

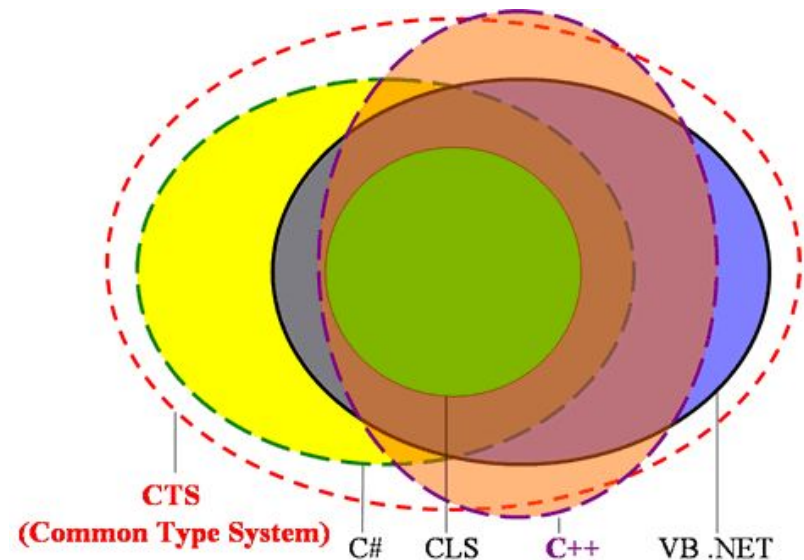
Common Type System. Value and reference types in C#.

AGENDA

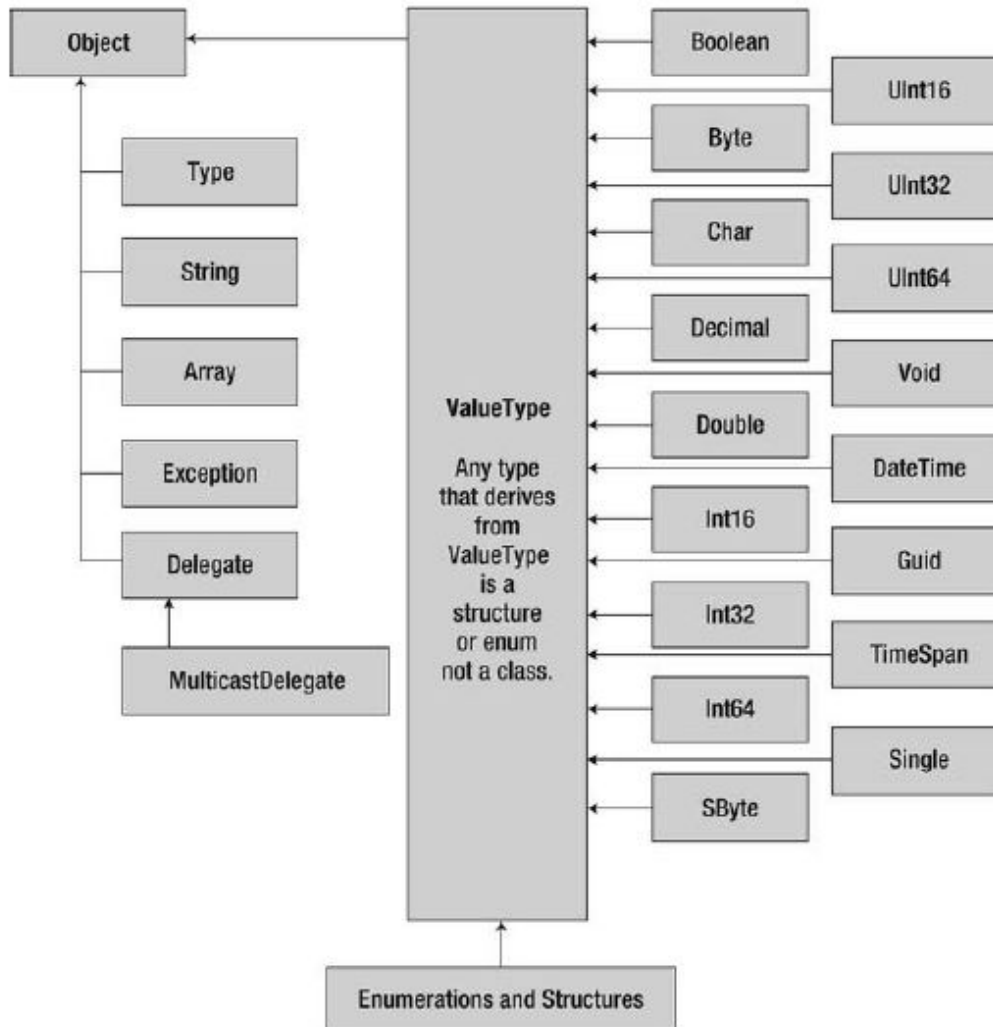
- Common Type System
- Data Type Class Hierarchy
- class Object
- Intrinsic Data Types
- Variable Declaration and Initialization
- Implicitly Typed Local Variables
- C# Nullable Types. ?? Operator
- Value and Reference Types
- Shallow and deep copy

Common Type System

- CTS – common type system:
 - defines how types are declared, used, and managed in the common language runtime,
 - is an important part of the runtime's support for cross-language integration.
- CTS performs the following functions:
 - ✓ Establishes a framework that helps enable cross-language integration, type safety, and high-performance code execution.
 - ✓ Provides an object-oriented model that supports the complete implementation of many programming languages.
 - ✓ Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.
 - ✓ Provides a library that contains the primitive data types (Boolean, Byte, Char, Int32, and UInt64)



The Data Type Class Hierarchy



A globally unique identifier (GUID) is a statistically unique 128-bit number

Base class Object

```
public class Object
{
    // Virtual members.
    public virtual bool Equals(object obj);
    protected virtual void Finalize();
    public virtual int GetHashCode();
    public virtual string ToString();

    // Instance-level, nonvirtual members.
    public Type GetType();
    protected object MemberwiseClone();

    // Static members.
    public static bool Equals(object objA, object objB);
    public static bool ReferenceEquals(object objA, obje
}
}
```

Equals() is used to compare object references, not the state of the object. **ValueType** class overrides it for the value-based comparisons.

GetHashCode() returns an int that identifies a specific object instance.

ToString() returns a string representation of this object
- *fully qualified name*

MemberwiseClone() creates a shallow copy by creating a new object, and then copying the nonstatic fields of the current object to the new object. If a field is a reference type, the reference is copied but the referred object is not;

Intrinsic Data Types

The Intrinsic Data Types of C#

C# Shorthand	CLS Compliant?	System Type	Range	Meaning in Life
bool	Yes	System.Boolean	true or false	Represents truth or falsity
sbyte	No	System.SByte	-128 to 127	Signed 8-bit number
byte	Yes	System.Byte	0 to 255	Unsigned 8-bit number
short	Yes	System.Int16	-32,768 to 32,767	Signed 16-bit number
ushort	No	System.UInt16	0 to 65,535	Unsigned 16-bit number
int	Yes	System.Int32	-2,147,483,648 to 2,147,483,647	Signed 32-bit number
uint	No	System.UInt32	0 to 4,294,967,295	Unsigned 32-bit number
long	Yes	System.Int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit number
ulong	No	System.UInt64	0 to 18,446,744,073,709,551,615	Unsigned 64-bit number
char	Yes	System.Char	U+0000 to U+ffff	Single 16-bit Unicode character
float	Yes	System.Single	-3.4 10^{38} to +3.4 10^{38}	32-bit floating-point number
double	Yes	System.Double	$\pm 5.0 \cdot 10^{-324}$ to $\pm 1.7 \cdot 10^{308}$	64-bit floating-point number
decimal	Yes	System.Decimal	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / (10^{0 \text{ to } 28})$	128-bit signed number
string	Yes	System.String	Limited by system memory	Represents a set of Unicode characters
Object	Yes	System.Object	Can store any data type in an object variable	The base class of all types in the .NET universe

Variable Declaration and Initialization

- It is a *compiler error* to make use of a local variable before assigning an initial value.
- All intrinsic data types support a *default constructor*. We can create a variable using the `new` keyword, which automatically sets the variable to its default value:
 - **bool** variables are set to `false`.
 - **Numeric data** is set to 0 (or 0.0 in the case of floating-point data types).
 - **char** variables are set to a single empty character.
 - **BigInteger** variables are set to 0. (from `System.Numerics.dll`)
 - **DateTime** variables are set to 1/1/0001 12:00:00 AM.
 - **Object references** (including **strings**) are set to `null`.

Variable Declaration and Initialization

```
static void LocalVarDeclarations()
{
    Console.WriteLine("=> Data Declarations:");
    // Local variables are declared and initialized as follows:
    // dataType varName = initialValue;
    int myInt = 0;

    // You can also declare and assign on two lines.
    string myString;
    myString = "This is my character data";

    Console.WriteLine();
}
```

- It is more cumbersome to use the **new** keyword when creating a basic data type variable:

```
static void NewingDataTypes()
{
    Console.WriteLine("=> Using new to create variables:");
    bool b = new bool();           // Set to false.
    int i = new int();             // Set to 0.
    double d = new double();      // Set to 0.
    DateTime dt = new DateTime(); // Set to 1/1/0001 12:00:00 AM
    Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
    Console.WriteLine();
}
```


Implicitly Typed Local Variables

```
static void DeclareExplicitVars()
{
    // Explicitly typed local variables
    // are declared as follows:
    // dataType variableName = initialValue;
    int myInt = 0;
    bool myBool = true;
    string myString = "Time, marches on...";
}
```

```
static void DeclareImplicitVars()
{
    // Implicitly typed local variables
    // are declared as follows:
    // var variableName = initialValue;
    var myInt = 0;
    var myBool = true;
    var myString = "Time, marches on...";
}
```

```
// Print out the underlying type.
Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

```
class ThisWillNeverCompile
{
    // Error! var cannot be used as field data!
    private var myInt = 10;

    // Error! var cannot be used as a return value
    // or parameter type!
    public var MyMethod(var x, var y){}
}
```

```
// Error! Can't assign null as initial value!
var myObj = null;

// OK, if SportsCar is a reference type!
var myCar = new SportsCar();
myCar = null;
```

C# Nullable Types

```
static void Main(string[] args)
{
    // Compiler error
    // Value types
    bool myBool = null;
    int myInt = null;

    // OK! Strings are reference types.
    string myString = null;
}
```

In C#, the ? suffix notation is a shorthand for creating an instance of the generic **System.Nullable<T>** structure type.

```
static void LocalNullableVariablesUsingNullable()
{
    // Define some local nullable types using Nullable<T>.
    Nullable<int> nullableInt = 10;
    Nullable<double> nullableDouble = 3.14;
    Nullable<bool> nullableBool = null;
    Nullable<char> nullableChar = 'a';
    Nullable<int>[] arrayOfNullableInts = new int?[10];
}
}
```

C# Nullable Types and operator ??

```
class DatabaseReader
{
    // Nullable data field.
    public int? numericValue = null;
    public bool? boolValue = true;

    // Note the nullable return type.
    public int? GetIntFromDatabase()
    { return numericValue; }

    // Note the nullable return type.
}

```

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Nullable Data *****\n");
    DatabaseReader dr = new DatabaseReader();

    // Get int from "database".
    int? i = dr.GetIntFromDatabase();
    if (i.HasValue)
        Console.WriteLine("Value of 'i' is: {0}", i.Value);
    else

```

?? Operator allows you to assign a value to a nullable type if the retrieved value is in fact null.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Nullable Data *****\n");
    DatabaseReader dr = new DatabaseReader();
    ...
    // If the value from GetIntFromDatabase() is null,
    // assign local variable to 100.
    int myData = dr.GetIntFromDatabase() ?? 100;
    Console.WriteLine("Value of myData: {0}", myData);
    Console.ReadLine();
}

```

```
'b' is: {0}", b.Value);
'b' is undefined.");

```

Value and Reference Types

Intriguing Question	Value Type	Reference Type
Where are objects allocated?	Allocated on the stack.	Allocated on the managed heap.
How is a variable represented?	Value type variables are local copies.	Reference type variables are pointing to the memory occupied by the allocated instance.
What is the base type?	Implicitly extends <code>System.ValueType</code> .	Can derive from any other type (except <code>System.ValueType</code>), as long as that type is not “sealed” (more details on this in Chapter 6).
Can this type function as a base to other types?	No. Value types are always sealed and cannot be inherited from.	Yes. If the type is not sealed, it may function as a base to other types.
What is the default parameter passing behavior?	Variables are passed by value (i.e., a copy of the variable is passed into the called function).	For value types, the object is copied-by-value. For reference types, the reference is copied-by-value.

Value and Reference Types

Intriguing Question	Value Type	Reference Type
Can this type override <code>System.Object.Finalize()</code> ?	No. Value types are never placed onto the heap and, therefore, do not need to be finalized.	Yes, indirectly (more details on this in Chapter 13).
Can I define constructors for this type?	Yes, but the default constructor is reserved (i.e., your custom constructors must all have arguments).	But, of course!
When do variables of this type die?	When they fall out of the defining scope.	When the object is garbage collected.

Value and References Types, Assignment Operator

```
class ShapeInfo
{
    public string infoString;

    public ShapeInfo(string info)
    {
        infoString = info;
    }
}
```

```
struct Rectangle
{
    // The Rectangle structure contains a reference type member.
    public ShapeInfo rectInfo;

    public int rectTop, rectLeft, rectBottom, rectRight;

    public Rectangle(string info, int top, int left, int bottom, int right)
    {
        rectInfo = new ShapeInfo(info);
        rectTop = top; rectBottom = bottom;
        rectLeft = left; rectRight = right;
    }
}
```

```
static void ValueTypeContainingRefType()
{
    // Create the first Rectangle.
    Console.WriteLine("-> Creating r1");
    Rectangle r1 = new Rectangle("First Rect", 10, 10, 50, 50);

    // Now assign a new Rectangle to r1.
    Console.WriteLine("-> Assigning r2 to r1");
    Rectangle r2 = r1;

    // Change some values of r2.
    Console.WriteLine("-> Changing values of r2");
    r2.rectInfo.infoString = "This is new info!";
    r2.rectBottom = 4444;

    // Print values of both rectangles.
    r1.Display();
    r2.Display();
}
```

```
ring = {0}, Top = {1}, Bottom = {2}, " +
= {4}";
rectTop, rectBottom, rectLeft, rectRight);
```

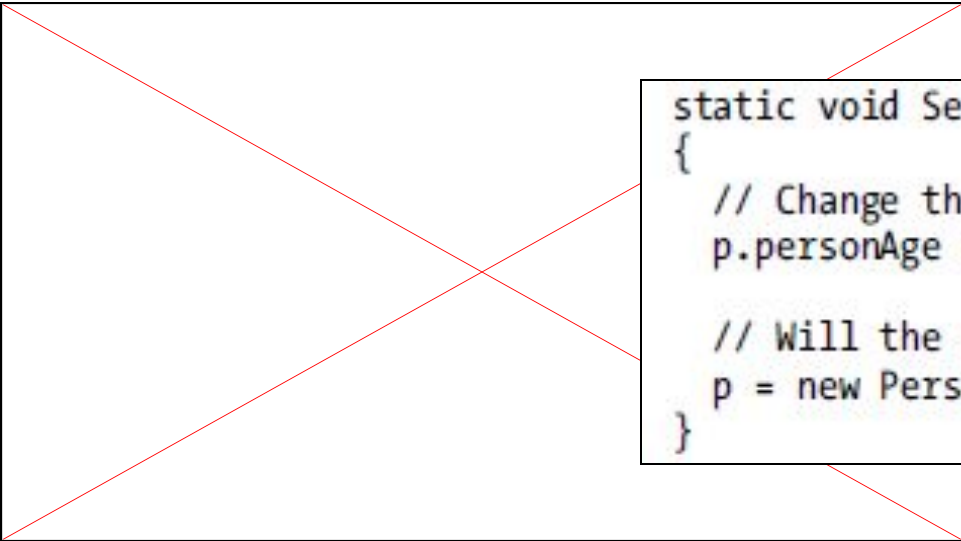
```
-> Creating r1
-> Assigning r2 to r1
-> Changing values of r2
String = This is new info!, Top = 10, Bottom = 50, Left = 10, Right = 50
String = This is new info!, Top = 10, Bottom = 4444, Left = 10, Right = 50
```

Parameter Modifiers

C# Parameter Modifiers

Parameter Modifier	Meaning in Life
(None)	If a parameter is not marked with a parameter modifier, it is assumed to be passed by value, meaning the called method receives a copy of the original data.
out	Output parameters must be assigned by the method being called, and therefore, are passed by reference. If the called method fails to assign output parameters, you are issued a compiler error.
ref	The value is initially assigned by the caller and may be optionally reassigned by the called method (as the data is also passed by reference). No compiler error is generated if the called method fails to assign a ref parameter.
params	This parameter modifier allows you to send in a variable number of arguments as a single logical parameter. A method can have only a single params modifier, and it must be the final parameter of the method. In reality, you might not need to use the params modifier all too often; however, be aware that numerous methods within the base class libraries do make use of this C# language feature.

Passing Reference Types by Value and by Reference



```
static void SendAPersonByValue(Person p)
{
    // Change the age of "p"?
    p.personAge = 99;

    // Will the caller see this reassignment?
    p = new Person("Nikki", 99);
}
```

```
static void SendAPersonByReference(ref Person p)
{
    // Change some data of "p".
    p.personAge = 555;

    // "p" is now pointing to a new object on the heap!
    p = new Person("Nikki", 999);
}
```


Shallow and deep copy

- If you have a class or structure that contains **only value types**, implement your **Clone()** method using **MemberwiseClone()**:
- If you have a custom type that maintains other reference types, you might want to create a new object that takes into reference type order to get a “

```
public class Point : ICloneable
{
    private int x, y;

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

```
public class Rectangle: ICloneable{
    public object Clone()
    {
        // First get a shallow copy.
        Rectangle newRect =
            (Rectangle)this.MemberwiseClone();
        // Then fill in the gaps.
        newRect.P1 = (Point)this.P1.Clone();
        //...
        return newRect;
    }
}
```

References

- [MSDN:Common Type System](#)
- [Built-in Data Types](#)
- [Value and Reference Types](#)
- [Object class](#)

Questions ?
