Telerik Academy

Common Type System

.NET Types Hierarchy, Cloning, Comparing, Value and Reference Types, Parameters Passing

C# OOP

Telerik Software Academy
https://telerikacademy.com

Telerik Academy

Table of Contents

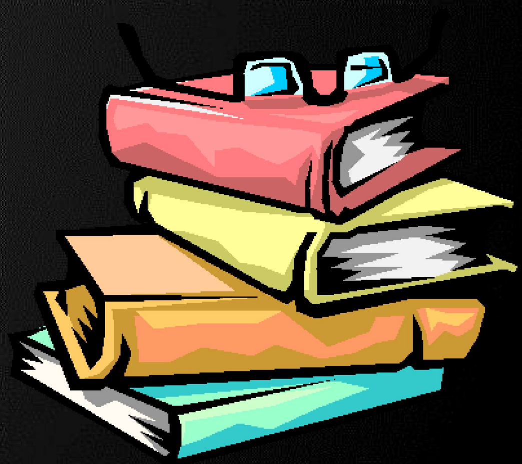
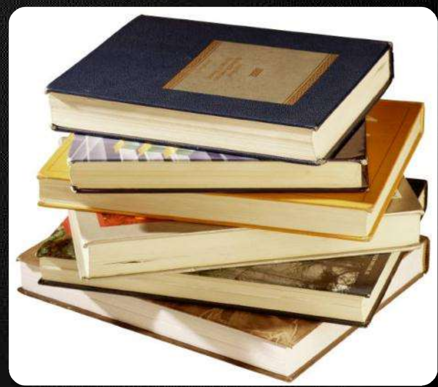


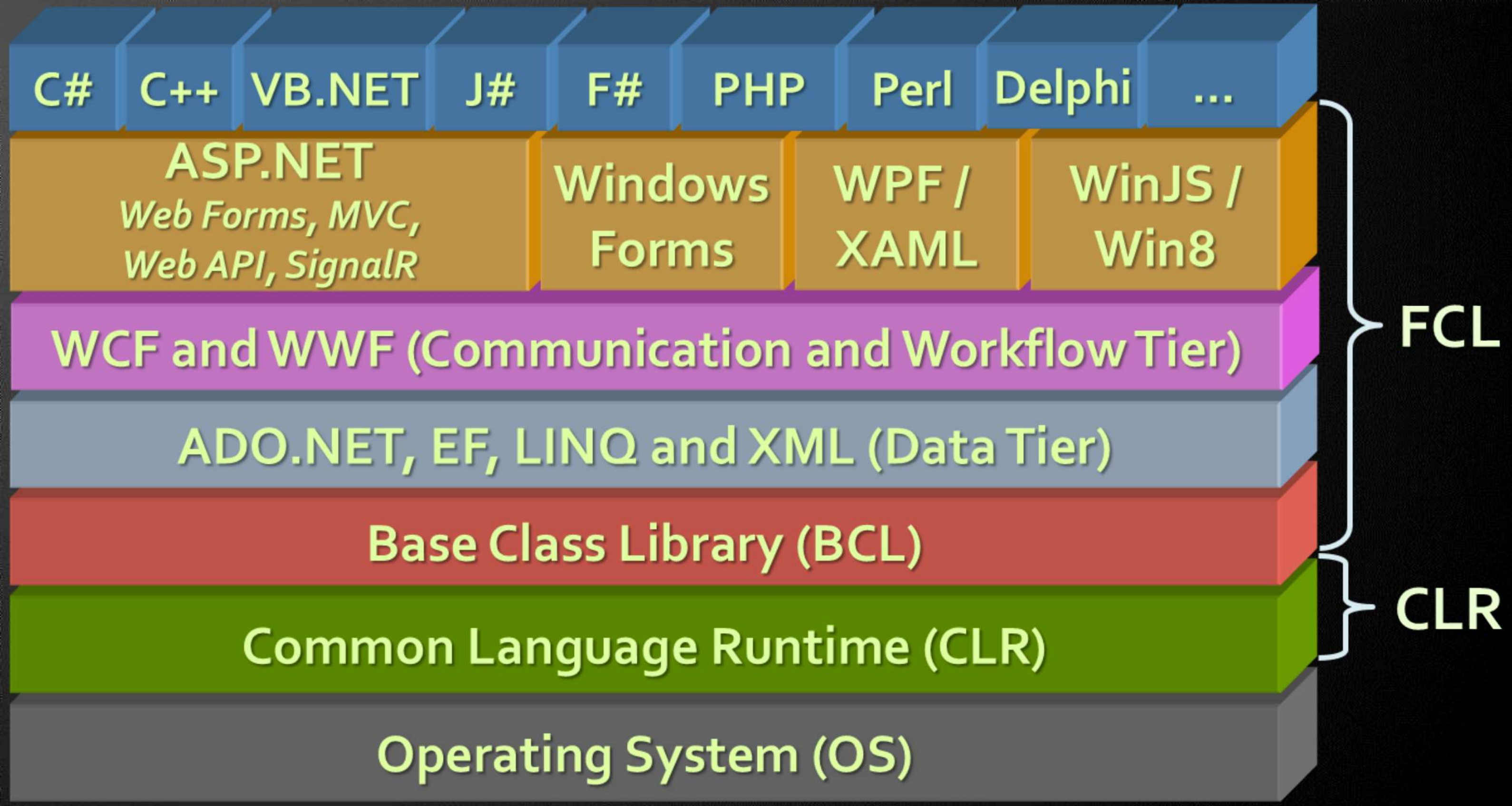
Table of Contents



What is Common Type System (CTS)?

Inside .NET Framework

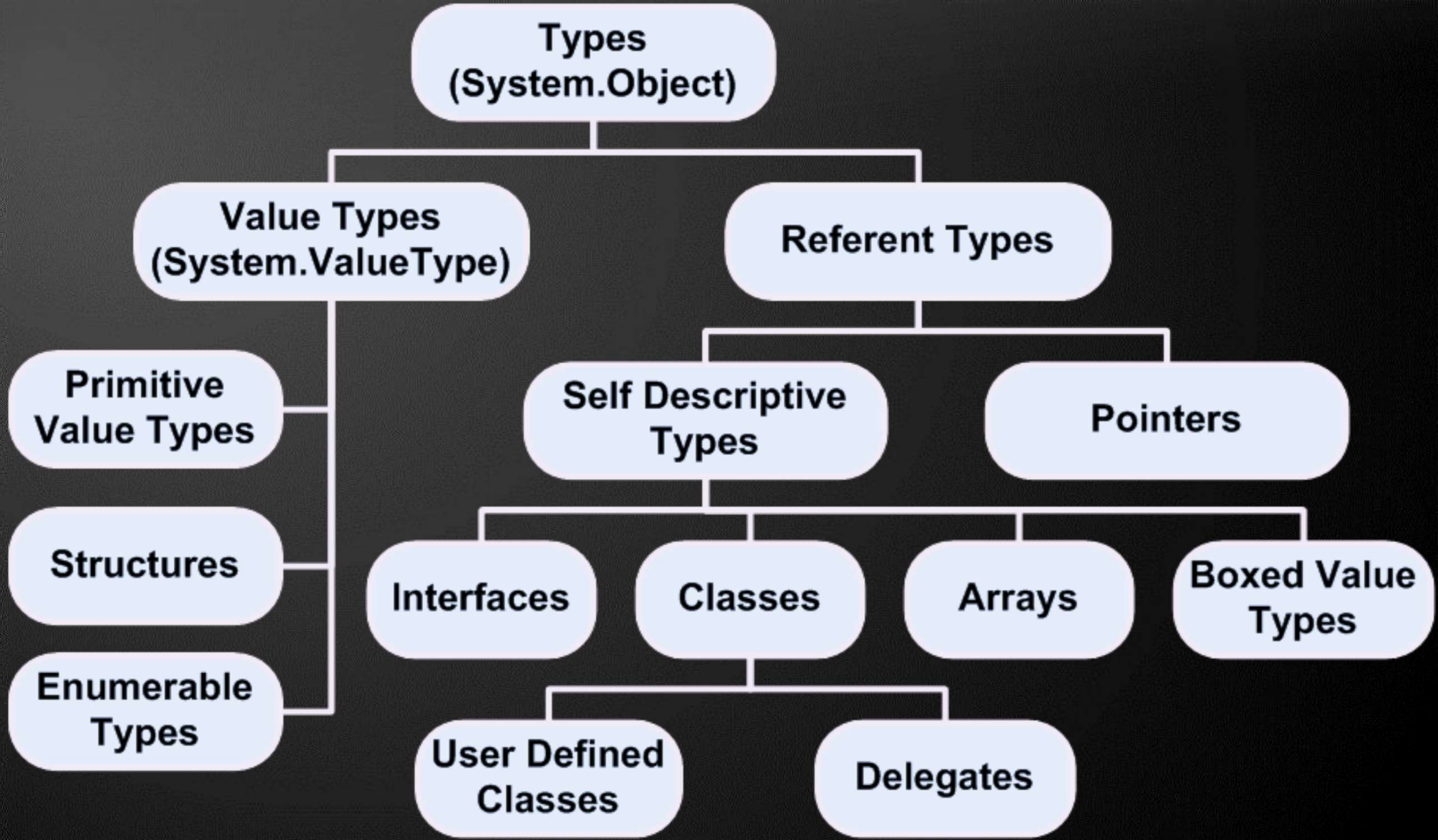
- Building blocks of .NET Framework



What is CTS?

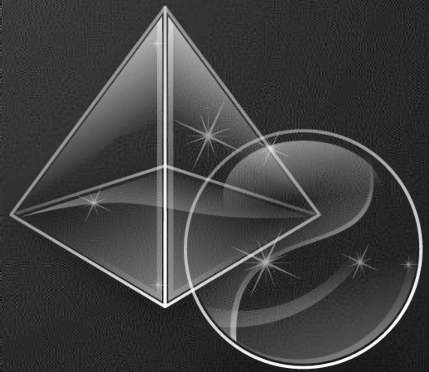
- .NET Common Type System (CTS)
- Defines CLR supported
  - Data types
  - Operations performed on them
- Extends the compatibility between different .NET languages
- Supports two types of data
  - Value types
  - Reference types
- All data types are inheritors of `System.Object`

.NET CTS Types Hierarchy



Telerik Academy

The System.Object Type

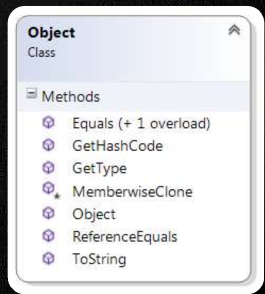


Follow us

Telerik Academy

System.Object Type

- Base class for each .NET type
  - Inherited by default when a new type is defined
- Important virtual methods:
  - `Equals()` – comparison with other object
  - `ToString()` – represents the object as a string
  - `GetHashCode()` – evaluates the hash code (used with hash-tables)
  - `Finalize()` – used for clean up purposes when an object is disposed



Telerik Academy

Overriding the Virtual Methods in System.Object

Overriding System.Object's Virtual Methods

- By default the operator `==` calls the `ReferenceEquals()` method
  - Compares the addresses for reference types
  - Or the binary representation for value types
- The methods `Equals()`, `GetHashCode()` should be defined at the same time
  - The same applies for the operators `==` and `!=`
  - You can override `Equals()` and use its implementation for `==` and `!=`

Overriding System.Object Methods

```
public class Student
{
  public string Name { get; set; }
  public int Age { get; set; }
  public override bool Equals(object param)
  {
    // If the cast is invalid, the result will be null
    Student student = param as Student;
    // Check if we have valid not null Student object
    if (student == null)
      return false;
    // Compare the reference type member fields
    if (! Object.Equals(this.Name, student.Name))
      return false;
    // Compare the value type member fields
    if (this.Age != student.Age)
      return false;
    return true;
  }
```

the example continues

Overriding System.Object Methods – *Example*

```
    public static bool operator == (Student student1,
                                    Student student2)
    {
        return Student.Equals(student1, student2);
    }
    public static bool operator !=(Student student1,
                                   Student student2)
    {
        return !(Student.Equals(student1, student2));
    }
    public override int GetHashCode()
    {
        return Name.GetHashCode() ^ Age.GetHashCode();
    }
}
```

Overriding the Virtual Methods in System.Object

Demo

More About System.Object

- The `System.Object` type has some other methods, which are inherited by all .NET types:
  - `GetType()`
    - Returns type's metadata as a `System.Type`
  - `MemberwiseClone()`
    - Copies the binary representation of the variable into a new variable (shallow clone)
  - `ReferenceEquals()`
    - Compares if two object have the same reference

Follow us

is and as operators

Telerik Academy

Type Operators in C#

- The `is` operator
  - Checks if an object an is instance of some type
  - Polymorphic operation
    - `5 is Int32`
    - `5 is object`
    - `5 is IComparable<int>`

- The `as` operator
  - Casts a reference type to another reference type
  - Returns `null` value if it fails
    - E.g. if the types are incompatible

Telerik Academy

Operators is and as – *Example*

```
class Base { }
class Derived : Base { }
class TestOperatorsIsAndAs
{
   static void Main()
   {
      Object objBase = new Base();
      if (objBase is Base)
         Console.WriteLine("objBase is Base");
      // Result: objBase is Base
      if (! (objBase is Derived))
         Console.WriteLine("objBase is not Derived");
      // Result : objBase is not Derived
      if (objBase is System.Object)
         Console.WriteLine("objBase is System.Object");
      // Result : objBase is System.Object
```

the example continues

Operators is and as – *Example*

```
    Base b = objBase as Base;
      Console.WriteLine("b = {0}", b);
      // Result: b = Base
      Derived d = objBase as Derived;
      if (d == null)
          Console.WriteLine("d is null");
      // Result: d is null
      Object o = objBase as Object;
      Console.WriteLine("o = {0}", o);
      // Result: o = Base
      Derived der = new Derived();
      Base bas = der as Base;
      Console.WriteLine("bas = {0}", bas);
      // Result: bas = Derived
    }
}
```

Telerik Academy

Operators is and as

[Demo](#)

Telerik Academy

Object Cloning

Follow us

Telerik Academy

Object Cloning

- In programming `cloning` an object means to create an identical copy of certain object
- `Shallow cloning`(shallow copy)
  - Uses the protected `MemberwiseClone()` method
  - Copies the value types bit by bit (binary)
  - Copies only the addresses of the reference types
- `Deepcloning`(deep copy)
  - Recursively copies all member data
  - Implemented manually by the programmer

Object Cloning

- Types which allow cloning implement the `ICloneable` interface
- The `Clone()` method of the `ICloneable`
  - The only method of the interface
  - Returns an identical copy of the object
    - Returns `object` → must be casted later
  - You decide whether to create a deep or shallow copy or something between

Follow us

Object Cloning – *Example*

```
class LinkedList: ICloneable
{
  public T Value { get; set; }
  public LinkedList NextNode { get; private set; }

  public LinkedList(T value,
    LinkedList nextNode = null)
  {
    this.Value = value;
    this.NextNode = nextNode;
  }

  object ICloneable.Clone()  // Implicit implementation
  {
    return this.Clone();
  }
}
```

the example continues

Object Cloning – *Example*

```
public LinkedList<T> Clone() // our method Clone()
{
  // Copy the first element
  LinkedList<T> original = this;
  T valueOriginal = original.Value;
  LinkedList<T> result = new LinkedList<T>(valueOriginal);
  LinkedList<T> copy = result;
  original = original.NextNode;
  // Copy the rest of the elements
  while (original != null)
  {
    valueOriginal = original.Value;
    copy.NextNode = new LinkedList<T>(valueOriginal);
    original = original.NextNode;
    copy = copy.NextNode;
  }
  return result;
}
```

Deep and Shallow Object Cloning

Demo

Telerik Academy

The Interface IComparable<T>

Follow us

Telerik Academy

IComparable<T> Interface

- The `System.IComparable<T>` interface
  - Implemented by the types, which can be compared (ordered in increasing order)
- The `CompareTo(T)` method defines the comparison. It returns:
  - Number `< 0` – if the passed object is bigger than the `this` instance
  - Number `= 0` – if the passed object is equal to the `this` instance
  - Number `> 0` – if the passed object is smaller than the `this` instance

Telerik Academy

IComparable – *Example*

```
class Point : IComparable
{
  public int X { get; set; }
  public int Y { get; set; }
  public int CompareTo(Point otherPoint)
  {
    if (this.X != otherPoint.X)
    {
      return (this.X - otherPoint.X);
    }
    if (this.Y != otherPoint.Y)
    {
      return (this.Y - otherPoint);
    }
    return 0;
  }
}
```

Implementing IComparable<T>

Demo

Telerik Academy

The IEnumerable<T> Interface

Telerik Academy

IEnumerable<T>

- The `IEnumerable<T>` interface provides collection classes with `foreach` traversal
  - It consists of 4 interfaces: `IEnumerable`, `IEnumerable`, `IEnumerator<T>`, `IEnumerator`

```
public interface IEnumerable<T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}

// Non-generic version (compatible with .NET 1.1)
public interface IEnumerable : IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Telerik Academy

IEnumerator<T>

- The `IEnumerator<T>` interface provides sequential read-only, forward-only iterator

```
public interface IEnumerator<T> : IEnumerator
{
  bool MoveNext();
  void Reset();
  T Current { get; }
}

public interface IEnumerator
{
  bool MoveNext();
  void Reset();
  object Current { get; }
}
```

Telerik Academy

Yield Return in C#

- The `yield return` construct in C# simplifies the `IEnumerator<T>` implementations
  - When a yield return statement is reached
    - The expression is returned, and the current location in code is retained (for later use)

```
public IEnumerator<int> GetEnumerator()
{
  for (int i=100; i<200; i++)
  {
    yield return i;
  }
}
```

Follow us

Implementing IEnumerable

Demo

Follow us

Value Types

Telerik Academy

Value Types

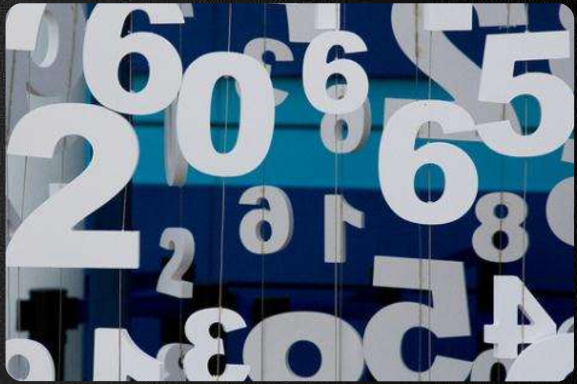
- Store their values in the stack
- Can not hold `null` value
- Destroyed when the given variable goes out of scope
- When a method is called they are:
  - Passed by value
  - Stored in the stack (copied)

Follow us

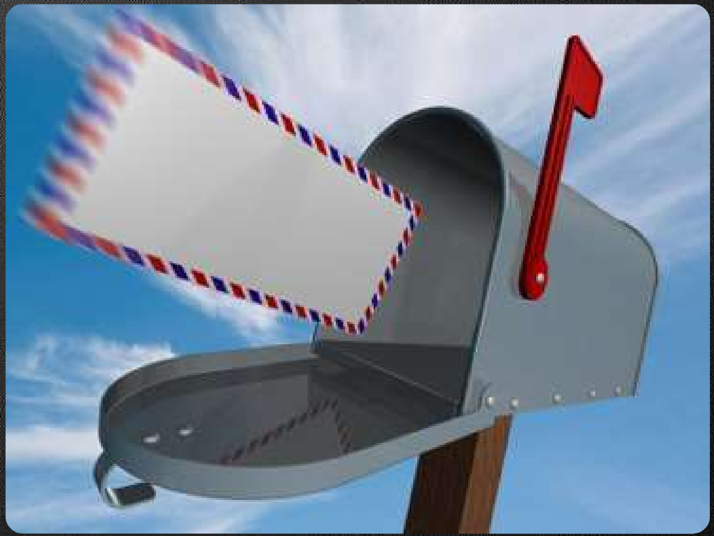
Telerik Academy

Value Types

- Inherit `System.ValueType`
- Value types are:
  - Primitive types
    - `int`, `char`
    - `float`, `bool`
    - Others
  - Structures
  - Enumerations (enumerable types)



Reference Types



Telerik Academy

Reference Types

- Implemented as type-safe pointers to objects
- Stored in the dynamic memory
- When a method is called they are passed by reference (by their address)
- Automatically destroyed by the CLR Garbage Collector, when they are out of scope or they are not in use
- Can hold `null` value

Telerik Academy

Reference Types

- It is possible for many variables to point to one and the same reference type object
- Referent objects are:
  - `System.Object`, `System.String`
  - Classes and interfaces
  - Arrays
  - Delegates
  - Pointers

Value vs. Reference Types

Assigning, Memory Location and Values

Follow us

Assigning Values

- Value Types
  - When assigning value types, their value is copied to the variable
- Reference Types
  - When assigning referent type, only the reference (address) is copied and the objects stays the same

Memory Location

- The memory location for value types is the program execution stack
- The memory location for reference types is the dynamic memory
  - Also called managed heap

Values

- Value types can not take `null` as a value, because they are not pointers
- Value types inherit `System.ValueType`
- Reference types inherit `System.Object`
- Value type variables can be stored in reference types with the boxing technique

Telerik Academy

Value and Reference Types - *Example*

```
class RefClass { public int value; }  // Reference type
struct ValStruct { public int value; }  // Value type
class TestValueAndReferenceTypes
{
    static void Main()
    {
        RefClass ref_Example_ = new RefClass();
        ref_Example_.value = 100;
        RefClass ref_Example_2 = ref_Example_;
        ref_Example_2.value = 200;
        Console.WriteLine(ref_Example_.value); // Prints 200

        ValStruct val_Example_ = new ValStruct();
        val_Example_.value = 100;
        ValStruct val_Example_2 = val_Example_;
        val_Example_2.value = 200;
        Console.WriteLine(val_Example_.value); // Prints 100
    }
}
```

Telerik Academy

Types, Variables and Memory

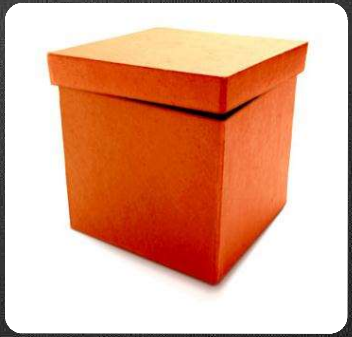
Stack for program execution		
Variable	Address	Value
(stack beginning)	...	...
class1	0x0012F680	0x04A41A44
class2	0x0012F67C	0x04A41A44
struct1	0x0012F678	100
struct2	0x0012F674	200
(free stack memory)	...	...
(end of stack)	0x00000000	...

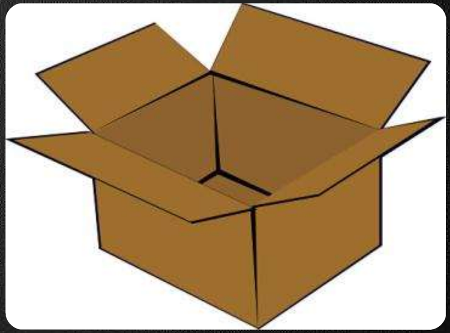
Dynamic memory (managed heap)	
Address	Value
...	...
...	...
0x04A41A44	200
...	...
...	...
...	...
...	...

Value and Reference Types

Demo

Boxing and Unboxing





Boxing and Unboxing

- Value types can be stored in reference types
- If needed CLR boxes and unboxes value types
- Boxing is operation, that converts a value type to a reference one

```
int i = 123;
// The following line boxes i.
object o = i;
```

- Unboxing is the opposite operation
  - Converts boxed value to ordinary value type

```
o = 123;
i = (int)o;  // unboxing
```

Boxing

1. Allocates dynamic memory for the creation of the object
2. Copies the contents of the variable from the stack to the allocated dynamic memory
3. Returns a reference to the created object in the dynamic memory
4. The original type is memorized
5. The dynamic memory contains information, that the object reference holds boxed object

Unboxing

1. If the reference is `null` a `NullReferenceException` is thrown
2. If the reference does not point to a valid boxed value an `InvalidCastException` is thrown
3. The value is pulled from the heap and is stored into the stack

Passing Parameters

ref and out Keywords

Follow us

Passing Parameters

- Parameters can be passed in several ways to the methods:
  - `in` (default)
    - Passing value for value types
    - Passing heap address for reference types
  - `out`
    - Passed by stack address for both value types and reference types
    - The initialization can be done by the called method

Passing Parameters

- Parameters can be passed in several ways to the methods:
  - `ref`
    - Passed by stack address for both value types and reference types
    - Initialization can't be done by the called method – access is for read and write

Telerik Academy

ref Parameters – *Example*

```
public class Student
{
  public string name;
  static void IncorrectModifyStudent(Student student)
  {
     student = new Student("Changed: " + student.name);
  }
  static void ModifyStudent(ref Student student)
  {
     student = new Student("Changed: " + student.name);
  }
  static void Main()
  {
     Student s = new Student("Ivaylo");
     Console.WriteLine(s.name); // Ivaylo
     IncorrectModifyStudent(s);
     Console.WriteLine(s.name); // Ivaylo
     ModifyStudent(ref s);
     Console.WriteLine(s.name); // Changed: Ivaylo
  }
}
```

`ref` Parameters

Demo

Follow us

Telerik Academy

out Parameters – *Example*

```
class TestOutParameters
{
  static void Main()
  {
    Rectangle rect = new Rectangle(5, 10, 12, 8);
    Point location;
    Dimensions dimensions;

    // Location and dimension are not pre-initialized!
    rect.GetLocationAndDimensions(
        out location, out dimensions);

    Console.WriteLine("({0}, {1}, {2}, {3})",
        location.x, location.y,
        dimensions.width, dimensions.height);
    // Result: (5, 10, 12, 8)
  }
}
```

out Parameters

Demo

Follow us