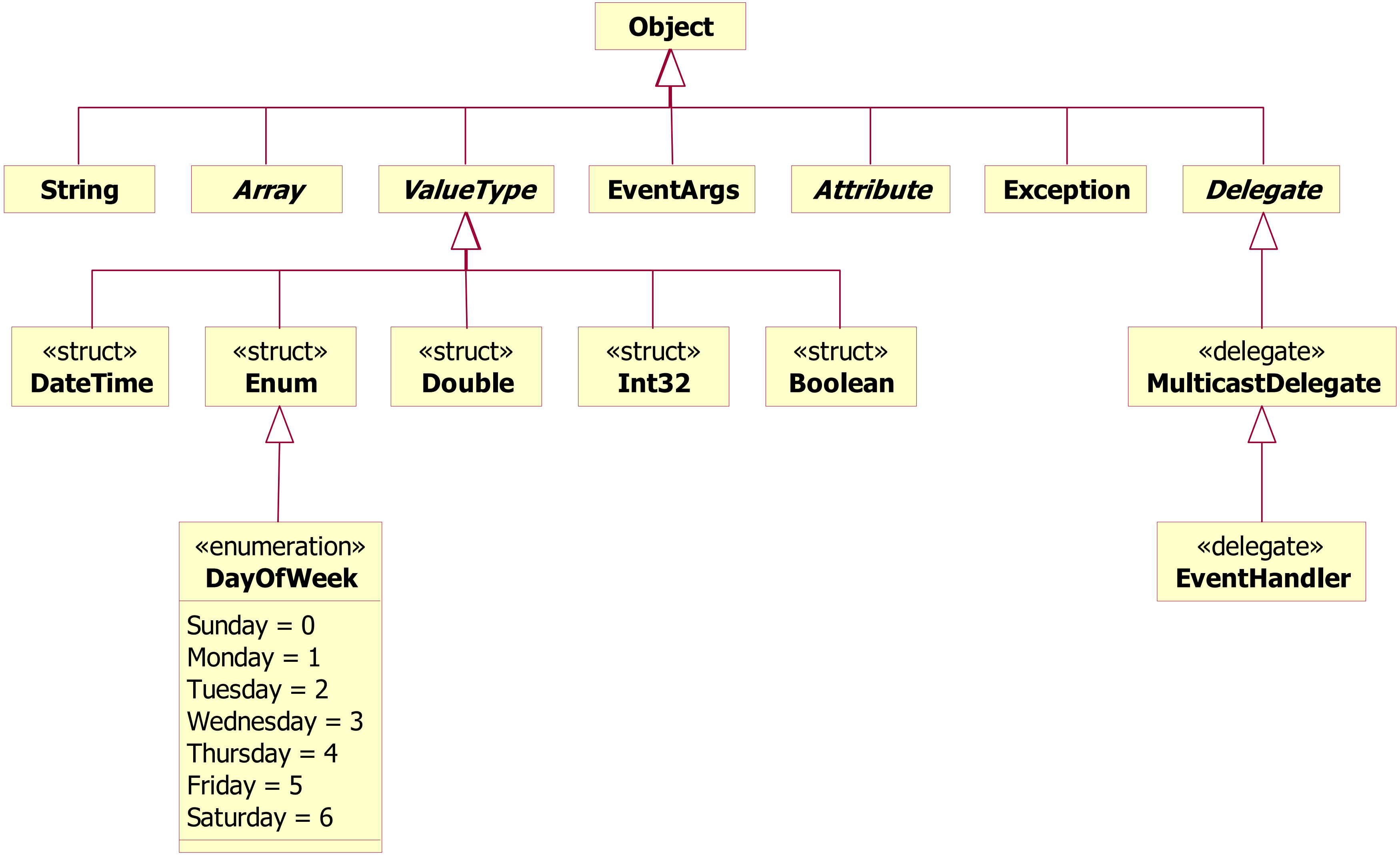
Laboratorio di Ingegneria del Software L-A

Framework .NET

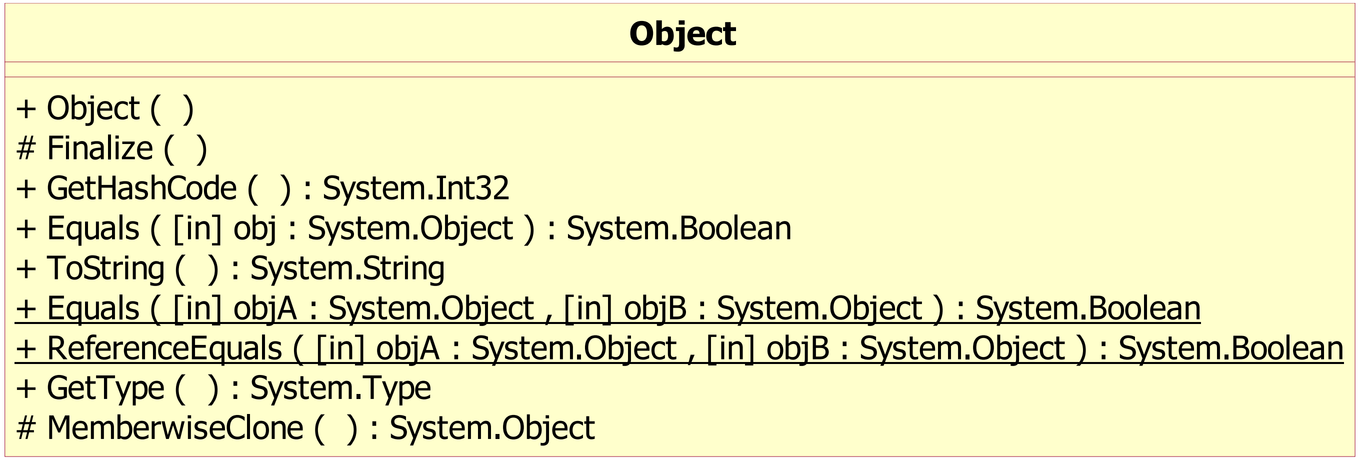
Framework .NET Overview



System.Object

- Supports all classes in the .NET Framework class hierarchy and **provides low-level services to derived classes**
- This is the ultimate superclass of all classes in the .NET Framework; it is the root of the type hierarchy
- Because all classes in the .NET Framework are derived from **`Object`**, **every method defined in the `Object` class is available in all objects in the system**

System.Object



Object

```
+ Object (  )
# Finalize (  )
+ GetHashCode (  ) : System.Int32
+ Equals ( [in] obj : System.Object ) : System.Boolean
+ ToString (  ) : System.String
+ Equals ( [in] objA : System.Object , [in] objB : System.Object ) : System.Boolean
+ ReferenceEquals ( [in] objA : System.Object , [in] objB : System.Object ) : System.Boolean
+ GetType (  ) : System.Type
# MemberwiseClone (  ) : System.Object
```

System.Object

- Derived classes can and do override some of these methods, including:
  - **`Equals`** - Supports comparisons between objects
  - **`ToString`** - Manufactures a human-readable text string that describes an instance of the class
  - **`GetHashCode`** - Generates a number corresponding to the value of the object to support the use of a hash table
  - **`Finalize`** - Performs cleanup operations before an object is automatically reclaimed

`Object.Equals`

- **`public virtual bool Equals(object obj);`**
- Return value: **`true`** if **`this`** is equal to **`obj`** otherwise, **`false`**
- The default implementation of **`Equals`** supports **reference equality** only, but derived classes can override this method to support **value equality**
- For **reference types**, equality is defined as object equality; that is, whether the references refer to the same object
- For **value types**, equality is defined as bitwise equality The **`ValueType`** class supports value types

`Object.Equals`

- The following statements must be **`true`** for all implementations of the **`Equals`** method.
  In the list, **`x`**, **`y`**, and **`z`** represent object references that are not a null reference
  - **`x.Equals(x)`** returns **`true`**
  - **`x.Equals(y)`** returns the same value as **`y.Equals(x)`**
  - **`x.Equals(y)`** returns **`true`** if both **`x`** and **`y`** are **`NaN`**
  - **`(x.Equals(y) && y.Equals(z))`** returns **`true`** if and only if **`x.Equals(z)`** returns **`true`**
  - Successive calls to **`x.Equals(y)`** return the same value as long as the objects referenced by **`x`** and **`y`** are not modified
  - **`x.Equals(`**a **`null`** reference**`)`** returns **`false`**
- Implementations of **`Equals`** must not throw exceptions

`Object.Equals`

- For some kinds of objects, it is desirable to have **`Equals`** test for value equality instead of referential equality
- Such implementations of **`Equals`** return **`true`** if the two objects have the same “value”, even if they are not the same instance
- The type’s implementer decides what constitutes an object’s “value”, but it is typically some or all the data stored in the instance variables of the object
- For example, the value of a **`String`** is based on the characters of the string; the **`Equals`** method of the **`String`** class returns **`true`** for any two string instances that contain exactly the same characters in the same order

`Object.Equals`

- Types that override **`Equals`** must also override **`GetHashCode`**; otherwise, **`Hashtable`** might not work correctly
- If your programming language supports **operator overloading** and if you choose to overload the **equality operator** for a given type, that type should override the **`Equals`** method
  Such implementations of the **`Equals`** method **should return the same results as the equality operator**

Object.Equals

```
public class Point
{
  private readonly int x, y;
  …
  public override bool Equals(object obj)
  {
    //Check for null and compare run-time types.
    if(obj == null || GetType() != obj.GetType())
       return false;
    Point p = (Point) obj;
    return (x == p.x) && (y == p.y);
  }
  public override int GetHashCode()
  {
    return x ^ y;
  }
}
```

Object.Equals

```
public class SpecialPoint : Point
{
  private readonly int w;
  …
  public SpecialPoint(int x, int y, int w) : base(x, y)
  {
    this.w = w;
  }
  public override bool Equals(object obj)
  {
    return base.Equals(obj) &&
      w == ((SpecialPoint) obj).w;
  }
  public override int GetHashCode()
  {
    return base.GetHashCode() ^ w;
  }
}
```

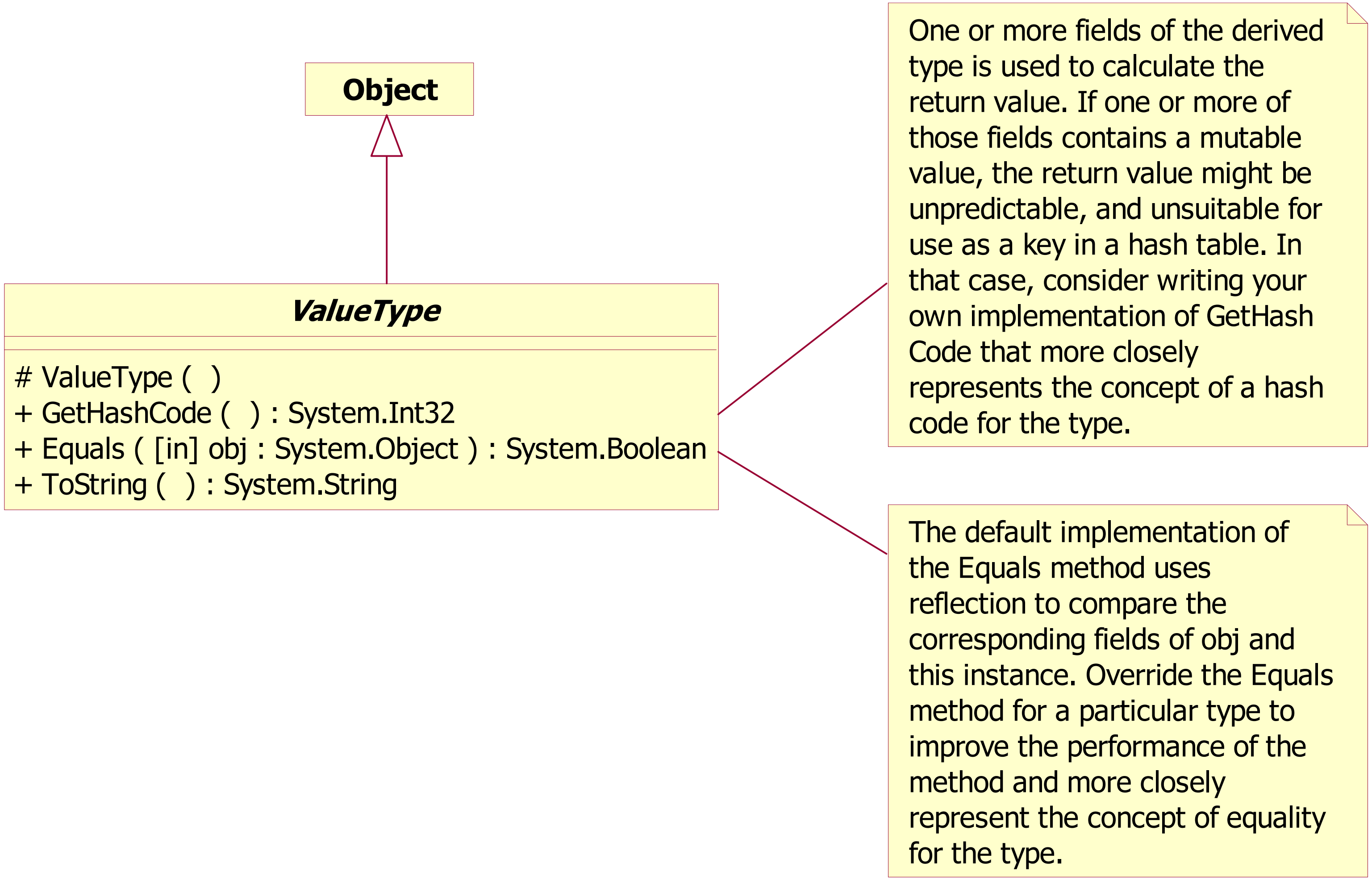
Object.Equals

```
public class Rectangle
{
  private readonly Point a, b;
  …
  public override bool Equals(object obj)
  {
    if(obj == null || GetType() != obj.GetType())
      return false;
    Rectangle r = (Rectangle) obj;
    // Uses Equals to compare variables.
    return a.Equals(r.a) && b.Equals(r.b);
  }
  public override int GetHashCode()
  {
    return a.GetHashCode() ^ b.GetHashCode();
  }
}
```

Object.Equals

```
public struct Complex
{
  private readonly double re, im;
  …
  public override bool Equals(object obj)
  {
    return obj is Complex && this == (Complex) obj;
  }
  public override int GetHashCode()
  {
    return re.GetHashCode() ^ im.GetHashCode();
  }
  public static bool operator ==(Complex x, Complex y)
  {
    return x.re == y.re && x.im == y.im;
  }
  public static bool operator !=(Complex x, Complex y)
  {
    return !(x == y);
  }
}
```

System.ValueType

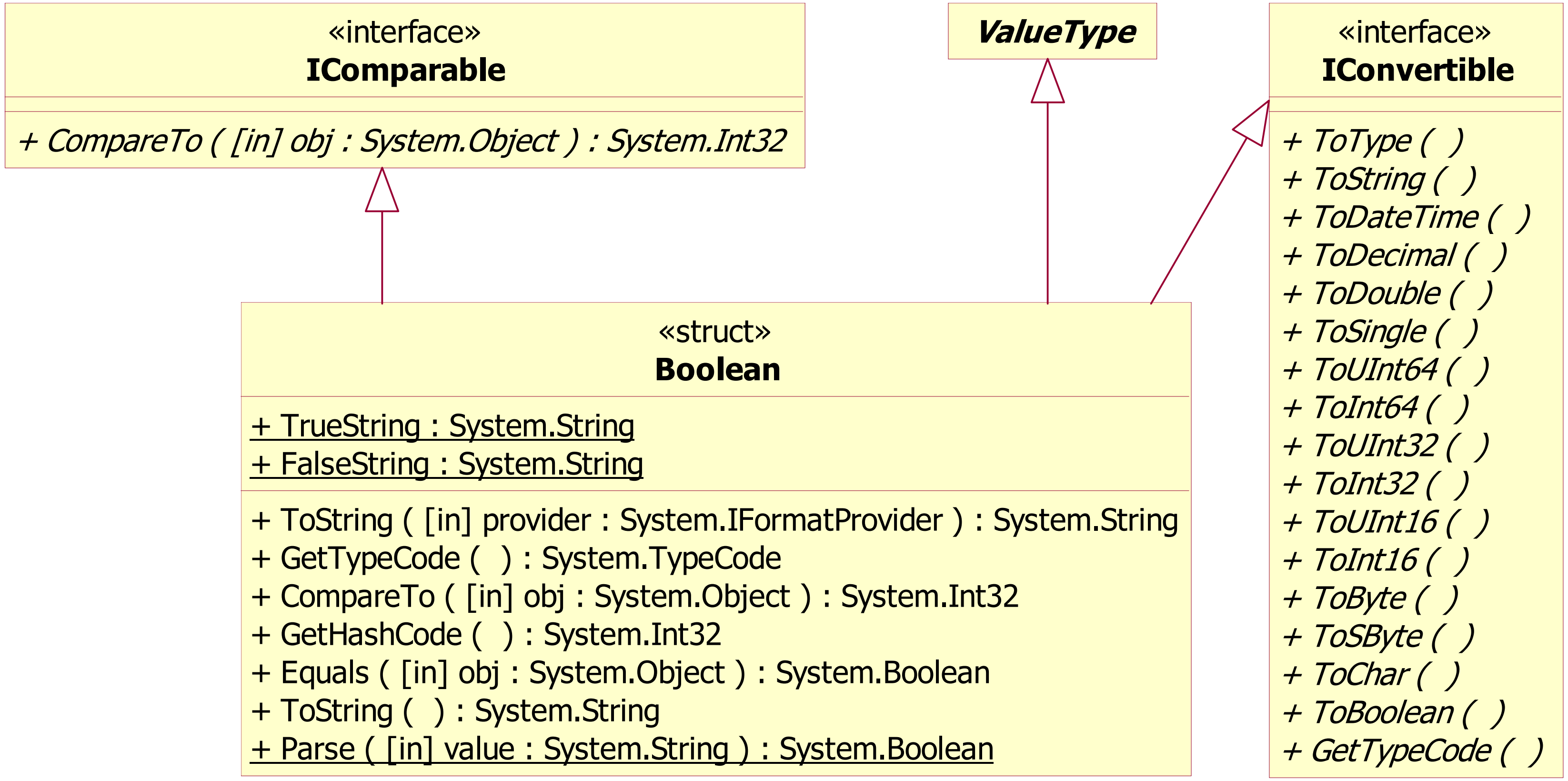


ValueType ( )
+ GetHashCode ( ) : System.Int32
+ Equals ([in] obj : System.Object) : System.Boolean
+ ToString ( ) : System.String

One or more fields of the derived type is used to calculate the return value. If one or more of those fields contains a mutable value, the return value might be unpredictable, and unsuitable for use as a key in a hash table. In that case, consider writing your own implementation of GetHash Code that more closely represents the concept of a hash code for the type.

The default implementation of the Equals method uses reflection to compare the corresponding fields of obj and this instance. Override the Equals method for a particular type to improve the performance of the method and more closely represent the concept of equality for the type.

System.Boolean



«interface»
IComparable

+ CompareTo ([in] obj : System.Object) : System.Int32

ValueType

«interface»
IConvertible

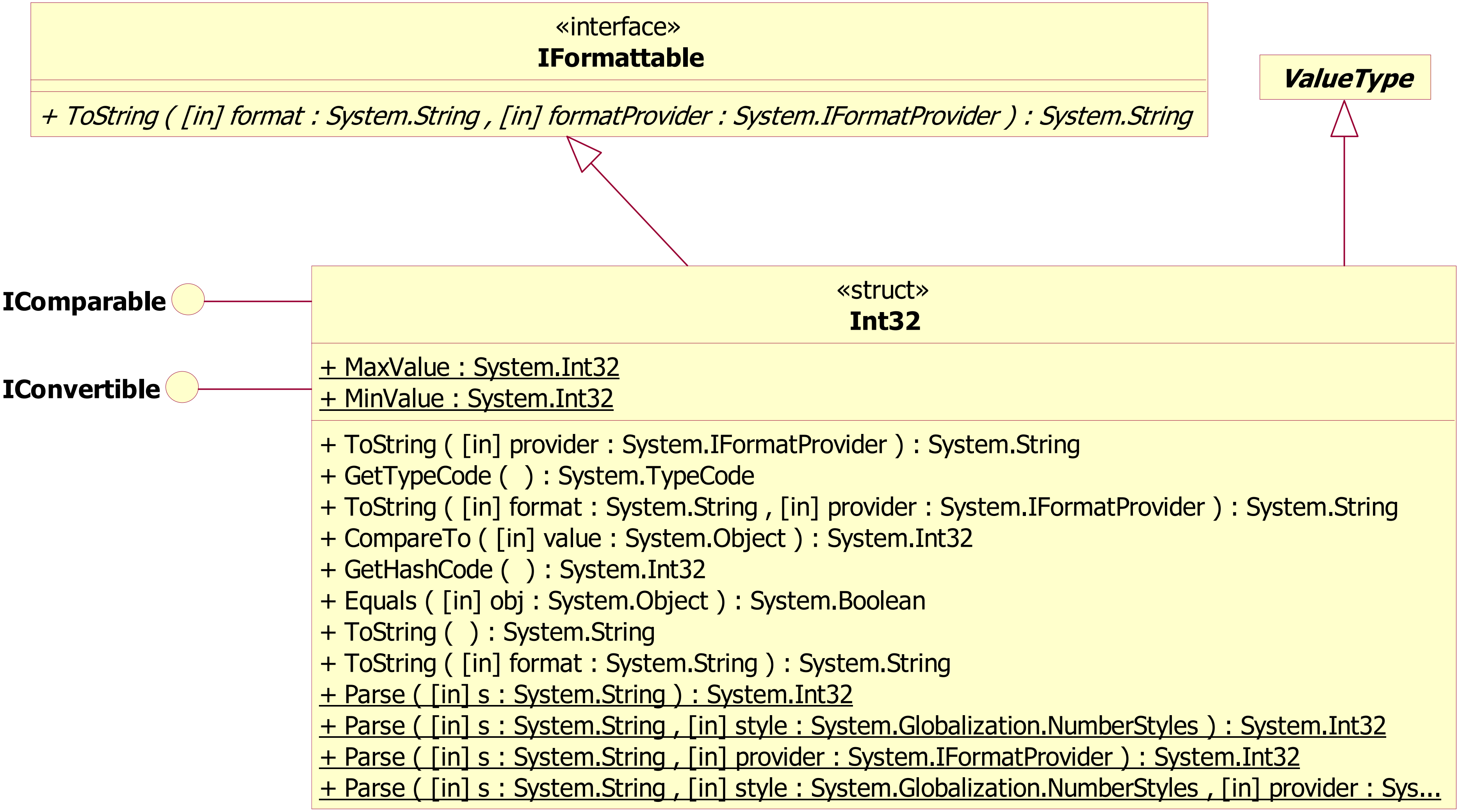
+ ToType ( )
+ ToString ( )
+ ToDateTime ( )
+ ToDecimal ( )
+ ToDouble ( )
+ ToSingle ( )
+ ToUInt64 ( )
+ ToInt64 ( )
+ ToUInt32 ( )
+ ToInt32 ( )
+ ToUInt16 ( )
+ ToInt16 ( )
+ ToByte ( )
+ ToSByte ( )
+ ToChar ( )
+ ToBoolean ( )
+ GetTypeCode ( )

«struct»
Boolean

+ TrueString : System.String
+ FalseString : System.String

+ ToString ([in] provider : System.IFormatProvider) : System.String
+ GetTypeCode ( ) : System.TypeCode
+ CompareTo ([in] obj : System.Object) : System.Int32
+ GetHashCode ( ) : System.Int32
+ Equals ([in] obj : System.Object) : System.Boolean
+ ToString ( ) : System.String
+ Parse ([in] value : System.String) : System.Boolean

System.Int32



«interface»
IFormattable

+ ToString ([in] format : System.String , [in] formatProvider : System.IFormatProvider) : System.String

ValueType

IComparable

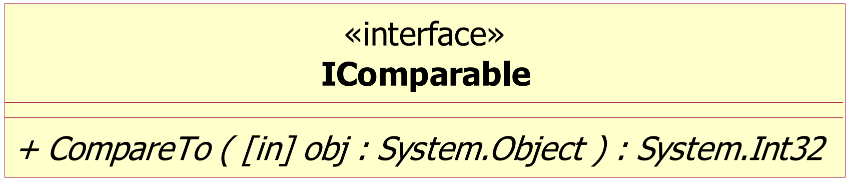
IConvertible

«struct»
Int32

+ MaxValue : System.Int32
+ MinValue : System.Int32

+ ToString ([in] provider : System.IFormatProvider) : System.String
+ GetTypeCode ( ) : System.TypeCode
+ ToString ([in] format : System.String , [in] provider : System.IFormatProvider) : System.String
+ CompareTo ([in] value : System.Object) : System.Int32
+ GetHashCode ( ) : System.Int32
+ Equals ([in] obj : System.Object) : System.Boolean
+ ToString ( ) : System.String
+ ToString ([in] format : System.String) : System.String
+ Parse ([in] s : System.String) : System.Int32
+ Parse ([in] s : System.String , [in] style : System.Globalization.NumberStyles) : System.Int32
+ Parse ([in] s : System.String , [in] provider : System.IFormatProvider) : System.Int32
+ Parse ([in] s : System.String , [in] style : System.Globalization.NumberStyles , [in] provider : Sys...

System.IComparable



«interface»
IComparable

+ CompareTo ([in] obj : System.Object) : System.Int32

- Compares the current instance with another object of the same type
- **Return Value**: a 32-bit signed integer that indicates the relative order of the comparands
- The return value has these meanings:
  - Less than zero - **`this`** instance is less than **`obj`**
  - Zero - **`this`** instance is equal to **`obj`**
  - Greater than zero - **`this`** instance is greater than **`obj`**
- By definition, any object compares greater than a null reference
- The parameter **`obj`** must be the same type as the class or value type that implements this interface; otherwise, an **`ArgumentException`** is thrown

`System.IComparable`

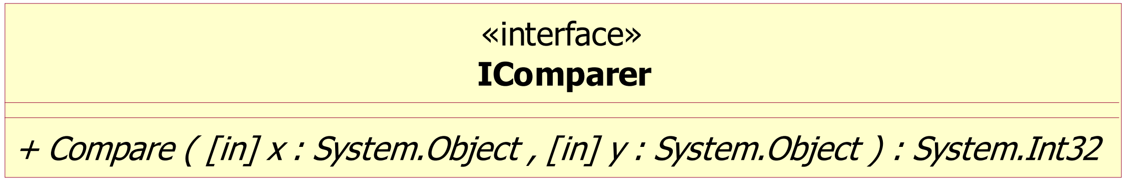
- Notes to Implementers:
  For any objects A, B and C, the following must be true:
  - **`A.CompareTo(A)`** is required to return zero
  - If **`A.CompareTo(B)`** returns zero then **`B.CompareTo(A)`** is required to return zero
  - If **`A.CompareTo(B)`** returns zero and **`B.CompareTo(C)`** returns zero then **`A.CompareTo(C)`** is required to return zero
  - If **`A.CompareTo(B)`** returns a value other than zero then **`B.CompareTo(A)`** is required to return a value of the opposite sign
  - If **`A.CompareTo(B)`** returns a value x not equal to zero, and **`B.CompareTo(C)`** returns a value y of the same sign as x, then **`A.CompareTo(C)`** is required to return a value of the same sign as x and y

Esempio 1

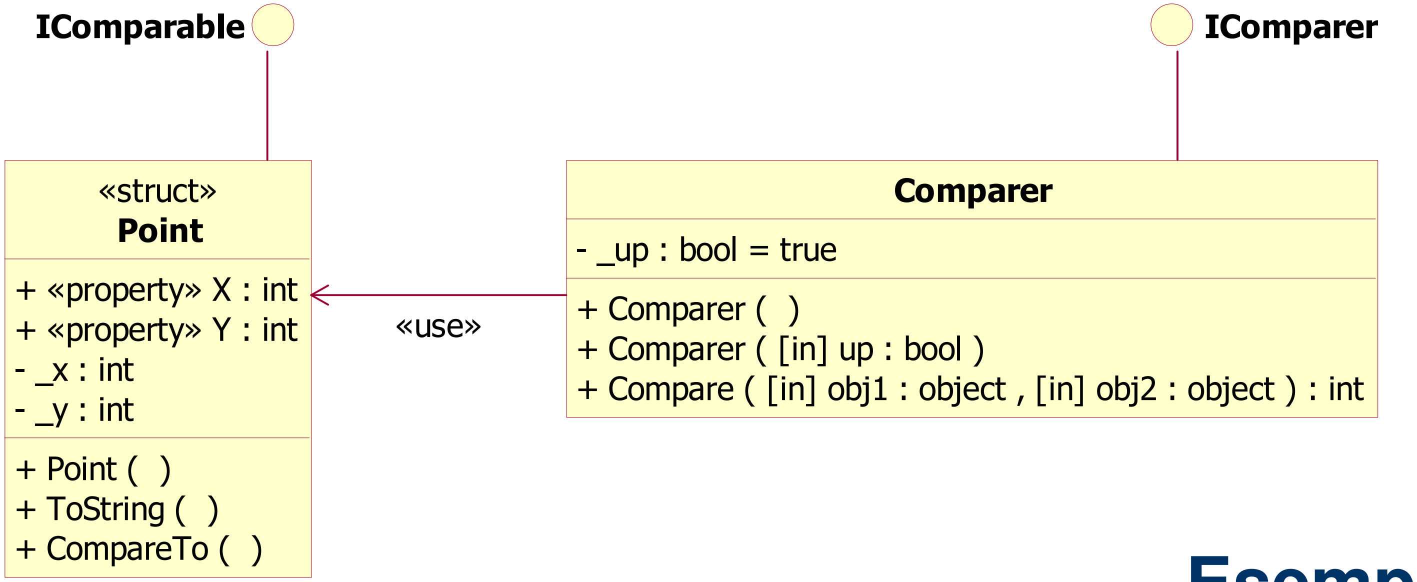
System.IComparable

- Se volessi:
  - Ordinare i punti in ordine decrescente
  - Ordinare dei film
    - Per genere, oppure
    - Per titolo
  - Ordinare degli studenti
    - Per cognome e nome, oppure
    - Per matricola, oppure
    - Per corso di studio
  - …

System.Collections.IComparer

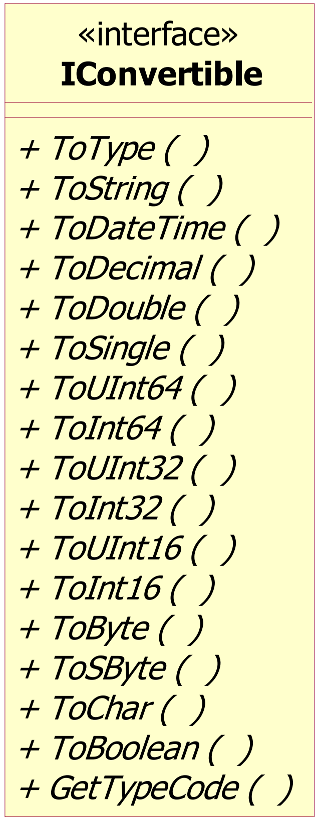

«interface»
IComparer
+ Compare ([in] x : System.Object , [in] y : System.Object) : System.Int32

- This interface is used in conjunction with the **`Array.Sort`** and **`Array.BinarySearch`** methods
- It provides a way to customize the sort order of a collection



Esempio 1

System.IConvertible

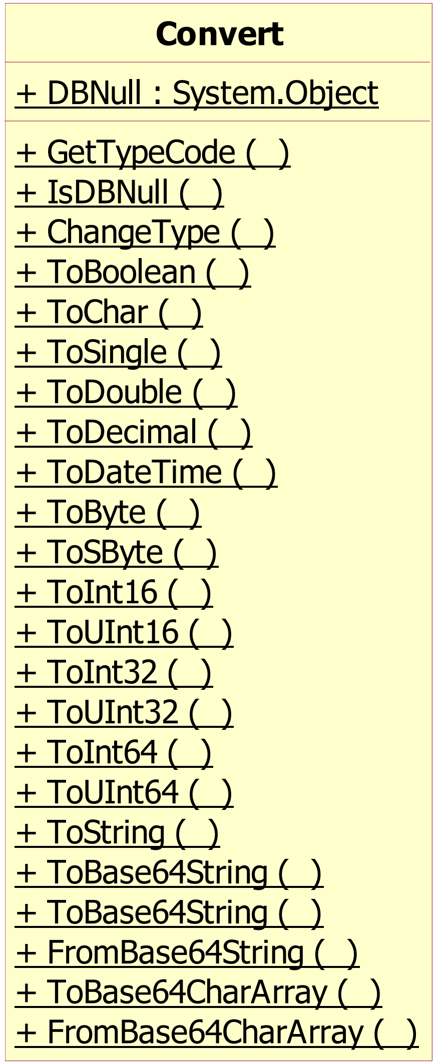


«interface»
IConvertible

- *+ ToType ( )*
- *+ ToString ( )*
- *+ ToDateTime ( )*
- *+ ToDecimal ( )*
- *+ ToDouble ( )*
- *+ ToSingle ( )*
- *+ ToUInt64 ( )*
- *+ ToInt64 ( )*
- *+ ToUInt32 ( )*
- *+ ToInt32 ( )*
- *+ ToUInt16 ( )*
- *+ ToInt16 ( )*
- *+ ToByte ( )*
- *+ ToSByte ( )*
- *+ ToChar ( )*
- *+ ToBoolean ( )*
- *+ GetTypeCode ( )*

- This interface provides methods to convert the value of an instance of an implementing type to a common language runtime type that has an equivalent value
- The **common language runtime types** are **`Boolean`**, **`SByte`**, **`Byte`**, **`Int16`**, **`UInt16`**, **`Int32`**, **`UInt32`**, **`Int64`**, **`UInt64`**, **`Single`**, **`Double`**, **`Decimal`**, **`DateTime`**, **`Char`**, and **`String`**
- If there is no meaningful conversion to a common language runtime type, then a particular interface method implementation throws **`InvalidCastException`** - For example, if this interface is implemented on a **`Boolean`** type, the implementation of the **`ToDateTime`** method throws an exception because there is no meaningful **`DateTime`** equivalent to a **`Boolean`** type

System.Convert



Convert

+ DBNull : System.Object

+ GetTypeCode ( )
+ IsDBNull ( )
+ ChangeType ( )
+ ToBoolean ( )
+ ToChar ( )
+ ToSingle ( )
+ ToDouble ( )
+ ToDecimal ( )
+ ToDateTime ( )
+ ToByte ( )
+ ToSByte ( )
+ ToInt16 ( )
+ ToUInt16 ( )
+ ToInt32 ( )
+ ToUInt32 ( )
+ ToInt64 ( )
+ ToUInt64 ( )
+ ToString ( )
+ ToBase64String ( )
+ ToBase64String ( )
+ FromBase64String ( )
+ ToBase64CharArray ( )
+ FromBase64CharArray ( )

- In `System.Int32`, l'implementazione dell'interfaccia `System.IConvertible` è un esempio di "*explicit interface implementation*":

```
int x = 32;
double d = x.ToDouble(…); // No!
```

È necessario scrivere:

```
((IConvertible) x).ToDouble(…)
```

- Se necessario, utilizzare la classe `Convert`:

```
Convert.ToDouble(x)
```

System.Convert

- Throws an exception if the conversion is not supported

```
bool b = Convert.ToBoolean(DateTime.Today);
  //  InvalidCastException
```

- Performs **checked conversions**

```
int k = 300;
byte b = (byte) k;  //  b == 44
byte b = Convert.ToByte(k);  //  OverflowException
```

- In alcuni casi, esegue un arrotondamento:

```
double d = 42.72;
int k = (int) d; // k == 42
int k = Convert.ToInt32(d); // k == 43
```

- Is also useful if you have a **string** that you want to convert to a numeric value:

```
string myString = "123456789";
int myInt = Convert.ToInt32(myString);
```

Conversione di tipo

- **Widening conversion** occurs when a value of one type is converted to another type that is of equal or greater size
  - Da `Int32` a `Int64`
  - Da `Int32` a `UInt64`
  - Da `Int32` a `Single` (con possibile perdita di precisione)
  - Da `Int32` a `Double`
- **Narrowing conversion** occurs when a value of one type is converted to a value of another type that is of a smaller size
  - Da `Int32` a `Byte`
  - Da `Int32` a `SByte`
  - Da `Int32` a `Int16`
  - Da `Int32` a `UInt16`
  - Da `Int32` a `UInt32`

Conversione di tipo

- **Conversioni implicite** – non generano eccezioni
  - **Conversioni numeriche**
    Il tipo di destinazione dovrebbe essere in grado di contenere, senza perdita di informazione, tutti i valori ammessi dal tipo di partenza
    Eccezione:

```
int k1 = 1234567891;
float b = k1;
int k2 = (int) b;  //  k2 == 1234567936
```

  - **Up cast**
    Principio di sostituibilità: deve sempre essere possibile utilizzare una classe derivata al posto della classe base

```
B b = new B(…);  //  class B : A
A a = b;
```

Conversione di tipo

- **Conversioni esplicite** – possono generare eccezioni
  - **Conversioni numeriche**
    Il tipo di destinazione non sempre è in grado di contenere il valore del tipo di partenza

```
int k1 = -1234567891;
uint k2 = (uint) k1;   //  k2 == 3060399405

int k1 = -1234567891;
uint k2 = checked((uint) k1);   //  OverflowException

int k1 = -1234567891;
uint k2 = Convert.ToUInt32(k1);   // OverflowException
```

Conversione di tipo

- **Conversioni esplicite** – possono generare eccezioni
  - **Down cast**

```
A a = new B(…);  //  class B : A
B b = (B) a;  //  Ok

a = new A(…);
b = (B) a;  //  InvalidCastException

if(a is B)  //  if(a.GetType() == typeof(B))
{
  b = (B) a;  //  Non genera eccezioni
  …
}

b = a as B;  //  b = (a is B) ? (B) a : null;
if(b != null)
{
  …
}
```

Conversione di tipo

- **Boxing – up cast** (conversione implicita)

```
int k1 = 100;
object o = k1;  //  Copia!
k1 = 200;
```

- **Unboxing – down cast** (conversione esplicita)

```
int k2 = (int) o;  //  k1 = 200, k2 = 100

double d1 = (double) k1;  //  Ok
d1 = k1;  //  Ok
d1 = o;  //  Non compila!
d1 = (double) o;  //  InvalidCastException
d1 = (int) o;  //  Ok
```

Conversione di tipo definite dall'utente

```
public static implicit operator typeOut(typeIn obj)

public static explicit operator typeOut(typeIn obj)
```

- Metodi statici di una classe o di una struttura
- La *keyword* **`implicit`** indica l'utilizzo automatico (cast implicito)
  Il metodo non deve generare eccezioni
- La *keyword* **`explicit`** indica la necessità di un cast esplicito
  Il metodo può generare eccezioni
- **`typeOut`** è il tipo del risultato del cast
- **`typeIn`** è il tipo del valore da convertire
- **`typeIn`** o **`typeOut`** deve essere il tipo che contiene il metodo

Esempio 1 - Digit

Conversioni a string

- Conversioni a **`string`** (di un **`Int32`**):
  - **`ToString()`**

    ```
    int k1 = -1234567891;
    string str = k1.ToString(); // str == "-1234567891"
    ```

  - **`ToString(string formatString)`**
    the instance is formatted with the **`NumberFormatInfo`** for the current culture

    ```
    k1.ToString("X"); // = "B669FD2D"
    k1.ToString("C"); // = "-€ 1.234.567.891,00"
    k1.ToString("C0"); // = "-€ 1.234.567.891"
    k1.ToString("N0"); // = "-1.234.567.891"
    k1.ToString("E"); // = "-1,234568E+009"
    ```

Conversioni a string

- Conversioni a `string` (di un `Int32`):
  - `String.Format(string format, params object[] args)`

    The `format` parameter is embedded with zero or more format items of the form, `{index[,alignment][:formatString]}`

```
int k1 = –1234567891;

String.Format(“{0}”,k1); // = “-1234567891”
String.Format(“{0:X}”,k1); // = “B669FD2D”
String.Format(“{0,5:X}”,k1); // = “B669FD2D”
String.Format(“{0,10:X}”,k1); // = “△△B669FD2D”
String.Format(“{0,-10:X}”,k1); // = “B669FD2D△△”
String.Format(“{0:N0}”,k1); // = “-1.234.567.891”
```

Conversioni da string

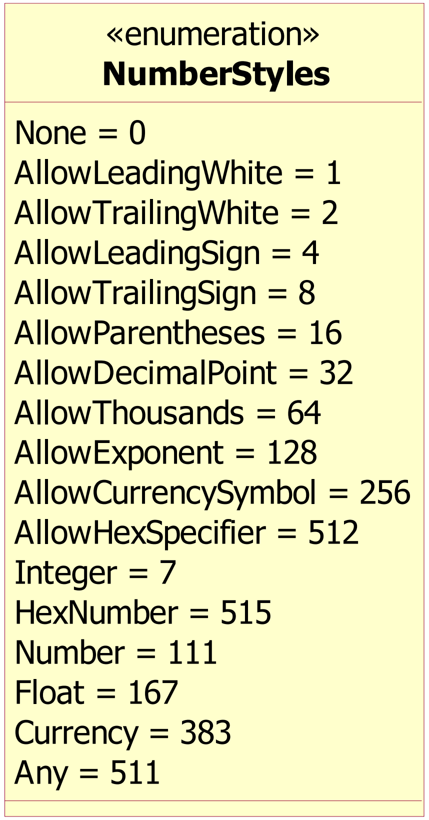
- Conversioni da **`string`** (in un **`Int32`**):
  - **`Int32.Parse(string str)`**

```
Int32.Parse("-1234567891"); // -1234567891
Int32.Parse("-1.234.567.891"); // FormatException
Int32.Parse(""); // FormatException
Int32.Parse("-1234567891999"); // OverflowException
Int32.Parse(null); // ArgumentNullException
```

  - **`Int32.Parse(string str, System.Globalization.NumberStyles style)`**

    `NumberStyles` determines the styles permitted in numeric string arguments that are passed to the **`Parse`** methods of the numeric base type classes

Conversioni da string



«enumeration»
NumberStyles

None = 0
AllowLeadingWhite = 1
AllowTrailingWhite = 2
AllowLeadingSign = 4
AllowTrailingSign = 8
AllowParentheses = 16
AllowDecimalPoint = 32
AllowThousands = 64
AllowExponent = 128
AllowCurrencySymbol = 256
AllowHexSpecifier = 512
Integer = 7
HexNumber = 515
Number = 111
Float = 167
Currency = 383
Any = 511

- The symbols to use for currency symbol, thousands separator, decimal point indicator, and leading sign are specified by **`NumberFormatInfo`**
- The attributes of **`NumberStyles`** are set by using the bitwise inclusive OR of the field flags

```
Int32.Parse(“-1.234.567.891”,
   System.Globalization.NumberStyles.Number); // ok
Int32.Parse(“B669FD2D”,
   System.Globalization.NumberStyles.HexNumber); // ok
```

Conversioni a/da string

- Conversioni a **`string`** (di un **`Int32`**):
  - **`Convert.ToString(int value, int toBase)`**
    `toBase` = 2, 8, 10, 16

```
int k1 = -1234567891;
Convert.ToString(k1); // "-1234567891"
Convert.ToString(k1,10); // "-1234567891"
Convert.ToString(k1,16); // "b669fd2d"
```

- Conversioni da **`string`** (in un **`Int32`**):
  - **`Convert.ToInt32(string str, int fromBase)`**
    `fromBase` = 2, 8, 10, 16

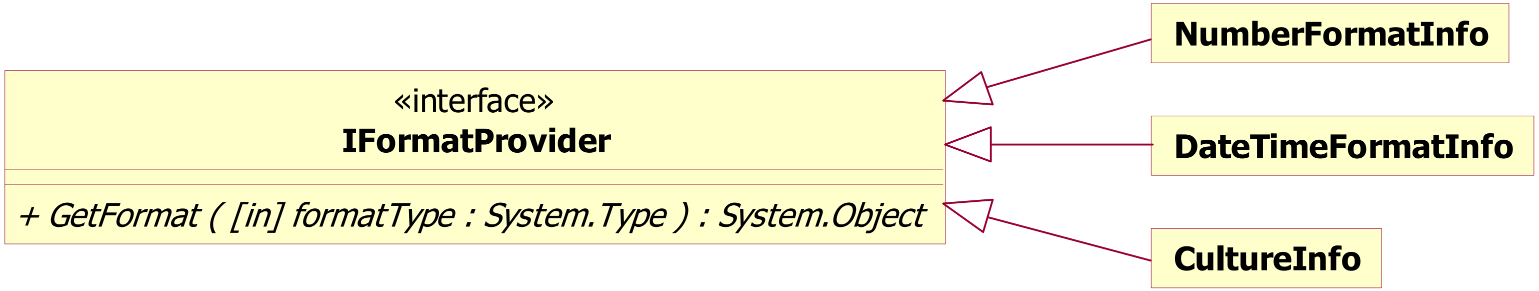
```
Convert.ToInt32("-1234567891"); // -1234567891
Convert.ToInt32("-1234567891",10); // -1234567891
Convert.ToInt32("B669FD2D",16); // -1234567891
Convert.ToInt32("0xB669FD2D",16); // -1234567891
Convert.ToInt32("B669FD2D",10); // FormatException
```

System.IFormattable

«interface»
IFormattable

+ ToString ([in] format : System.String , [in] formatProvider : System.IFormatProvider) : System.String

Provides functionality to format the value of an object into a string representation



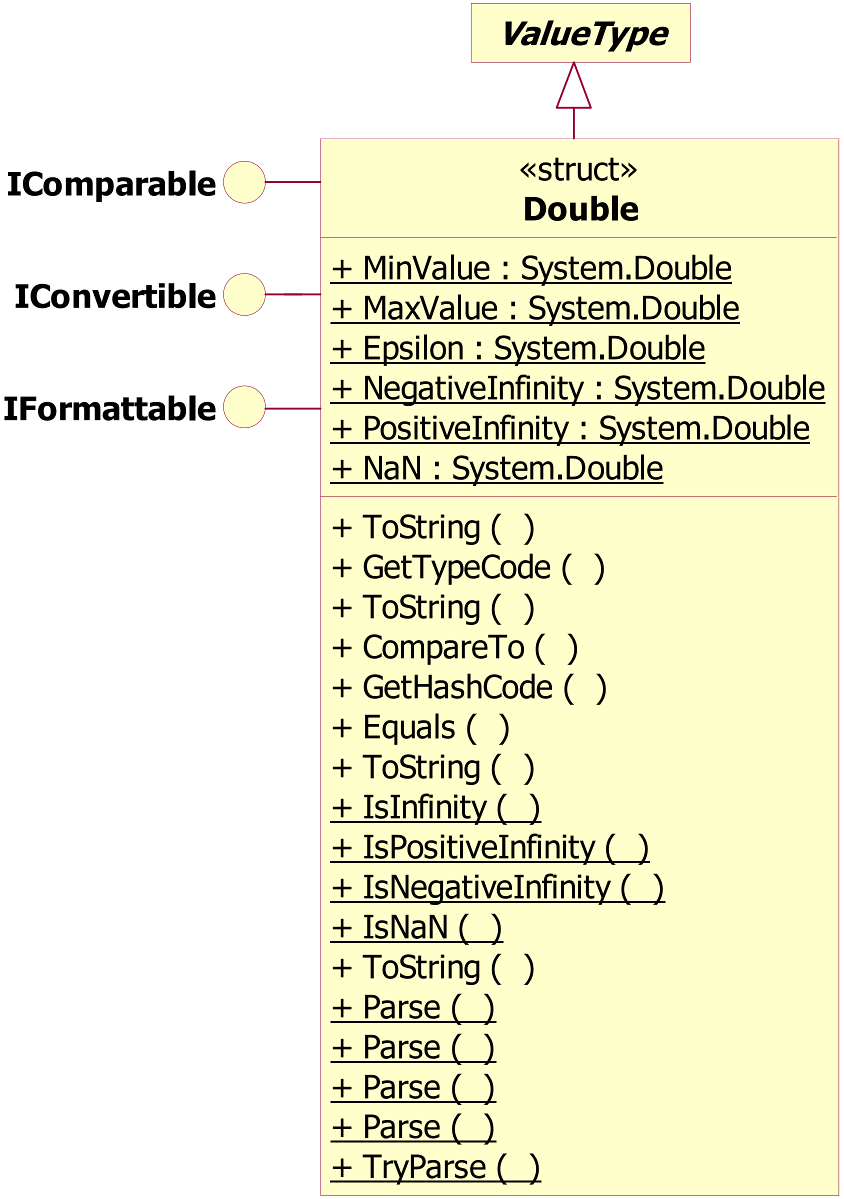
Provides a mechanism for retrieving an object to control formatting

«interface»
ICustomFormatter

+ Format ([in] format : System.String , [in] arg : System.Object , [in] formatProvider : System.IFormatProvider) : System.String

Defines a method that supports custom, user-defined formatting of the value of an object

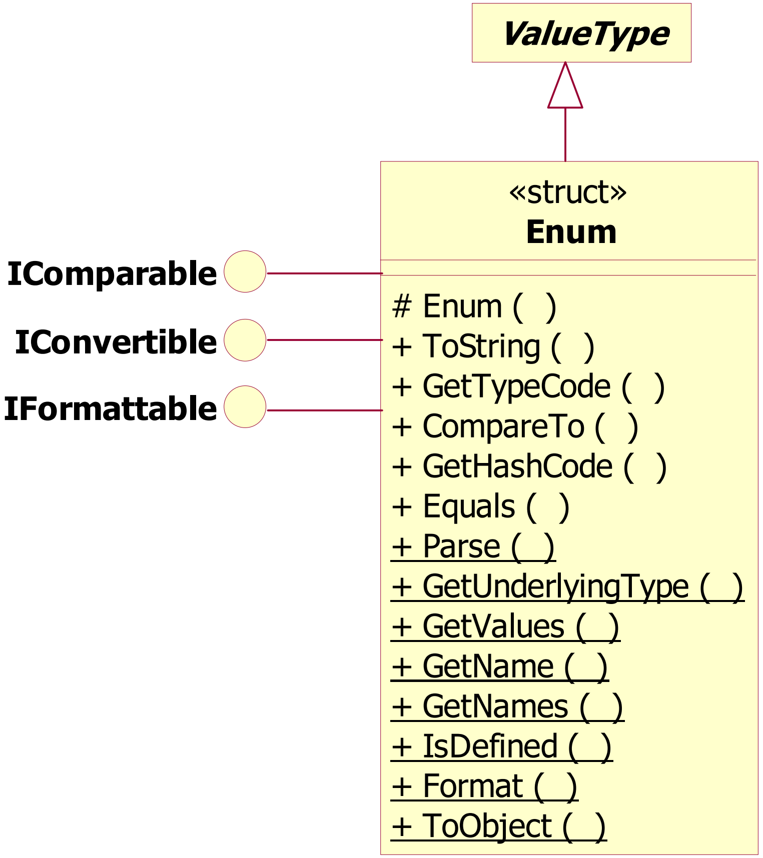
System.Double



- Follows IEEE 754 specification
- Supports ± 0, ± Infinity, NaN

- **`Epsilon`** represents the smallest positive **`Double`** > 0
- The **`TryParse`** method is like the **`Parse`** method, except this method does not throw an exception if the conversion fails
  - If the conversion succeeds, the return value is **`true`** and the result parameter is set to the outcome of the conversion
  - If the conversion fails, the return value is **`false`** and the result parameter is set to zero

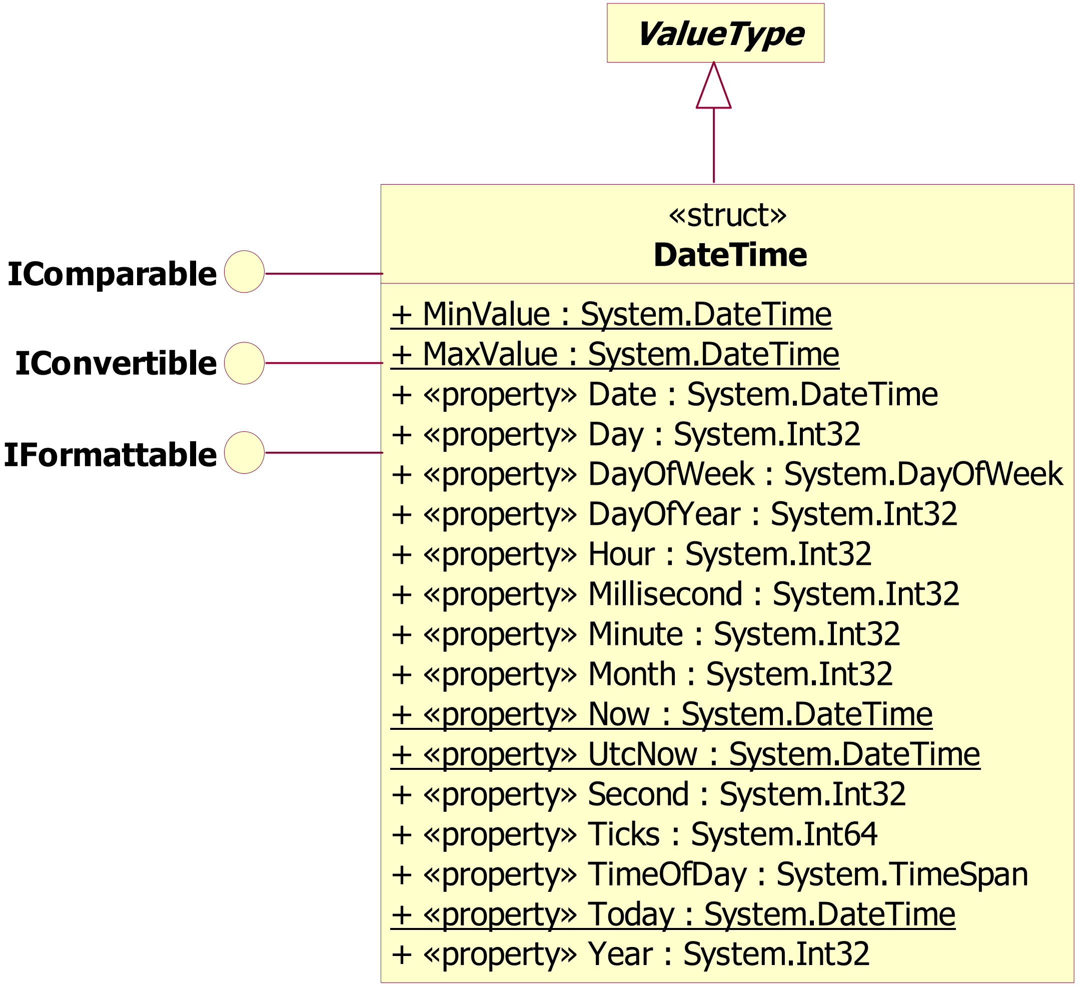
System.Enum



- **`Enum`** provides methods to
  - Compare instances of this class
  - Convert the value of an instance to its string representation
  - Convert the string representation of a number to an instance of this class
  - Create an instance of a specified enumeration and value
- You can also treat an **`Enum`** as a bit field

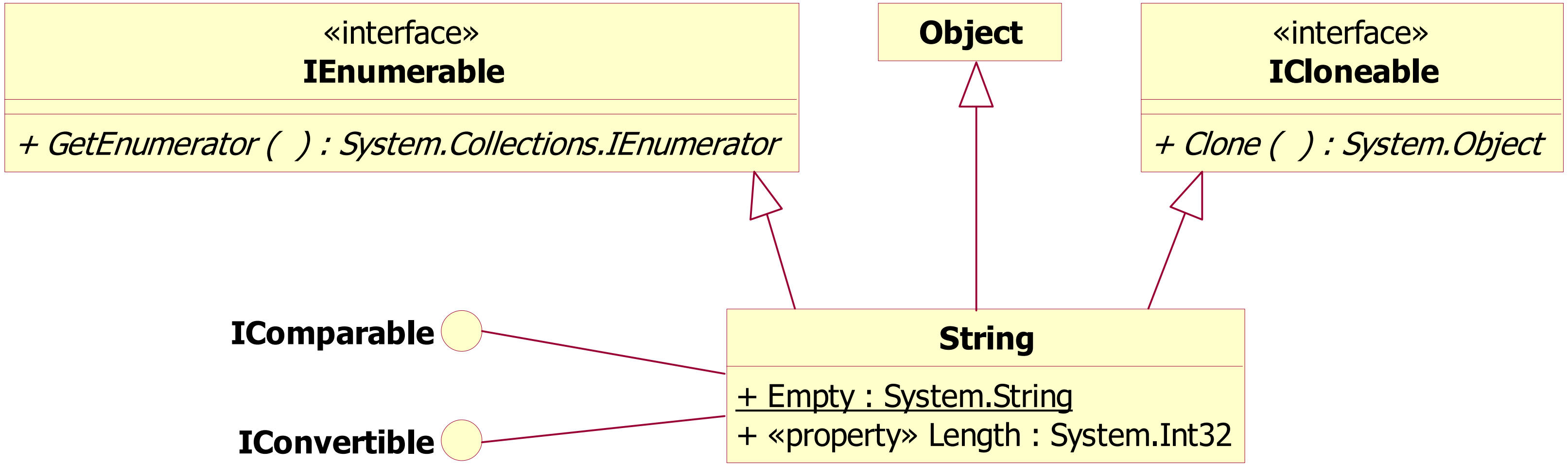
Esempio 1 - Color

System.DateTime


ValueType
«struct»
DateTime
+ MinValue : System.DateTime
+ MaxValue : System.DateTime
+ «property» Date : System.DateTime
+ «property» Day : System.Int32
+ «property» DayOfWeek : System.DayOfWeek
+ «property» DayOfYear : System.Int32
+ «property» Hour : System.Int32
+ «property» Millisecond : System.Int32
+ «property» Minute : System.Int32
+ «property» Month : System.Int32
+ «property» Now : System.DateTime
+ «property» UtcNow : System.DateTime
+ «property» Second : System.Int32
+ «property» Ticks : System.Int64
+ «property» TimeOfDay : System.TimeSpan
+ «property» Today : System.DateTime
+ «property» Year : System.Int32
IComparable
IConvertible
IFormattable

- Represents an instant in time, typically expressed as a date and time of day
- The **`DateTime`** value type represents dates and times with values ranging from 12:00:00 midnight, January 1, 0001 Anno Domini (Common Era) to 11:59:59 P.M., December 31, 9999 A.D. (C.E.)
- Time values are measured in 100ns units called ticks
- **`DateTime`** represents an instant in time, whereas a **`TimeSpan`** represents a time interval

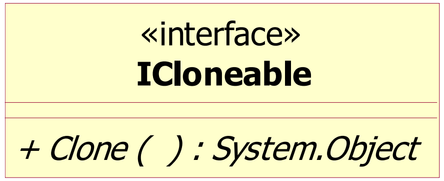
System.String


«interface»
IEnumerable
+ GetEnumerator ( ) : System.Collections.IEnumerator
Object
«interface»
ICloneable
+ Clone ( ) : System.Object
IComparable
IConvertible
String
+ Empty : System.String
+ «property» Length : System.Int32

- An immutable, fixed-length string of Unicode characters
- A **`String`** is called immutable because its value cannot be modified once it has been created
- Methods that appear to modify a **`String`** actually return a new **`String`** containing the modification
- If it is necessary to modify the actual contents of a string-like object, use the **`System.Text.StringBuilder`** class

Esempio 1

System.ICloneable



«interface»
ICloneable

+ Clone ( ) : System.Object

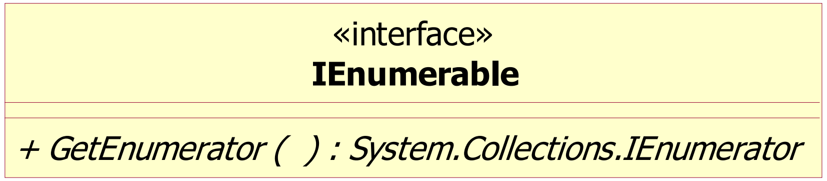
- Supports cloning, which creates a new instance of a class with the same value as an existing instance

- `Clone` creates a new object that is a copy of the current instance
- `Clone` can be implemented either as
  - a **shallow copy**, only the top-level objects are duplicated, no new instances of any fields are created

```
public object Clone()
{
  return MemberwiseClone();
}
```

  - a **deep copy**, all objects are duplicated
- `Clone` returns a new instance that is of the same type as (or occasionally a derived type of) the current object

System.Collections.IEnumerable

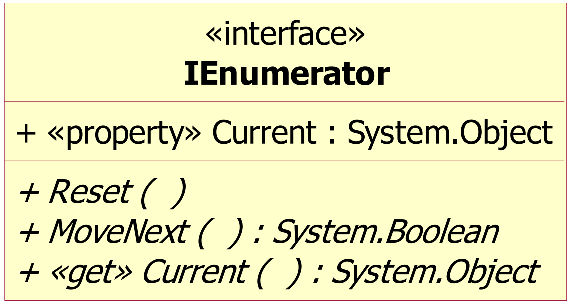


«interface»
IEnumerable

+ GetEnumerator ( ) : System.Collections.IEnumerator

- Exposes the enumerator, which supports a simple iteration over a collection

- **`GetEnumerator`** returns an enumerator that can be used to iterate through a collection



«interface»
IEnumerator

+ «property» Current : System.Object

+ Reset ( )
+ MoveNext ( ) : System.Boolean
+ «get» Current ( ) : System.Object

- Enumerators only allow **reading** the data in the collection
- Enumerators cannot be used to modify the underlying collection

- **`Reset`** returns the enumerator to its initial state
- **`MoveNext`** moves to the next item in the collection, returning
  - **`true`** if the operation was successful
  - **`false`** if the enumerator has moved past the last item
- **`Current`** returns the object to which the enumerator currently refers

`System.Collections.IEnumerator`

- Non deve essere implementata direttamente da una classe contenitore
- Deve essere implementata da una classe separata (eventualmente annidata nella classe contenitore) che fornisce la funzionalità di iterare sulla classe contenitore
- Tale suddivisione di responsabilità permette di utilizzare contemporaneamente più enumeratori sulla stessa classe contenitore
- La classe contenitore deve implementare l'interfaccia **`IEnumerable`**
- Se una classe contenitore viene modificata, tutti gli enumeratori ad essa associati vengono invalidati e non possono più essere utilizzati (**`InvalidOperationException`**)

System.Collections.IEnumerator

```
IEnumerator enumerator = enumerable.GetEnumerator();
while (enumerator.MoveNext())
{
  MyType obj = (MyType) enumerator.Current;
  …
}
```

```
foreach (MyType obj in enumerable)
{
  …
}
```

System.Collections.IEnumerator

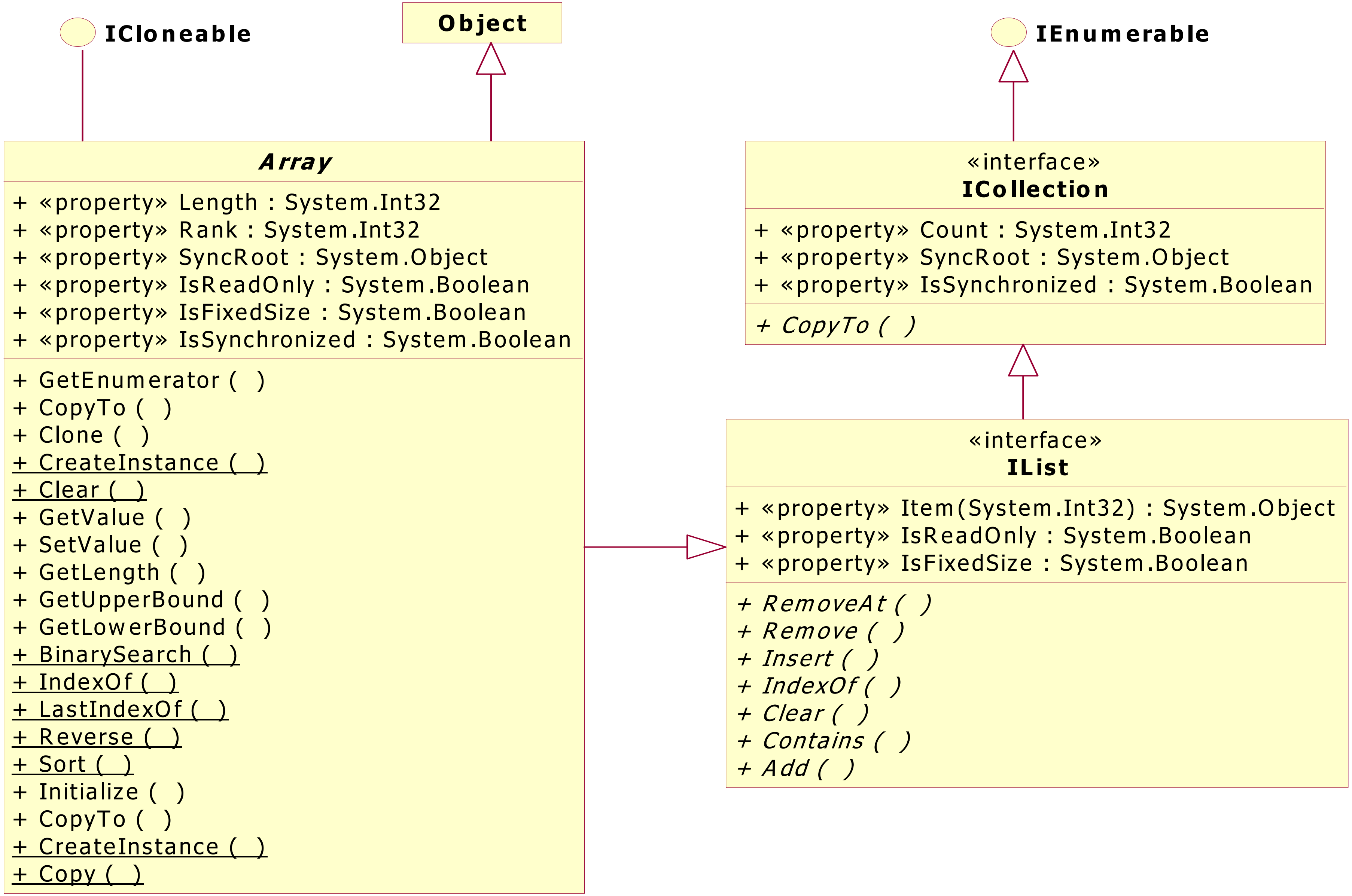
```
public class Contenitore : IEnumerable
{
  …
  public IEnumerator GetEnumerator()
  {
    return new Enumeratore(this);
  }

  class Enumeratore : IEnumerator
  {
    public Enumeratore(Contenitore contenitore) …

  }
}
```

Esempio 1 - Contenitore

System.Array



ICloneable
Object
IEnumerable
Array
+ «property» Length : System.Int32
+ «property» Rank : System.Int32
+ «property» SyncRoot : System.Object
+ «property» IsReadOnly : System.Boolean
+ «property» IsFixedSize : System.Boolean
+ «property» IsSynchronized : System.Boolean
+ GetEnumerator ( )
+ CopyTo ( )
+ Clone ( )
+ CreateInstance ( )
+ Clear ( )
+ GetValue ( )
+ SetValue ( )
+ GetLength ( )
+ GetUpperBound ( )
+ GetLowerBound ( )
+ BinarySearch ( )
+ IndexOf ( )
+ LastIndexOf ( )
+ Reverse ( )
+ Sort ( )
+ Initialize ( )
+ CopyTo ( )
+ CreateInstance ( )
+ Copy ( )
«interface»
ICollection
+ «property» Count : System.Int32
+ «property» SyncRoot : System.Object
+ «property» IsSynchronized : System.Boolean
+ CopyTo ( )
«interface»
IList
+ «property» Item(System.Int32) : System.Object
+ «property» IsReadOnly : System.Boolean
+ «property» IsFixedSize : System.Boolean
+ RemoveAt ( )
+ Remove ( )
+ Insert ( )
+ IndexOf ( )
+ Clear ( )
+ Contains ( )
+ Add ( )

System.Array

- One-dimensional arrays

```
int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
// array of references
SomeClass[] d = new SomeClass[10];
// array of values (directly in the array)
SomeStruct[] e = new SomeStruct[10];
```

System.Array

- Multidimensional arrays (jagged)

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

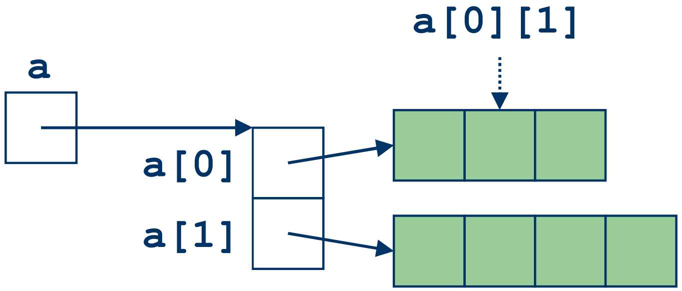
- Multidimensional arrays (rectangular)

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

System.Array

- **Jagged** (like in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];
…
int x = a[0][1];
```



- **Rectangular** (more compact and efficient)

```
int[,] a = new int[2, 3];
…
int x = a[0, 1];
```