification**

M>0 **– right-justification**

S **– (optional) formatting string – if absent the corresponding** ToString **method defines the formatting:**

Xm**, where** X **– format specifier;** m **– precision**

C/c	**currency**
D/d	**decimal (integers only)**
E/e	**exponential**
F/f	**fixed point**
G/g	**general**
N/n	**number (digits are in groups of three)**
P/p	**percentage**
R/r	**round trip (fixed point only) – correct converting**
X/x	**hexadecimal (integers only)**

Custom format specifiers:

0	**zero placeholder**
#	**digit placeholder**
.	**decimal point**
,	**thousand separator and number scaling**
%	**percentage placeholder**
E+0 E-0 e+0 e-0	**scientific notation**
\	**escape character (new line** \n**)**
'ABC′ "ABC"	**literal string**
;	**section separator – separate sections for positive, negative, and zero numbers in the format string**
argument$_N$	**expression; if it is** null**, an empty string is used**

Structure DateTime **– represents an instant in time, typically expressed as a date and time of day.**

Properties

Now	**(static) gets a** DateTime **object that is set to the current date and time on this computer, expressed as the local time**
Date	**gets the date component of this instance**
TimeOfDay	**gets the time of day for this instance**
Today	**(static) gets the current date**

DateTime formatters:

D	LongDatePattern	ddd, mmmm dd, yyyy
d	ShortDatePattern	mm/dd/yyyy
T	LongTimePattern	hh:mm:ss
t	ShortTimePattern	HH:mm
m, M	MonthDayPattern	mmmm dd

Example:

```
DateTime dt = DateTime.Now;
Console.WriteLine (dt);
Console.WriteLine
        ("Date={0:d}, Time={1:T}. Today is {2:m}",
        dt.Date, dt.TimeOfDay, DateTime.Today);
```

Results:

```
27.22014 11:32:46
Date=27.2.2014, Time=11:32:46.1234567. Today is 27 February
```

Example:

```
using System;
class TestWriteLine
{   static void Main (string[] args)
   {     Console.WriteLine("{0,5},{1:D5}", 123, 456);
         // □□123,00456
         Console.WriteLine("{0,-10:D6},{1,-10:D6}", 123, 456);
         // 000123□□□□,000456□□□□
         Console.WriteLine("{0,-10}{1,-8}", "Name", "Fac.N");
         // Name□□□□□□Fac.N□□□
         Console.WriteLine("------------------");
         // - - - - - - - - - - - - - - - - - -
         Console.WriteLine("{0,-10}{1,8}", "Peter", 123456);
         // Peter□□□□□□□123456
         Console.WriteLine("{0,-10}{1,8}", "Ann", 7890);
         // Ann□□□□□□□□□□□7890
         Console.WriteLine("{0:C}{1,5:F2}", 7890, 5.6);
         // 7□890,00□лв□5,60
```

```
         float f=-123456.7890F;
         Console.WriteLine("{0:$#,##0.00;($#,##0.00);Zero}", f);
         // ($123□456,80)
         int i=1234567890;
         Console.WriteLine("{0:(###) ### - ####}", i);
         // (123) □456□-□7890
         Console.WriteLine("{0:#%}", i);
         // 123456789000%
   }
}
```

Building and Running a Console Application in Visual Studio .NET

1. **Start the Visual Studio .NET.**

2. **File ⇒ New ⇒ Project**

Project Type	⇒ **Visual C# Project**
Templates	⇒ **Console Application**
Location	⇒ **project directory**
Name	⇒ **project name**

3. **View ⇒ Solution Explorer**

   application_file_name.sln – **solution file (one for an application with some projects)**

   project_file_name.csproj – **C# project file (some source files using the same programming language)**

   class_file_name.cs – **C# source file**

   AssemblyInfo.cs – **C# source file for program attributes**

   App.ico – **application icon**

4. **Enter the programming code**

5. **Build and run the application**

   Build ⇒ Build
   Debug ⇒ Start Without Debugging

Building a Console Application using the Command-Line Compiler

C# Compiler

csc.exe

Set the path to the execution file csc.exe

VCVARS32.bat

(C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat**)**

1. Compile

csc application.cs

2. Run

application

Assembly – fundamental part in .NET

- created by the compiler
- contains code that is executed by the system
- a named and versioned collection of modules, exported types, and optionally, resources (from a client's perspective)
- a means of packaging related modules, types, and resources and exporting only what a client should use (from the assembly creator's viewport)

Manifest – contains metadata information; consists of three records:

- .assembly **record to reference an external assembly**
- .assembly **record with information about this assembly**
- .module **record contains the name of the physical file housing the assembly and certain offsets into the file where important information can be located**

Disassemble the executable file – the ILDASM application

Start ⇒ Programs ⇒

Microsoft Visual Studio 2012 ⇒

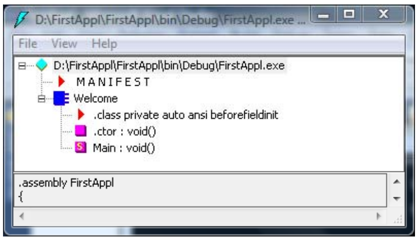
Visual Studio Tools ⇒

Developer Command Prompt for VS2012 ⇒

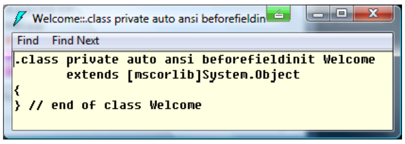
Command windows : ildasm

ILDASM window:

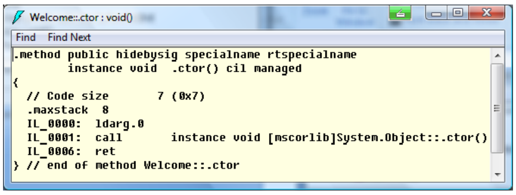
File ⇒ Open ⇒
drive:\...\application_directory\bin\Debug\application.exe


D:\FirstAppl\FirstAppl\bin\Debug\FirstAppl.exe ...
File
View
Help
D:\FirstAppl\FirstAppl\bin\Debug\FirstAppl.exe
MANIFEST
Welcome
.class private auto ansi beforefieldinit
.ctor : void()
Main : void()

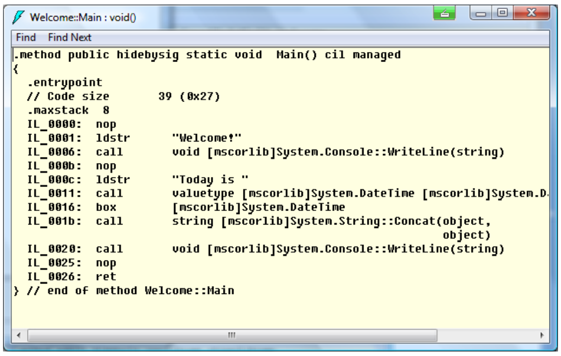
```
.assembly FirstAppl
{
```



```
.class private auto ansi beforefieldinit Welcome
       extends [mscorlib]System.Object
{
} // end of class Welcome
```



```
.method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  ret
} // end of method Welcome::.ctor
```



```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       39 (0x27)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Welcome!"
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  ldstr      "Today is "
  IL_0011:  call       valuetype [mscorlib]System.DateTime [mscorlib]System.D
  IL_0016:  box        [mscorlib]System.DateTime
  IL_001b:  call       string [mscorlib]System.String::Concat(object,
                                                              object)
  IL_0020:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0025:  nop
  IL_0026:  ret
} // end of method Welcome::Main
```

Basic Input/Output Operations

1. **Input operations**

   Method Console.ReadLine **– reads the next line of characters from the standard input stream.**

   Method Console.Parse **– converts the string representation of a number in a specified style to its number equivalent.**

   ```
   string_variable  = Console.ReadLine();
   other_variable   = type.Parse(Console.ReadLine());
   ```

2. **Output operations**

   Methods Console.WriteLine **and** Console.Write**.**

   ```
   Console.WriteLine (data);
   Console.Write (data);
   ```

Example:

```
using System;
class CalculateFee
{
   static void Main (string[] args)
   {
        float dailyRate, fee;
        int noOfDays;
        Console.Write ("Enter daily rate: ");
        dailyRate = float.Parse(Console.ReadLine());
        Console.Write ("Enter number of working days: ");
        noOfDays = int.Parse(Console.ReadLine());
        fee = dailyRate * noOfDays;
        Console.WriteLine("Fee: {0:C}", fee);
   }
}
```

Example: **String conversion into number**

```
int j = int.Parse (" 123456 ");
Console.WriteLine ("j={0}", j);                    // j=123456
float f = float.Parse ("-123,456");
Console.WriteLine ("f={0}", f);                    // f=-123,456
string s = j.ToString ();
Console.WriteLine ("s={0}", s);                    // s=123456
```

Documentation with XML (Extensible Markup Language)

three-slash comments

```
/// <tag>
/// description
/// </tag>
```

Tags:

<summary>	**describe one line specific details for a class, method or property**
<remarks>	**specify overview information about a class or other type**
<value>	**describe a property**
<exception>	**specify which exceptions a member can throw**
<param>	**describe parameters**

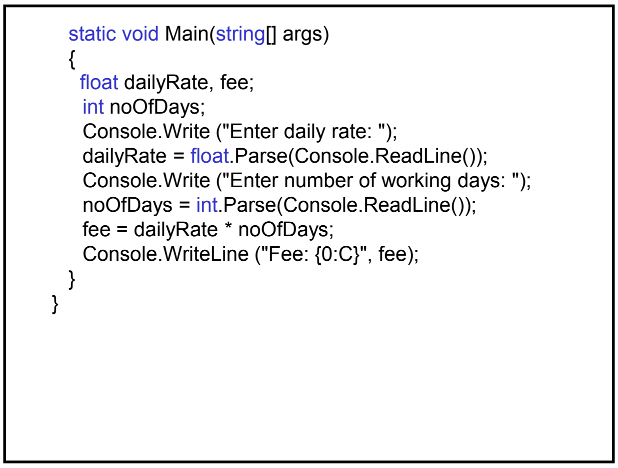
1. **Solution Explorer ⇒ Project ⇒ Properties**
2. **Build ⇒ XML documentation file**

   (bin/Debug/project_name.xml)

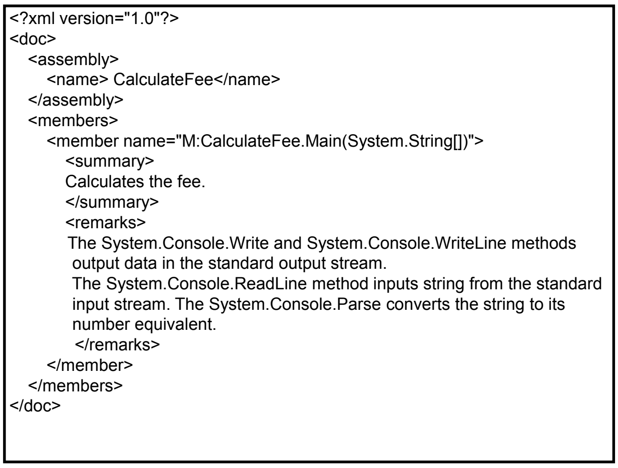
   A structure sequence is created with hyperlinks of HTML documents based on XML.
3. **Solution Explorer ⇒ Show All Files**

Example:

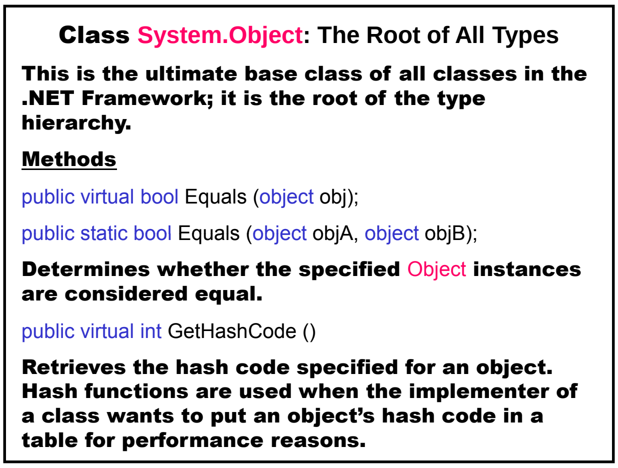
```
using System;
class CalculateFee
{
    /// <summary>
    /// Calculates the fee.
    /// </summary>
    /// <remarks>
    /// The System.Console.Write and
    /// System.Console.WriteLine methods output data in
    /// the standard output stream.
    /// The System.Console.ReadLine method inputs
    /// string from the standard input stream.
    /// The System.Console.Parse converts the
    /// string to its number equivalent.
    /// </remarks>
```



```
      static void Main(string[] args)
      {
       float dailyRate, fee;
       int noOfDays;
       Console.Write ("Enter daily rate: ");
       dailyRate = float.Parse(Console.ReadLine());
       Console.Write ("Enter number of working days: ");
       noOfDays = int.Parse(Console.ReadLine());
       fee = dailyRate * noOfDays;
       Console.WriteLine ("Fee: {0:C}", fee);
      }
    }
```



```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name> CalculateFee</name>
   </assembly>
   <members>
      <member name="M:CalculateFee.Main(System.String[])">
         <summary>
         Calculates the fee.
         </summary>
         <remarks>
         The System.Console.Write and System.Console.WriteLine methods
          output data in the standard output stream.
          The System.Console.ReadLine method inputs string from the standard
          input stream. The System.Console.Parse converts the string to its
          number equivalent.
           </remarks>
      </member>
   </members>
</doc>
```



Class System.Object: The Root of All Types

This is the ultimate base class of all classes in the .NET Framework; it is the root of the type hierarchy.

Methods

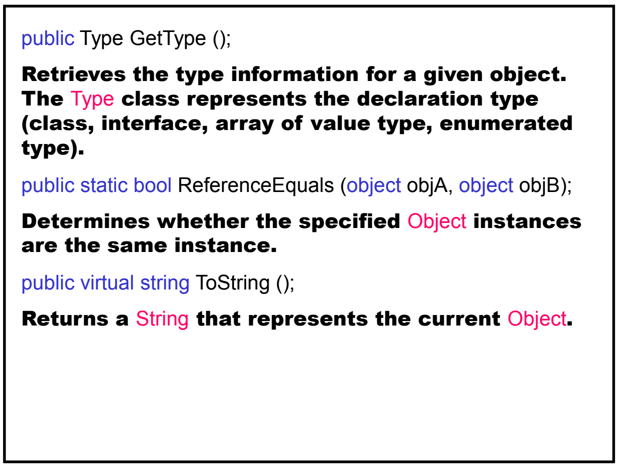
public virtual bool Equals (object obj);

public static bool Equals (object objA, object objB);

Determines whether the specified Object instances are considered equal.

public virtual int GetHashCode ()

Retrieves the hash code specified for an object. Hash functions are used when the implementer of a class wants to put an object's hash code in a table for performance reasons.



public Type GetType ();

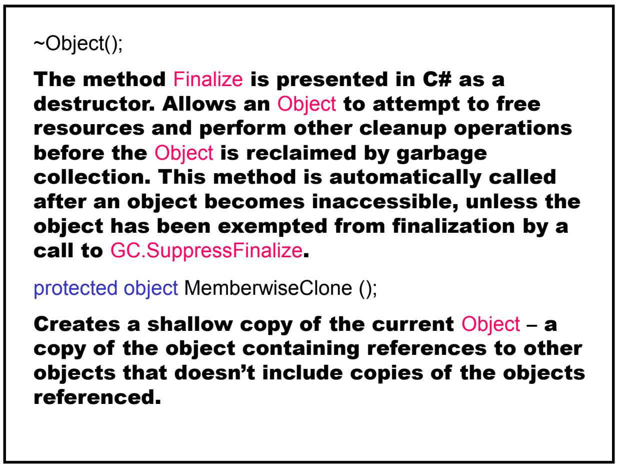
Retrieves the type information for a given object. The Type class represents the declaration type (class, interface, array of value type, enumerated type).

public static bool ReferenceEquals (object objA, object objB);

Determines whether the specified Object instances are the same instance.

public virtual string ToString ();

Returns a String that represents the current Object.

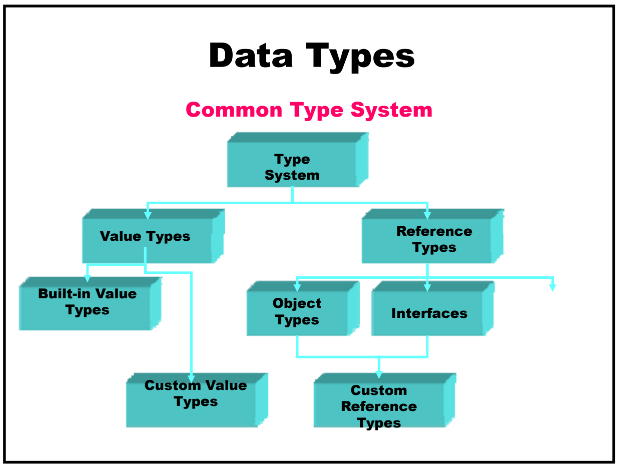


~Object();

The method Finalize is presented in C# as a destructor. Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. This method is automatically called after an object becomes inaccessible, unless the object has been exempted from finalization by a call to GC.SuppressFinalize.

protected object MemberwiseClone ();

Creates a shallow copy of the current Object – a copy of the object containing references to other objects that doesn't include copies of the objects referenced.



Data Types

Common Type System

Data Types

I. Value types

- **Primitive types**
- **Structures**
- **Enumerated types**

1. **Contain actual data – direct access**
2. **Allocated on the stack**
3. **Can't be** null
4. **The variable's value is passed as a parameter – the variable is not modified**

II. Reference types

- **Classes**
- **Arrays**
- **Interfaces**
- **Delegates**

1. **Contain the address of the object – indirect access with a pointer to the object of the type specified**
2. **Allocated on the heap**
3. **Can be** null
4. **The object address is passed as a parameter – the object is modified**

Identifiers – the names that identify the elements in the program

- **letters (uppercase and lowercase) and digits**
- **start with a letter (an underscore _ is a letter)**
- **case sensitive**

result _score twentyOne plan9 TwentyOne

Variables – storage location that holds a value. Using a variable's name to refer to the value it holds.

Microsoft .NET Framework recommendation about variable naming

- **Start the name with a lowercase letter**
- **In a multiword identifier start the second and each subsequent word with an uppercase letter**
  twentyOne
- **Don't use underscores**
- **Don't create identifiers that differ only by case**
  myVariable **and** MyVariable

Primitive Data Types

Data Type	Size [bits]	Range	Examples
byte	8	0÷255	byte b = 42;
short	16	$-2^{16} \div 2^{16}-1$	short s = 42;
int	32	$-2^{31} \div 2^{31}-1$	int count = 42;
long	64	$-2^{63} \div 2^{63}-1$	long wait = 42L;
float	32	$\pm 3.4 \times 10^{38}$	float away = 0.42F;
double	64	$\pm 1.7 \times 10^{308}$	double trouble = 0.42;
decimal	128	28 significant figures	decimal coin = 0.42M; (monetary values)
string	16/character	not applicapble	string vest = ”42”;
char	16	$0 \div 2^{16}-1$	char grill = ’4’;
bool	8	true or false	bool flag = false;

$2^{16}=32768$ $2^{31}=2147483648$ $2^{63}=9223372036854775808$

Declaring Variables

type identifier;

Assignment Operator =

identifier = expression;

Nullable Types (for Value Types)

- **Extensions of all other value types with a null value**
- **Do not have to be declared before they can be used**
- **For each non-nullable value type T there is a corresponding nullable type T?, which can hold an additional value null.**

```
int x = 3;
int? y = 5;
y += x;
if(y==null)
      Console.WriteLine("y=null");
else
      Console.WriteLine(y);                    // 8
```

Implicit Type var

- **Static determination of the type – the compiler determines the type of the variable from the expression on the right side of the initialization statement**
- **Only for declaring local variables**
- **Allows explicit initialization of variables**

var *identifier = expression*;

Dynamic Type dynamic

- **Dynamic determination of the type – at run time**
- **For local variables, fields and arguments of methods**
- **Does not allow explicit initialization of variables**

dynamic *identifier*;

```
using System;
class Program
{
   public static dynamic Add(dynamic x, dynamic y)
   {
      return x + y;
   }
   static void Main()
   {
      var v1 = 5;
      v1 += 6;
      Console.WriteLine(v1);                          // 11
      var v2 = "123";
      v2 += 4;
      Console.WriteLine(v2);                          // 1234
      dynamic sum1 = Add(5, 6);                       // 11
      Console.WriteLine(sum1);
      dynamic sum2 = Add("123","4");
      Console.WriteLine(sum2);                        // 1234
   }
}
```

Overflow-checking for integral-type arithmetic operations and conversions checked/unchecked

In checked context, if an expression produces a value that is outside the range of the destination type, constant expressions cause compile-time errors, and non-constant expressions are evaluated at run-time and raise exceptions.

checked *block*
checked (*expression*)

In unchecked context, if an expression produces a value that is outside the range of the destination type, the result is truncated.

unchecked *block*
unchecked (*expression*)

Type Conversion

1. **Implicit conversion – widening conversion (value of one type is converted to another type that is of equal or greater size)**
2. **Explicit conversion (cast)**

   (*type*) *expression*
   checked ((*type*) *expression*)

   In checked context non-constant expression is evaluated at run-time and raises OverflowException, if arithmetic operation results in an overflow.

Boxing and Unboxing

1. **Boxing – converting a value type instance to a reference type.**

```
int i = 42;                  // value type
object bar = i;              // i is boxed to bar
```

2. **Unboxing – converting a reference type to a value type.**

```
int i = 42;                  // value type
object bar = i;              // i is boxed to bar
int baz = (int)bar;          // unboxed back to int
```

Expressions and Operators

Operator

1. **Symbol that indicates an operation to be performed on operands.**
2. **Result**
   - **a new value from the operation on the operands**
   - **must be stored somewhere in memory**
3. **Operator types – in accordance with number of operands:**
   - **unary**
   - **binary**
   - **ternary**

Operator Precedence and Associativity

1. **Precedence – the order in which the operators are evaluated when a single expression or statement contains multiple operators**
2. **Associativity – left or right side of an expression is evaluated first**

Operator Precedence and Associativity

Category of Operator	Operators	Associativity
Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked	right
Unary	+ - ! ~ ++x –x (T)x	right
Multiplicative	* / &	left
Additive	+ -	left
Shift	<< >>	left
Relational	< > <= >= is as	left
Equality	== !=	left
Logical AND	&	left
Logical XOR	^	left
Logical OR	|	left
Conditional AND	&&	left
Conditional OR	||	left
Conditional	?:	left
Assignment	= += -= *= /= %= >>= <<= &= ^= !=	right
Comma	,	left

Program Flow Control

1. **Selection Statements**
2. **Iteration Statements**
3. **Branching with Jump Statements**

Selection Statements

The if statement

if (*expression*)
  statement$_1$
[else
  statement$_2$]

expression – **any test that produces** bool **result**

The if-else-if statement

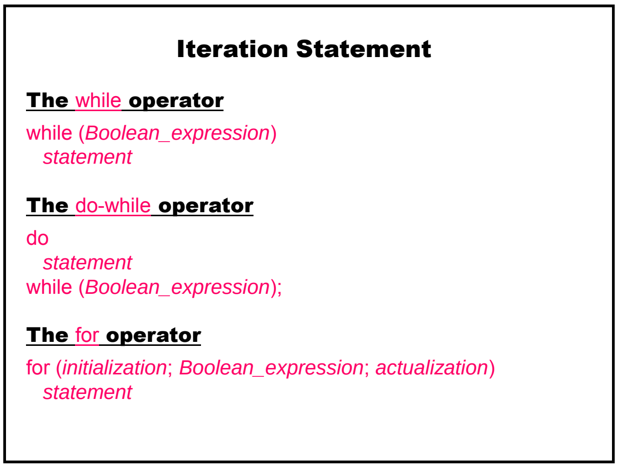
if (*expression*$_1$)
  statement$_1$
else if (*expression*$_2$)
  statement$_2$
…
else
  statement$_n$

The switch statement

switch (*expression*)
{
  case *constant_expression*$_1$: *statement*$_1$
              jump_statement
  …
  case *constant_expression*$_N$: *statement*$_N$
              jump_statement
  [default *statement*$_{N+1}$
        jump_statement]
}

expression – **type** byte**,** short**,** int**,** long**,** char**,** string

jump_statement **(such as** break**) – for each** case **statement**



Iteration Statement

The while operator

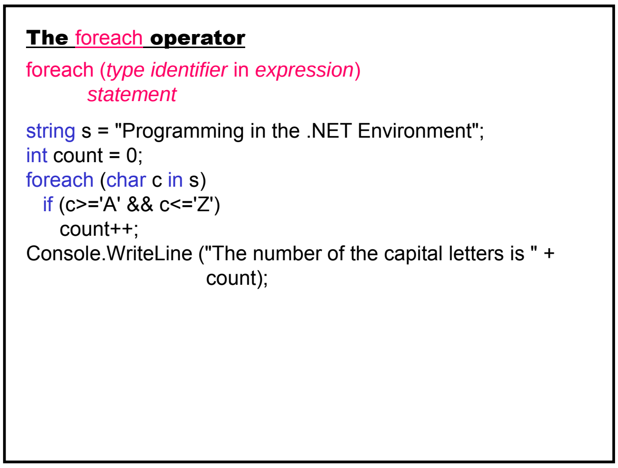
while (*Boolean_expression*)
  statement

The do-while operator

do
  statement
while (*Boolean_expression*);

The for operator

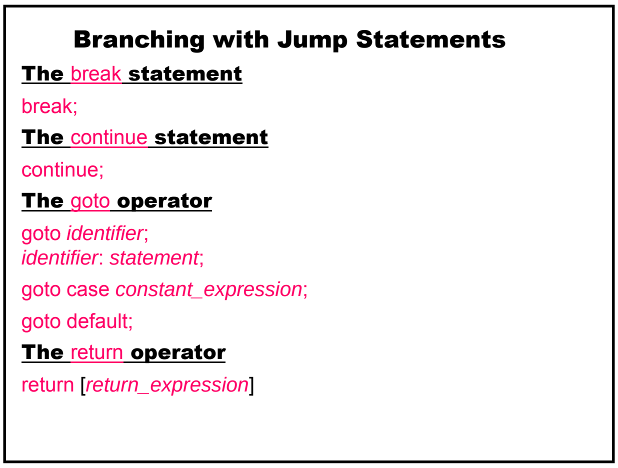
for (*initialization*; *Boolean_expression*; *actualization*)
  statement



The foreach operator

foreach (*type identifier* in *expression*)
        statement

```
string s = "Programming in the .NET Environment";
int count = 0;
foreach (char c in s)
   if (c>='A' && c<='Z')
      count++;
Console.WriteLine ("The number of the capital letters is " +
                           count);
```



Branching with Jump Statements

The break statement

break;

The continue statement

continue;

The goto operator

goto *identifier*;
identifier: *statement*;

goto case *constant_expression*;

goto default;

The return operator

return [*return_expression*]