C# Language Overview (Part II)

Strings, Exceptions, Generics, Collections, Attributes

Abdallah Moujahid
PMP®, COBIT® V5 , ITIL® V3, ISO 27002

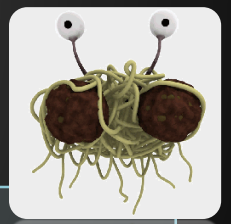
Table of Contents

1. Strings and Text Processing
2. Exceptions Handling
3. Generics
4. Collection Classes
5. Attributes

Strings and Text Processing

What Is String?

- Strings are sequences of characters
- Each character is a Unicode symbol
- Represented by the `string` data type in C# (`System.String`)
- Example:



```
string s = "Hello, C#";
```

The System.String Class

- Strings are represented by `System.String` objects in .NET Framework
  - String objects contain an immutable (read-only) sequence of characters
  - Strings use Unicode in to support multiple languages and alphabets
- `System.String` is reference type

The System.String Class

- String objects are like arrays of characters (`char[]`)
  - Have fixed length (`String.Length`)
  - Elements can be accessed directly by index
    - The index is in the range `[0...Length-1]`

```
string s = "Hello!";
int len = s.Length; // len = 6
char ch = s[1]; // ch = 'e'
```

index =	0	1	2	3	4	5
s[index] =	H	e	l	l	o	!

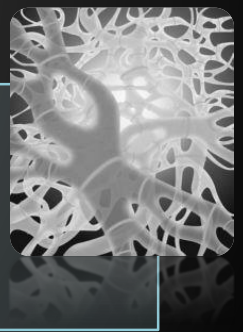
Strings – Example

```
static void Main()
{
    string s =
        "Stand up, stand up, Balkan Superman.";
    Console.WriteLine("s = \"{0}\"", s);
    Console.WriteLine("s.Length = {0}", s.Length);
    for (int i = 0; i < s.Length; i++)
    {
        Console.WriteLine("s[{0}] = {1}", i, s[i]);
    }
}
```

Declaring Strings

- There are two ways of declaring string variables:
  - Using the C# keyword `string`
  - Using the .NET's fully qualified class name `System.String`

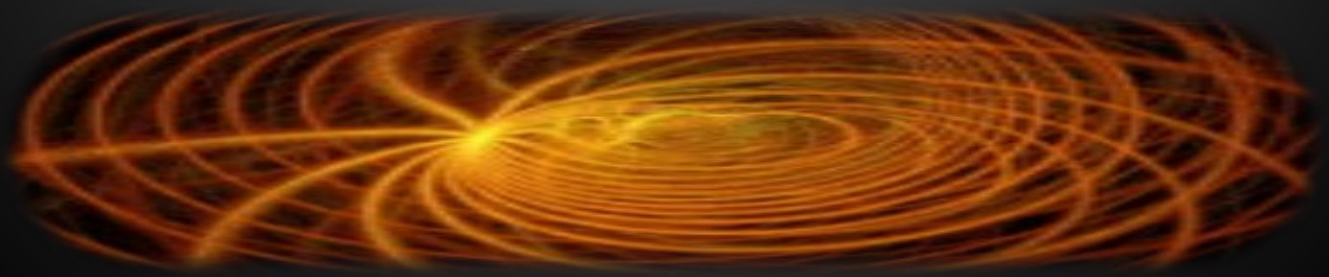
```
string str1;
System.String str2;
String str3;
```



  - The above three declarations are equivalent

Creating Strings

- Before initializing a string variable has `null` value
- Strings can be initialized by:
  - Assigning a string literal to the string variable
  - Assigning the value of another string variable
  - Assigning the result of operation of type string



Creating Strings (2)

- Not initialized variables has value of `null`

```
string s; // s is equal to null
```

- Assigning a string literal

```
string s = "I am a string literal!";
```

- Assigning from another string variable

```
string s2 = s;
```

- Assigning from the result of string operation

```
string s = 42.ToString();
```

Reading and Printing Strings

- Reading strings from the console
  - Use the method `Console.ReadLine()`

```
string s = Console.ReadLine();
```

- Printing strings to the console
  - Use the methods `Write()` and `WriteLine()`

```
Console.Write("Please enter your name: ");
string name = Console.ReadLine();
Console.Write("Hello, {0}! ", name);
Console.WriteLine("Welcome to our party!");
```

Comparing Strings

- A number of ways exist to compare two strings:
  - Dictionary-based string comparison
    - Case-insensitive

```
int result = string.Compare(str1, str2, true);
// result == 0 if str1 equals str2
// result < 0 if str1 if before str2
// result > 0 if str1 if after str2
```

    - Case-sensitive

```
string.Compare(str1, str2, false);
```

Comparing Strings – Example

- Finding the first string in a lexicographical order from a given list of strings:

```
string[] towns = {“Casa", “Rabat", “Tanger",
       “Agadir", “Oujda", “Errachidia", “Dakhla"};
string firstTown = towns[0];
for (int i=1; i<towns.Length; i++)
{
    string currentTown = towns[i];
    if (String.Compare(currentTown, firstTown) < 0)
    {
        firstTown = currentTown;
    }
}
Console.WriteLine("First town: {0}", firstTown);
```

Concatenating Strings

- There are two ways to combine strings:
  - Using the `Concat()` method

```
string str = String.Concat(str1, str2);
```

  - Using the + or the += operators

```
string str = str1 + str2 + str3;
string str += str1;
```

- Any object can be appended to string

```
string name = "Peter";
int age = 22;
string s = name + " " + age; // → "Peter 22"
```

Searching in Strings

- Finding a character or substring within given string
  - First occurrence

```
IndexOf(string str)
```

  - First occurrence starting at given position

```
IndexOf(string str, int startIndex)
```

  - Last occurrence

```
LastIndexOf(string)
```

Searching in Strings – Example

```
string str = "C# Programming Course";
int index = str.IndexOf("C#"); // index = 0
index = str.IndexOf("Course"); // index = 15
index = str.IndexOf("COURSE"); // index = -1
// IndexOf is case-sensetive. -1 means not found
index = str.IndexOf("ram"); // index = 7
index = str.IndexOf("r"); // index = 4
index = str.IndexOf("r", 5); // index = 7
index = str.IndexOf("r", 8); // index = 18
```

index =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
s[index] =	C	#		P	r	o	g	r	a	m	m	i	n	g	…

Extracting Substrings

- Extracting substrings
  - str.Substring(int startIndex, int length)

```
string filename = @"C:\Pics\Rila2009.jpg";
string name = filename.Substring(8, 8);
// name is Rila2009
```

  - str.Substring(int startIndex)

```
string filename = @"C:\Pics\Summer2009.jpg";
string nameAndExtension = filename.Substring(8);
// nameAndExtension is Summer2009.jpg
```

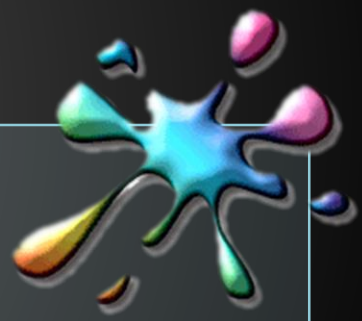
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
C	:	\	P	i	c	s	\	R	i	l	a	2	0	0	9	.	j	p	g

Splitting Strings

- To split a string by given separator(s) use the following method:

```
string[] Split(params char[])
```

- Example:



```
string listOfSodas =
    “Coca, Pepsi, Ace, Poms.";
string[] sodas =
    listOfSodas.Split(' ', ',', '.');
Console.WriteLine("Available sodas are:");
foreach (string soda in sodas)
{
    Console.WriteLine(soda);
}
```

Replacing and Deleting Substrings

- `Replace(string, string)` – replaces all occurrences of given string with another
  - The result is new string (strings are immutable)

```
string cocktail = “Coca + Fanta + Olmès";
string replaced = cocktail.Replace("+", "and");
// Coca and Fanta and Olmès
```

- `Remove(index, length)` – deletes part of a string and produces new string as result

```
string price = "$ 1234567";
string lowPrice = price.Remove(2, 3);
// $ 4567
```

Changing Character Casing

- Using method `ToLower()`

```
string alpha = "aBcDeFg";
string lowerAlpha = alpha.ToLower(); // abcdefg
Console.WriteLine(lowerAlpha);
```

- Using method `ToUpper()`

```
string alpha = "aBcDeFg";
string upperAlpha = alpha.ToUpper(); // ABCDEFG
Console.WriteLine(upperAlpha);
```

Trimming White Space

- Using method `Trim()`

```
string s = "    example of white space    ";
string clean = s.Trim();
Console.WriteLine(clean);
```

- Using method `Trim(chars)`

```
string s = " \t\nHello!!! \n";
string clean = s.Trim(' ', ',' ,'!', '\n','\t');
Console.WriteLine(clean); // Hello
```

- Using `TrimStart()` and `TrimEnd()`

```
string s = "   C#   ";
string clean = s.TrimStart(); // clean = "C#   "
```

Constructing Strings

- Strings are immutable
  - `Concat()`, `Replace()`, `Trim()`, … return new string, do not modify the old one
- Do not use "+" for strings in a loop!
  - It runs very, very inefficiently!

```
public static string DupChar(char ch, int count)
{
    string result = "";
    for (int i=0; i<count; i++)
        result += ch;
    return result;
}
```

Very bad practice. Avoid this!

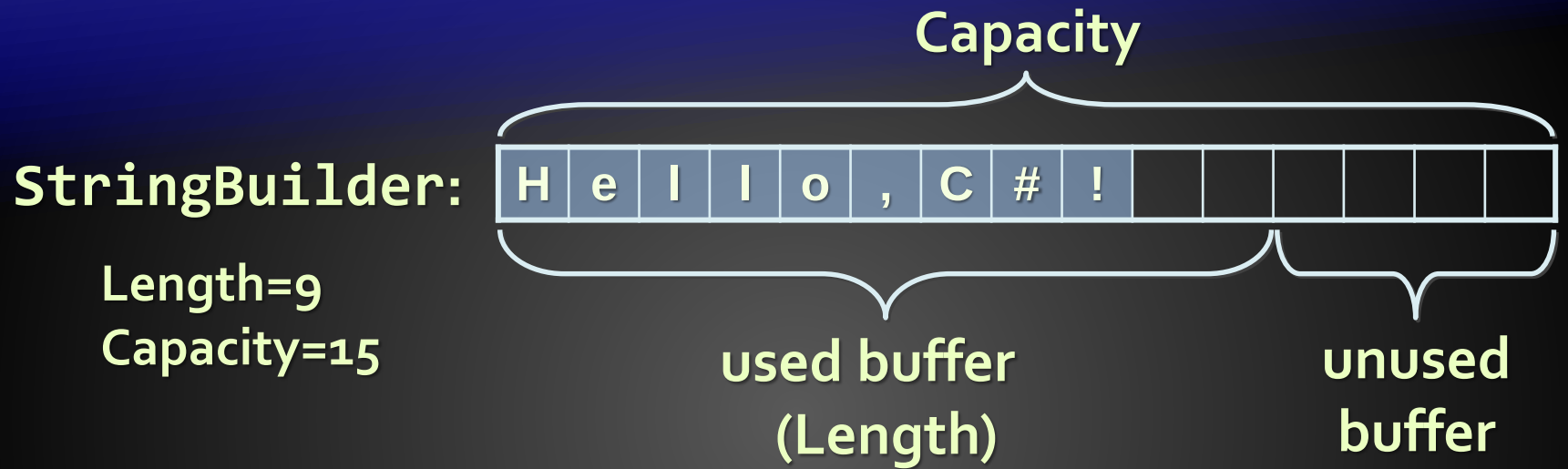
Changing the Contents of a String – StringBuilder

- Use the `System.Text.StringBuilder` class for modifiable strings of characters:

```
public static string ReverseString(string s)
{
    StringBuilder sb = new StringBuilder();
    for (int i = s.Length-1; i >= 0; i--)
        sb.Append(s[i]);
    return sb.ToString();
}
```

- Use `StringBuilder` if you need to keep adding characters to a string

The StringBuilder Class



- `StringBuilder` keeps a buffer memory, allocated in advance
  - Most operations use the buffer memory and do not allocate new objects

Let's practise - Exercise

Extracting all capital letters from a string

StringBuilder – Example

- Extracting all capital letters from a string

```
public static string ExtractCapitals(string s)
{
    StringBuilder result = new StringBuilder();
    for (int i = 0; i<s.Length; i++)
    {
        if (Char.IsUpper(s[i]))
        {
            result.Append(s[i]);
        }
    }
    return result.ToString();
}
```

Method ToString()

- All classes have public virtual method `ToString()`
  - Returns a human-readable, culture-sensitive string representing the object
  - Most .NET Framework types have own implementation of `ToString()`
    - `int`, `float`, `bool`, `DateTime`

```
int number = 5;
string s = "The number is " + number.ToString();
Console.WriteLine(s); // The number is 5
```

Method ToString(format)

- We can apply specific formatting when converting objects to string
  - ToString(formatString) method

```
int number = 42;
string s = number.ToString("D5"); // 00042

s = number.ToString("X"); // 2A


double d = 0.375;
s = d.ToString("P2"); // 37,50 %
```

Formatting Strings

- The formatting strings are different for the different types
- Some formatting strings for numbers:
  - D – number (for integer types)
  - C – currency (according to current culture)
  - E – number in exponential notation
  - P – percentage
  - X – hexadecimal number
  - F – fixed point (for real numbers)

Method String.Format()

- Applies templates for formatting strings
  - Placeholders are used for dynamic text
  - Like `Console.WriteLine(…)`

```
string template = "If I were {0}, I would {1}.";
string sentence1 = String.Format(
    template, "developer", "know C#");
Console.WriteLine(sentence1);
// If I were developer, I would know C#.

string sentence2 = String.Format(
    template, "elephant", "weigh 4500 kg");
Console.WriteLine(sentence2);
// If I were elephant, I would weigh 4500 kg.
```

Composite Formatting

- The placeholders in the composite formatting strings are specified as follows:

```
{index[,alignment][:formatString]}
```

- Examples:

```
double d = 0.375;
s = String.Format("{0,10:F5}", d);
// s = "   0,37500"

int number = 42;
Console.WriteLine("Dec {0:D} = Hex {1:X}",
    number, number);
// Dec 42 = Hex 2A
```

Formatting Dates

- Dates have their own formatting strings
  - d, dd – day (with/without leading zero)
  - M, MM – month
  - yy, yyyy – year (2 or 4 digits)
  - h, HH, m, mm, s, ss – hour, minute, second

```
DateTime now = DateTime.Now;
Console.WriteLine(
    "Now is {0:d.MM.yyyy HH:mm:ss}", now);
// Now is 31.11.2009 11:30:32
```

Exceptions Handling

The Paradigm of Exceptions in OOP

What are Exceptions?

- The exceptions in .NET Framework are classic implementation of the OOP exception model
- Deliver powerful mechanism for centralized handling of errors and unusual events
- Substitute procedure-oriented approach, in which each function returns error code
- Simplify code construction and maintenance
- Allow the problematic situations to be processed at multiple levels

Handling Exceptions

- In C# the exceptions can be handled by the `try-catch-finally` construction

```
try
{
    // Do some work that can raise an exception
}
catch (SomeException)
{
    // Handle the caught exception
}
```

- `catch` blocks can be used multiple times to process different exception types

Handling Exceptions – Example

```
static void Main()
{
    string s = Console.ReadLine();
    try
    {
        int.Parse(s);
        Console.WriteLine(
            "You entered valid int number {0}.", s);
    }
    catch (FormatException)
    {
        Console.WriteLine("Invalid integer number!");
    }
    catch (OverflowException)
    {
        Console.WriteLine(
            "The number is too big to fit in int!");
    }
}
```

The System.Exception Class

- Exceptions in .NET are objects
- The `System.Exception` class is base for all exceptions in CLR
  - Contains information for the cause of the error or the unusual situation
    - `Message` – text description of the exception
    - `StackTrace` – the snapshot of the stack at the moment of exception throwing

Exception Properties – Example

```
class ExceptionsTest
{
  public static void CauseFormatException()
  {
    string s = "an invalid number";
    int.Parse(s);
  }
  static void Main()
  {
    try
    {
      CauseFormatException();
    }
    catch (FormatException fe)
    {
      Console.Error.WriteLine("Exception caught: {0}\n{1}",
        fe.Message, fe.StackTrace);
    }
  }
}
```

Exception Properties

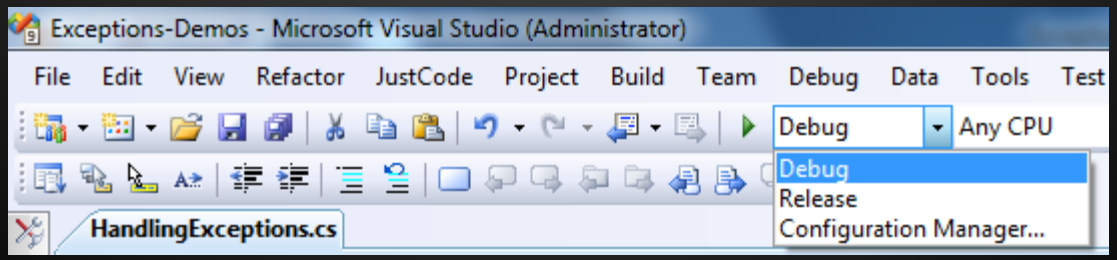
- The `Message` property gives brief description of the problem
- The `StackTrace` property is extremely useful when identifying the reason caused the exception

```
Exception caught: Input string was not in a correct format.
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   at System.int.Parse(String s)
   at ExceptionsTest.CauseFormatException() in c:\consoleapplication1\exceptionstest.cs:line 8
   at ExceptionsTest.Main(String[] args) in c:\consoleapplication1\exceptionstest.cs:line 15
```

Exception Properties (2)

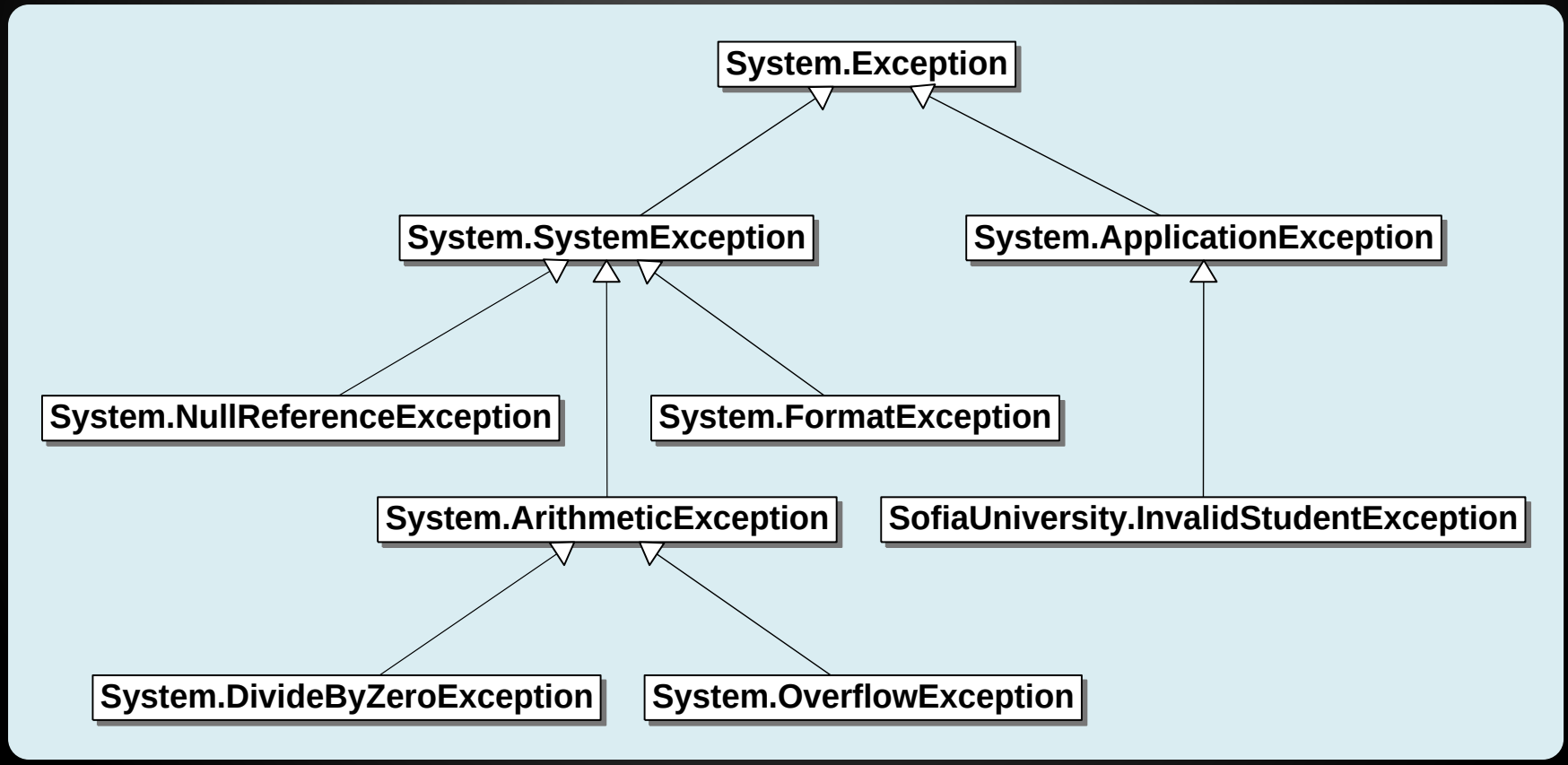
- File names and line numbers are accessible only if the compilation was in Debug mode
- When compiled in Release mode, the information in the property `StackTrace` is quite different:

```
Exception caught: Input string was not in a correct
format.
   at System.Number.ParseInt32(String s, NumberStyles
style, NumberFormatInfo info)
   at ExceptionsTest.Main(String[] args)
```



Exception Hierarchy

- Exceptions in .NET Framework are organized in a hierarchy

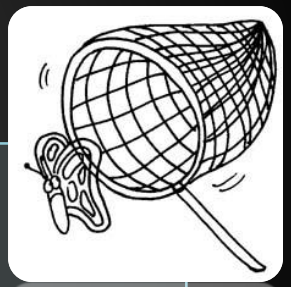


Types of Exceptions

- All .NET exceptions inherit from `System.Exception`
- The system exceptions inherit from `System.SystemException`, e.g.
  - `System.ArgumentException`
  - `System.NullReferenceException`
  - `System.OutOfMemoryException`
  - `System.StackOverflowException`
- User-defined exceptions should inherit from `System.ApplicationException`

Handling Exceptions

- When catching an exception of a particular class, all its inheritors (child exceptions) are caught too
- Example:



```
try
{
    // Do some works that can raise an exception
}
catch (System.ArithmeticException)
{
    // Handle the caught arithmetic exception
}
```

Handles `ArithmeticException` and its successors `DivideByZeroException` and `OverflowException`

Handling All Exceptions

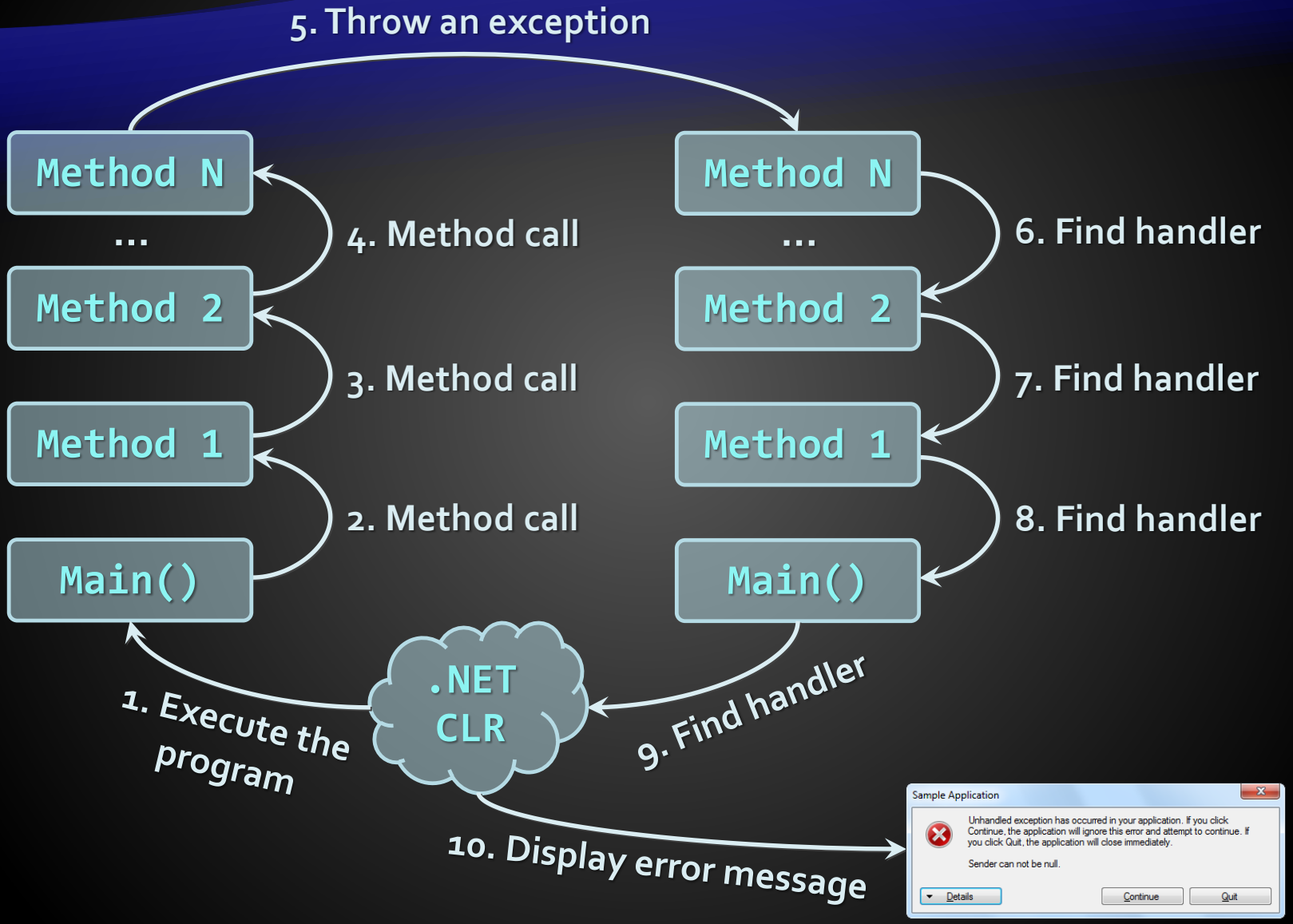
- All exceptions thrown by .NET managed code inherit the `System.Exception` exception
- Unmanaged code can throw other exceptions
- For handling all exceptions (even unmanaged) use the construction:

```
try
{
    // Do some works that can raise any exception
}
catch
{
    // Handle the caught exception
}
```

Throwing Exceptions

- Exceptions are thrown (raised) by `throw` keyword in C#
  - Used to notify the calling code in case of error or unusual situation
- When an exception is thrown:
  - The program execution stops
  - The exception travels over the stack until a suitable `catch` block is reached to handle it
- Unhandled exceptions display error message

How Exceptions Work?



Using throw Keyword

- Throwing an exception with error message:

```
throw new ArgumentException("Invalid amount!");
```

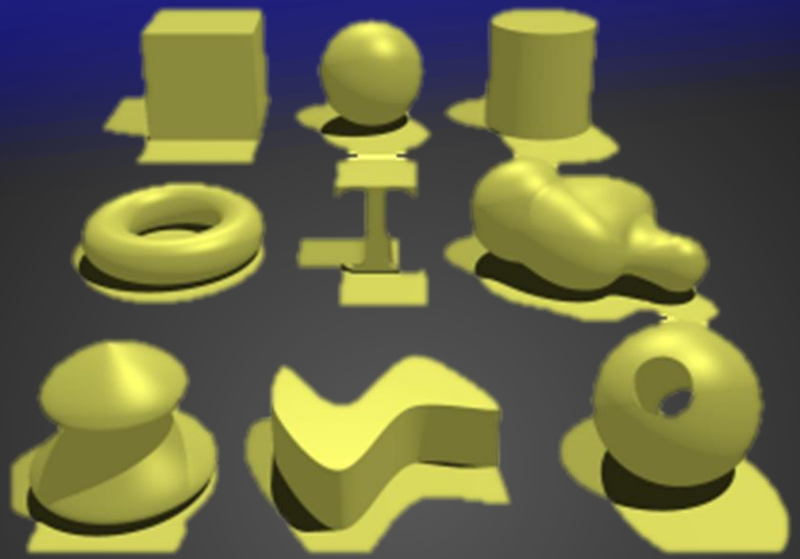
- Exceptions can take message and cause:

```
try
{
  Int32.Parse(str);
}
catch (FormatException fe)
{
  throw new ArgumentException("Invalid number", fe);
}
```

- Note: if the original exception is not passed the initial cause of the exception is lost

Throwing Exceptions – Example

```
public static double Sqrt(double value)
{
    if (value < 0)
        throw new System.ArgumentOutOfRangeException(
            "Sqrt for negative numbers is undefined!");
    return Math.Sqrt(value);
}
static void Main()
{
    try
    {
        Sqrt(-1);
    }
    catch (ArgumentOutOfRangeException ex)
    {
        Console.Error.WriteLine("Error: " + ex.Message);
        throw;
    }
}
```



Using Classes and Objects

Using the Standard .NET Framework Classes

What is Class?

- The formal definition of class:

> Classes act as templates from which an instance of an object is created at run time. Classes define the properties of the object and the methods used to control the object's behavior.

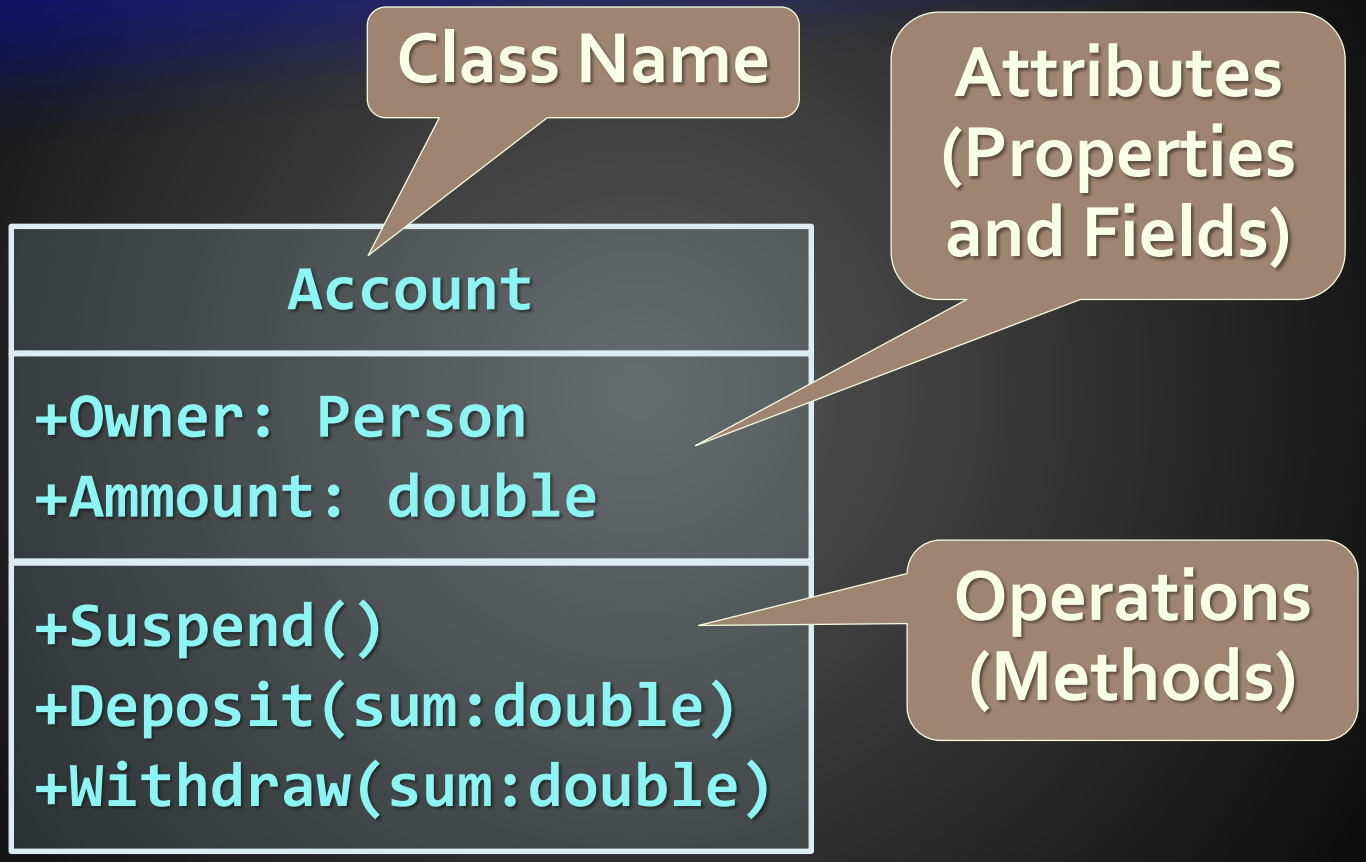
Definition by Google

Classes

- Classes provide the structure for objects
  - Define their prototype, act as template
- Classes define:
  - Set of attributes
    - Represented by fields and properties
    - Hold their state
  - Set of actions (behavior)
    - Represented by methods

- A class defines the methods and types of data associated with an object

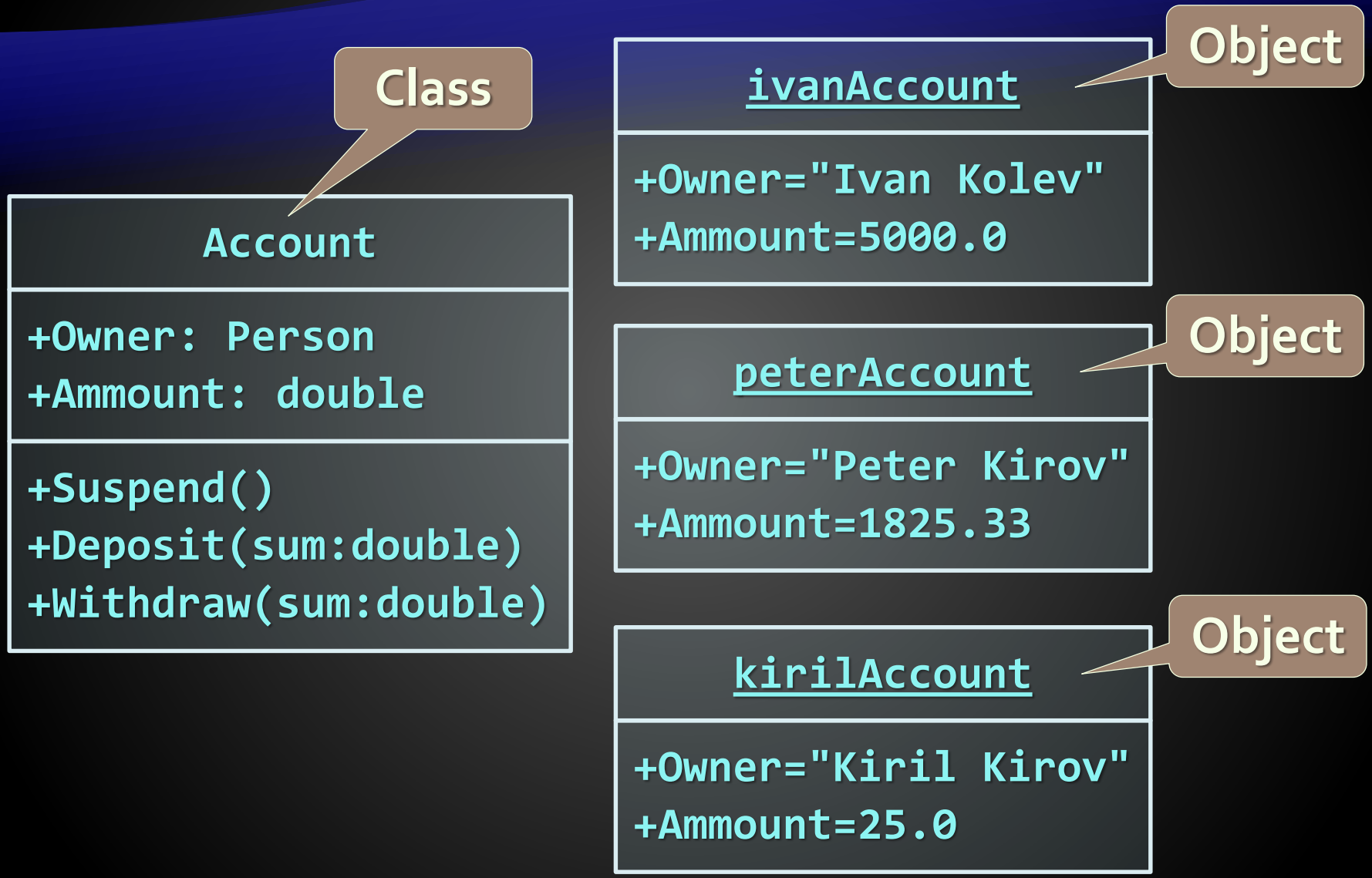
Classes – Example



Objects

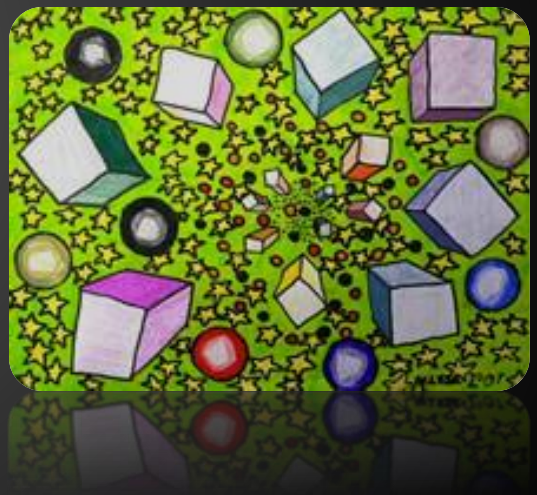
- An object is a concrete instance of a particular class
- Creating an object from a class is called instantiation
- Objects have state
  - Set of values associated to their attributes
- Example:
  - Class: Account
  - Objects: Ivan's account, Peter's account

Objects – Example



Classes in C#

- Basic units that compose programs
- Implementation is encapsulated (hidden)
- Classes in C# can contain:
  - Fields (member variables)
  - Properties
  - Methods
  - Constructors
  - Etc. (events, indexers, operators, …)



Classes in C# – Examples

- Example of classes:
  - System.Console
  - System.String (string in C#)
  - System.Int32 (int in C#)
  - System.Array
  - System.Math
  - System.Random

Declaring Objects

- An instance of a class or structure can be defined like any other variable:

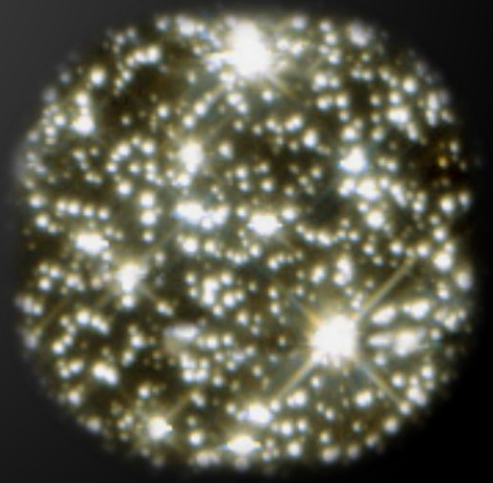
```
using System;
...
// Define two variables of type DateTime
DateTime today;
DateTime halloween;
```

- Instances cannot be used if they are not initialized

```
// Declare and initialize a structure instance
DateTime today = DateTime.Now;
```

Fields

- Fields are data members of a class
- Can be variables and constants
- Accessing a field doesn't invoke any actions of the object
- Example:
  - `String.Empty` (the `""` string)



Accessing Fields

- Constant fields can be only read
- Variable fields can be read and modified
- Usually properties are used instead of directly accessing variable fields
- Examples:

```
// Accessing read-only field
String empty = String.Empty;

// Accessing constant field
int maxInt = Int32.MaxValue;
```

Properties

- Properties look like fields (have name and type), but they can contain code, executed when they are accessed
- Usually used to control access to data fields (wrappers), but can contain more complex logic
- Can have two components (and at least one of them) called accessors
  - get for reading their value
  - set for changing their value

Properties (2)

- According to the implemented accessors properties can be:
  - Read-only (get accessor only)
  - Read and write (both get and set accessors)
  - Write-only (set accessor only)
- Example of read-only property:
  - `String.Length`

Accessing Properties and Fields – Example

```
using System;

...

DateTime christmas = new DateTime(2009, 12, 25);
int day = christmas.Day;
int month = christmas.Month;
int year = christmas.Year;
Console.WriteLine(
  "Christmas day: {0}, month: {1}, year: {2}",
  day, month, year);
Console.WriteLine(
  "Day of year: {0}", christmas.DayOfYear);
Console.WriteLine("Is {0} leap year: {1}",
  year, DateTime.IsLeapYear(year));
```

Instance and Static Members

- Fields, properties and methods can be:
  - Instance (or object members)
  - Static (or class members)
- Instance members are specific for each object
  - Example: different dogs have different name
- Static members are common for all instances of a class
  - Example: `DateTime.MinValue` is shared between all instances of `DateTime`

Instance and Static Members – Examples

- Example of instance member
  - `String.Length`
    - Each string object has different length
- Example of static member
  - `Console.ReadLine()`
    - The console is only one (global for the program)
    - Reading from the console does not require to create an instance of it

Methods

- Methods manipulate the data of the object to which they belong or perform other tasks
- Examples:
  - `Console.WriteLine(…)`
  - `Console.ReadLine()`
  - `String.Substring(index, length)`
  - `Array.GetLength(index)`

Instance Methods

- Instance methods manipulate the data of a specified object or perform any other tasks
  - If a value is returned, it depends on the particular class instance
- Syntax:
  - The name of the instance, followed by the name of the method, separated by dot

```
<object_name>.<method_name>(<parameters>)
```

Calling Instance Methods – Examples

- Calling instance methods of `String`:

```
String sampleLower = new String('a', 5);
String sampleUpper = sampleLower.ToUpper();

Console.WriteLine(sampleLower); // aaaaa
Console.WriteLine(sampleUpper); // AAAAA
```

- Calling instance methods of `DateTime`:

```
DateTime now = DateTime.Now;
DateTime later = now.AddHours(8);

Console.WriteLine("Now: {0}", now);
Console.WriteLine("8 hours later: {0}", later);
```

Static Methods

- Static methods are common for all instances of a class (shared between all instances)
  - Returned value depends only on the passed parameters
  - No particular class instance is available
- Syntax:
  - The name of the class, followed by the name of the method, separated by dot

```
<class_name>.<method_name>(<parameters>)
```

Calling Static Methods – Examples

```
using System;

double radius = 2.9;
double area = Math.PI * Math.Pow(radius, 2);
Console.WriteLine("Area: {0}", area);
// Area: 26,4207942166902

double precise = 8.7654321;
double round3 = Math.Round(precise, 3);
double round1 = Math.Round(precise, 1);
Console.WriteLine(
    "{0}; {1}; {2}", precise, round3, round1);
// 8,7654321; 8,765; 8,8
```

Constant field

Static method

Static method

Static method

Constructors

- Constructors are special methods used to assign initial values of the fields in an object
  - Executed when an object of a given type is being created
  - Have the same name as the class that holds them
  - Do not return a value
- A class may have several constructors with different set of parameters

Constructors (2)

- Constructor is invoked by the new operator

```
<instance_name> = new <class_name>(<parameters>)
```

- Examples:

```
String s = new String("Hello!"); // s = "Hello!"
```

```
String s = new String('*', 5); // s = "*****"
```

```
DateTime dt = new DateTime(2009, 12, 30);
```

```
DateTime dt = new DateTime(2009, 12, 30, 12, 33, 59);
```

```
Int32 value = new Int32(1024);
```

What is a Namespace?

- Namespaces are used to organize the source code into more logical and manageable way
- Namespaces can contain
  - Definitions of classes, structures, interfaces and other types and other namespaces
- Namespaces can contain other namespaces
- For example:
  - `System` namespace contains `Data` namespace
  - The name of the nested namespace is `System.Data`

Full Class Names

- A full name of a class is the name of the class preceded by the name of its namespace

```
<namespace_name>.<class_name>
```

- Example:
  - `Array` class, defined in the `System` namespace
  - The full name of the class is `System.Array`

Including Namespaces

- The using directive in C#:

```
using <namespace_name>
```

- Allows using types in a namespace, without specifying their full name

  Example:

```
using System;
DateTime date;
```

  instead of

```
System.DateTime date;
```

CTS and Different Languages

- CTS is common for all .NET languages
  - C#, VB.NET, J#, JScript.NET, …
- CTS type mappings:

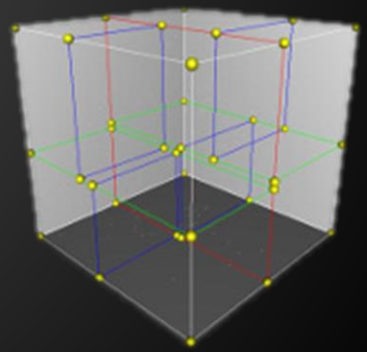
CTS Type	C# Type	VB.NET Type
System.Int32	int	Integer
System.Single	float	Single
System.Boolean	bool	Boolean
System.String	string	String
System.Object	object	Object

Value and Reference Types

- In CTS there are two categories of types
  - Value types
  - Reference types
- Placed in different areas of memory
  - Value types live in the execution stack
    - Freed when become out of scope
  - Reference types live in the managed heap (dynamic memory)
    - Freed by the garbage collector

Value and Reference Types – Examples

- Value types
  - Most of the primitive types
  - Structures
  - Examples: `int`, `float`, `bool`, `DateTime`
- Reference types
  - Classes and interfaces
  - Strings
  - Arrays
  - Examples: `string`, `Random`, `object`, `int[]`



Collection Classes

Lists, Trees, Dictionaries

What are Generics?

- Generics allow defining parameterized classes that process data of unknown (generic) type
  - The class can be instantiated with several different particular types
  - Example: `List<T>` → `List<int>` / `List<string>` / `List<Student>`
- Generics are also known as "parameterized types" or "template types"
  - Similar to the templates in C++
  - Similar to the generics in Java

The List<T> Class

- Implements the abstract data structure list using an array
  - All elements are of the same type T
  - T can be any type, e.g. `List<int>`, `List<string>`, `List<DateTime>`
  - Size is dynamically increased as needed
- Basic functionality:
  - `Count` – returns the number of elements
  - `Add(T)` – appends given element at the end

List<T> – Simple Example

```
static void Main()
{
   List<string> list = new List<string>();

   list.Add("C#");
   list.Add("Java");
   list.Add("PHP");

   foreach (string item in list)
   {
      Console.WriteLine(item);
   }

   // Result:
   //    C#
   //    Java
   //    PHP
}
```

List<T> – Functionality

- `list[index]` – access element by index
- `Insert(index, T)` – inserts given element to the list at a specified position
- `Remove(T)` – removes the first occurrence of given element
- `RemoveAt(index)` – removes the element at the specified position
- `Clear()` – removes all elements
- `Contains(T)` – determines whether an element is part of the list

List<T> – Functionality (2)

- `IndexOf()` – returns the index of the first occurrence of a value in the list (zero-based)
- `Reverse()` – reverses the order of the elements in the list or a portion of it
- `Sort()` – sorts the elements in the list or a portion of it
- `ToArray()` – converts the elements of the list to an array
- `TrimExcess()` – sets the capacity to the actual number of elements

List<T> - Example

```
List<string> jours = new List<string>();

jours.Add("Lundi");
jours.Add("Mardi");
jours.Add("Mercredi");
jours.Add("Jeudi");
jours.Add("Vendredi");
jours.Add("Samedi");
jours.Add("Dimanche");

int indice = jours.IndexOf("Mercredi"); // indice vaut 2

jours.RemoveAt(1);
```

C# Language Overview (Part II)