VALUE-TYPES VS REFERENCE-TYPES

Value Types (Structures, Enumerations)	• Mainly meant for storing simple values. • Instances (examples) are called as "structure instances" or "enumeration instances". • Instances are stored in "Stack". Every time a method is called, a new stack will be created.
Reference Types (string, Classes, Interfaces, Delegates)	• Mainly meant for storing complex/large amounts of values.•Instances (examples) are called "Objects" (Class Instances / Interface Instances / Delegate Instances). • Instances (examples) are called "Objects" (Class Instances / Interface Instances / Delegate Instances). • Instances (objects) are stored in a "heap". Heap is only one for the entire application.

STRUCTURES

What?

- Structure is a "type", similar to "class", which can contain fields, methods, parameterized constructors, properties and events.

Syntax:

```
struct StructureName
{
  fields
  methods
  parameterized constructors
  properties
  events
}
```

Example:

```
struct Student
{
  public int studentId;
  public string studentName;

  public string GetStudentName( )
  {
    return studentName;
  }
}
```

STRUCTURES

Defination:

- The instance of structure is called a "structure instance" or "structure variable"; but not called an 'object'. We can't create objects for structure. Objects can be created only based on 'class'.
- Structure instances are stored in 'stack'.
- Structure doesn't support 'user-defined parameter-less constructor and also destructor.
- Structure can't be inherited from other classes or structures.
- The structure can implement one or more interfaces.
- The structure doesn't support virtual and abstract methods.
- Structures are mainly meant for storing small amounts of data (one or very few values).
- Structures are faster than classes, as their instances are stored in 'stack'.

STRUCTURES

Class (vs) Structure

#	Structures	Classes
1	Structures "value-types".	Classes are "reference-types".
2	Structure instances (includes fields) are stored in stack. Structures doesn't require Heap.Structure instances (includes fields) are stored in stack. Structures doesn't require Heap.	Class instances (objects) are stored in Heap; Class reference variables are stored in stack.
3	Suitable to store small data (only one or two values).	Suitable to store large data (any no. of values)
4	Memory allocation and de-allocation is faster, in case of one or two values.	Memory allocation and de-allocation is a bit slower.
5	Structures doesn't support Parameter-less Constructor.	Classes support Parameter-less Constructor.
6	Structures doesn't support inheritance (can't be parent or child).	Classes support Inheritance.

STRUCTURES

Class (vs) Structure

#	Structures	Classes
7	The "new" keyword just initializes all fields of the "structure instance".	The "new" keyword creates a new object.
8	Structures doesn't support abstract methods and virtual methods.	Classes support abstract methods and virtual methods.
9	Structures doesn't support destructors.	Classes support destructors.
10	Structures are internally derived from "System.ValueType". System.Object -> System.ValueType -> Structures	Classes are internally and directly derived from "System.Object". System.Object -> Classes
11	Structures doesn't support to initialize "non-static fields“, in declaration.	Classes supports to initialize "non-static fields“, in declaration.
12	Structures doesn't support "protected" and "protected internal" access modifiers.	Classes support "protected" and "protected internal" access modifiers.
13	Structure instances doesn't support to assign "null".	Class's reference variables support to assign "null".

STRUCTURES

Comparison Table - Class (vs) Structure

Class Type	Can Inherit from Other Classes	Can Inherit from Other Interfaces	Can be Inherited	Can be Instantiated
Normal Class	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	No
Interface	No	Yes	Yes	No
Sealed Class	Yes	Yes	No	Yes
Static Class	No	No	No	No
Structure	No	Yes	No	Yes

Type	1. Non-Static Fields	2. Non-Static Methods	3. Non-Static Constructors	4. Non-Static Properties	5. Non-Static Events	6. Non-Static Destructors	7. Constants
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	Yes	No	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Static Class	No	No	No	No	No	No	Yes
Structure	Yes	Yes	Yes	Yes	Yes	No	Yes

STRUCTURES

Comparison Table - Class (vs) Structure

Type	8. Static Fields	9. Static Methods	10. Static Constructors	11. Static Properties	12. Static Events	13. Virtual Methods	14. Abstract Methods	15. Non-Static Auto-Impl Properties	16. Non-Static Indexers
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	No	No	Yes	Yes	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Static Class	Yes	Yes	Yes	Yes	Yes	No	No	No	No
Structure	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes

STRUCTURES

Constructors in Structures

- C# provides a parameter-less constructor for every structure by default, which initializes all fields.
- You can also create one or more user-defined parameterized constructors in the structure.
- Each parameterized constructor must initialize all fields; otherwise, it will be a compile-time error.
- The "new" keyword used with structure, doesn't create any object / or allocate any memory in the heap; It is just a syntax to call the constructor of the structure.

Example

```
public StructureName( datatype parameter)
{
  field = parameter;
}
```

STRUCTURES

Read-only Structures

- All fields are readonly.
- All properties have only 'get' accessors (readonly properties).
- There is a parameterized constructor that initializes all the fields.
- You don't want to be allowed to change any field or property of the structure.
- Methods can read fields, but can't modify them.
- 'Readonly structures' is a new feature in C# 8.0.
- This feature improves the performance of structures.

Example

```
readonly struct Student
{
  public readonly int studentId;
  public string studentName { get; }
  public Student( )
  {
    studentId = 1;
    studentName = "Scott";
  }
}
```

STRUCTURES

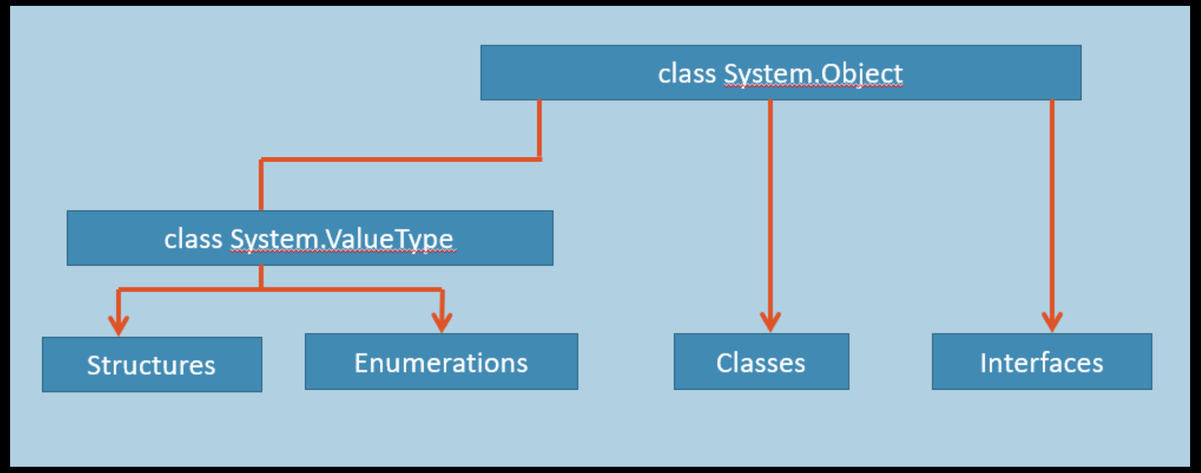
Primitive Types as Structures

- All primitive types are structures.
- For example, "sbyte" is a primitive type, which is equivalent to "System.SByte" (can also be written as 'SByte') structure.
- In C#, it is recommended to always use primitive types, instead of structure names.

SYSTEM.OBJECT CLASS

What

- The "System.Object" is a pre-defined class, which is the "Ultimate super class (base class)" in .net.
- All the classes and other types are inherited from System.Object directly / indirectly.
- All C# classes, structures, interfaces, and enumerations are children of System.Object class.
- Every method defined in the Object class is available in all objects in the system as all classes in the .NET Framework are derived from the Object class.
- Derived classes can override Equals, GetHashCode, and ToString methods of Object class.
- System.Object class is meant for achieving "type safety" in C#.



SYSTEM.OBJECT CLASS

Methods

```
namespace System
{
  class Object
  {
    virtual bool Equals( object value );
    virtual int GetHashCode( );
    Type GetType( );
    virtual string ToString( );
  }
}
```

bool Equals(object value)

Compares the current object with the given argument object; returns true, if both are same objects; returns false, if both are different objects.

int GetHashCode (object value)

Returns the a number that represents the object. It is not guarantee that the hash code is unique, by default.

Type GetType( )

Returns the name of the class (including namespace path), based on which, the object is created.

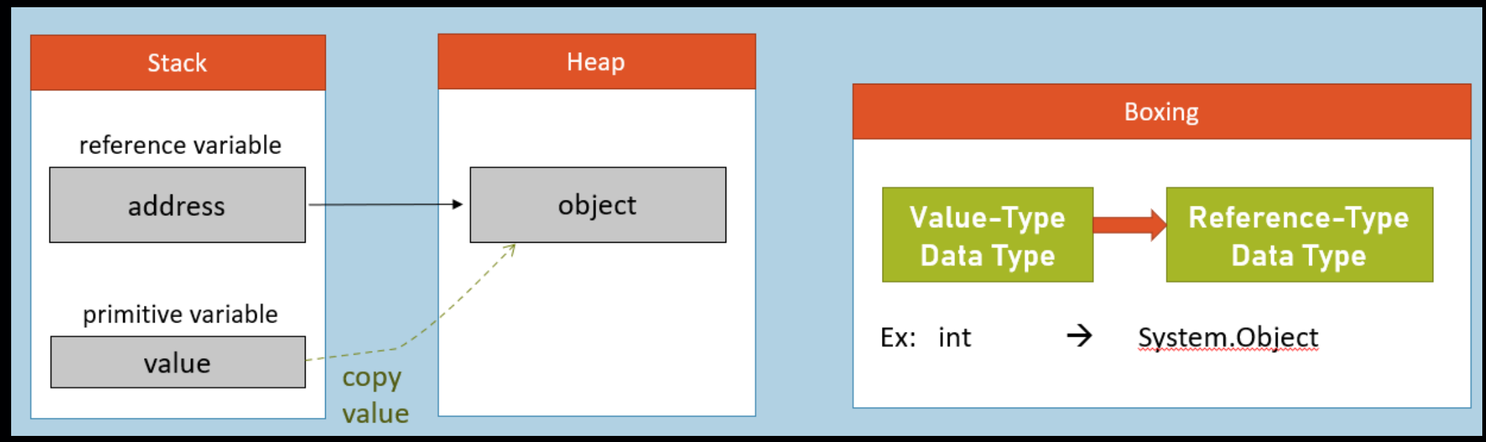
SYSTEM.OBJECT CLASS

string ToString( )

By default, it returns the name of the class (including namespace path), based on which, the object is created.
It is virtual method, which can be overridden in the child class.

Boxing

It is a process of converting a value from "Value-Type Data Type" to "Reference-Type Data Type", if they are compatible data types.
This can be done automatically [no need of any syntax].



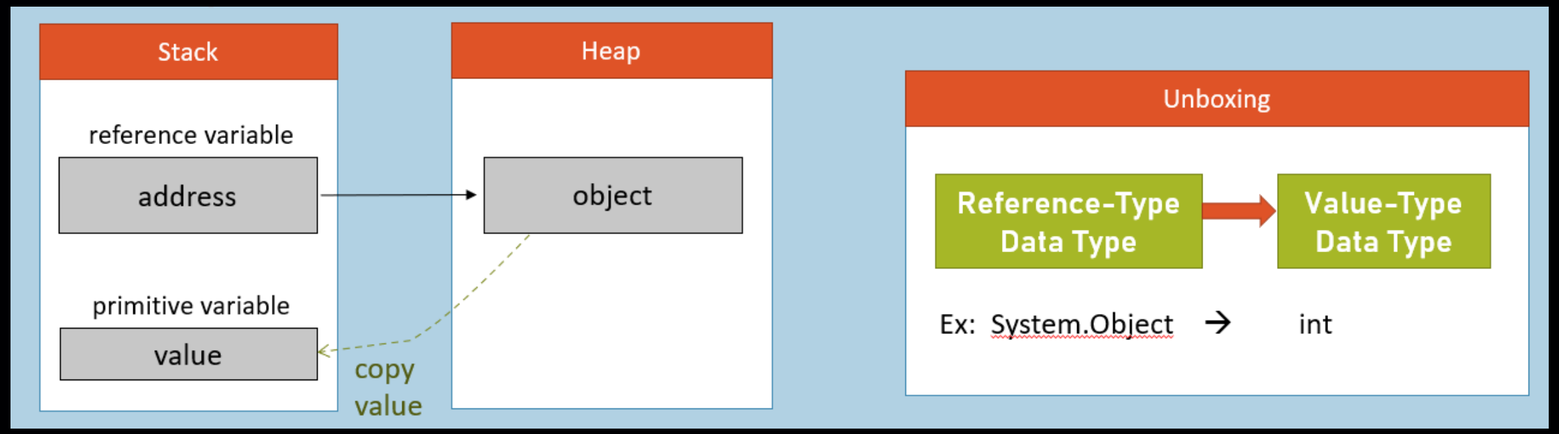
SYSTEM.OBJECT CLASS

Unboxing

It is a process of converting a value from "Reference-Type Data Type" to "Value-Type Data Type", if they are compatible data types.
This should be done explicitly (by using explicit casting).

Syntax

(DestinationDataType)SourceValue



GENERICS CLASS

What

Generic class is a class, which contains one or more "type parameters".

You must pass any data type (standard data type / structure / class), while creating object for the generic class.

How

```
class ClassName<T>
{
  public T FieldName;
}
```

```
ClassName<int> referenceVariable = new ClassName<int> (
);
```

GENERICS CLASS

Advantage

- The same field may belong to different data types, w.r.t. different objects of the same class.
- You will decide the data type of the field, while creating the object, rather than while creating field in the class.
- It helps you in code reuse, performance and type-safety.
- You can create your own generic-classes, generic-methods, generic-interfaces and generic-delegates.
- You can create generic collection classes. The .NET framework class library contains many new generic collection classes in System.Collections.Generic namespace.
- The generic type parameter (T) acts as "temporary data type", which represents the actual data type, provided by the user, while creating object.
- You can have multiple "generic type parameters" in the same class (for use for different fields.
- Generics are introduced in C# 2.0.

GENERIC CONSTRAINTS

What

Generic Constraints are used to specify the types allowed to be accepted in the "generic type parameter".

- where T : class
- where T : struct
- where T : ClassName
- where T : InterfaceName
- where T : new( )

How

```
class ClassName<T> where T : class
{
  public T FieldName;
}
```

```
ClassName<int> referenceVariable = new ClassName<int> ( );
```

GENERIC CONSTRAINTS

Advantage

You can restrict what type of data types (class names) allowed to be passed while creating object.

- In C#, constraints are used to restrict a generics to accept only particular type or its derived types.
- By using 'where' keyword, we can apply constraints on generics.
- You can apply multiple constraints on generic classes or methods based on your requirements.

GENERIC METHODS

What

Generic Method is a method that has one or more generic parameter(s).
You can restrict what type of data types to be allowed to be passed to the parameter while calling the method.

- In C#, constraints are used to restrict a generics to accept only particular type or its derived types.
- By using 'where' keyword, we can apply constraints on generics.
- You can apply multiple constraints on generic classes or methods based on your requirements.

How

Generic Method - Example

```
public void MethodName<T>
{
}
```

Calling Generic Method - Example

```
MethodName<datatype>( valueHere );
```