# The Type System

## *Defining the structure of Values*

---

**THIS DOCUMENT COVERS**



---

# Overview of Type

A .NET executable or DLL generally doesn't contain x86 machine instructions and as such can't just be directly mapped into a process address space in the same way as a traditional unmanaged DLL. The .NET runtime provides the virtual execution environment which must be loaded before the managed code can be executed. When a DLL/EXE is first loaded into a .NET process the loader takes responsibility for converting its meta-data into in-process representations of classes and their members. The meta-data describes the types included in the module (name, method signatures, inheritance relationships etc.) When a .NET executable is started, by double clicking for instance, stub code in its PE/COFF file starts up the .NET runtime.

The CLR provides memory management, garbage collection, thread execution and JIT compilation. When a .NET executable starts the managed heap is initialised and a thread is created with a 1MB stack. Each .NET executable contains in its header a token that is used to index into a table of IL methods and locate the point of entry of the .NET program. The .NET JIT compiler must then compile the method's IL into machine level instructions in order for it to run. The main method will create data items, modify those data items and call other methods thus controlling the logical flow of the process.

In .NET every item of data created has a type that defines

- How much memory needs to be set aside for that item
- The set of possible values that an instance of that type can take
- The operations that can be carried out on that item

For the purposes of this article we will refer to these pieces of data as object instances or sometimes just objects. Typically, object instances are assigned to variables. Variables are synonyms for stack locations. The data actually stored at a variable's stack address depends on whether the object's type is a struct or a reference type. When a struct is created and assigned to a local variable the data for the struct is allocated on the stack and the variable is a synonym for the start of the actual data. When a reference type is created its data is stored on the heap and the variable's stack location simply holds a reference to the heap.
As C# is a statically typed language both variables and objects have a type. The variables type is known at compile time which allows the compiler to determine the valid operations on instances of that type. The memory requirements and the valid operations that can be performed on a data item that represents a date are different from the memory requirements and valid operations for a data item that represents an integer.

# Categorising Type

## ValueType and Reference Type

Perhaps one of the most important means of categorising type is as either reference or value type. Any type that inherits from `System.ValueType` is a value type and everything else is a reference type.

### VALUE TYPE

If the variable is of value type, then its name is a direct synonym for the location on the stack where the raw bytes can be found. Value types are known as structs. If a value type instance is living on the stack then its type is essentially implicit. Value types are implicitly sealed and cannot be sub-classed (although they can implement interfaces). When a value type is used as a local variable or as a formal parameter its fields are directly stored on the stack. No pointer dereference is needed to access its fields. As a value type does not reside on the managed heap it can reduce the number of garbage collections needed. If we assign one value type to another there is always a copy of the raw bytes of data. The following list describes when we might choose to make our type a value type

#### WHEN TO USE A VALUETYPE

- Type acts like a primitive and is immutable
- Type doesn't need to inherit from another type
- Other types wont need to inherit from it
- Instances of the type are small OR
- Instances of the type are not passed as method parameters

### REFERENCE TYPE

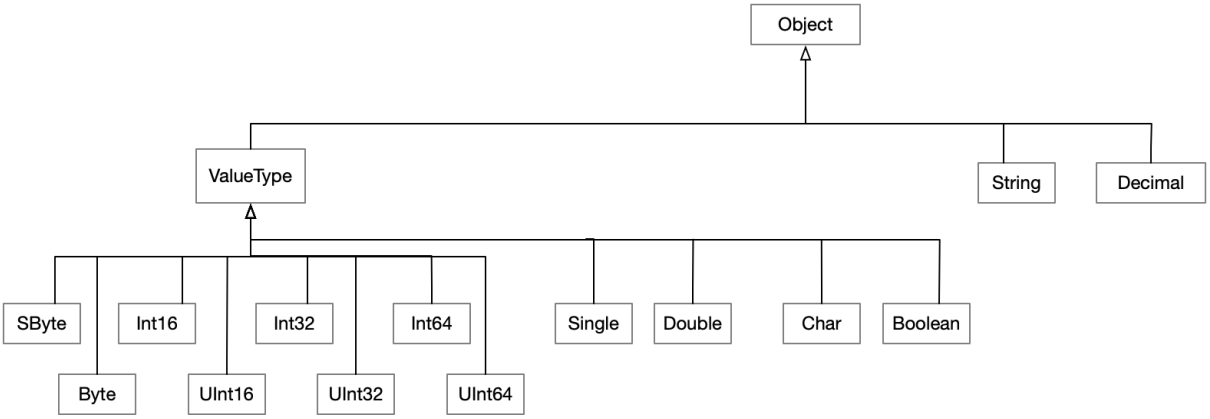
If a variable is of a reference type then its name is a synonym for the stack address that holds its data. In addition to the memory required for the type's fields, additional space is also required for the object's type pointer and a sync block for locking. This extra level of indirection is very powerful and allows us to implement features such as polymorphism. It also, however, necessitates increased memory usage and pointer indirection. All user defined types created with the class keyword are reference types. Arrays, delegates and interfaces are also reference types.

## Comparison of Value and Reference Types

- Value types extend ValueType and Reference types extend Object
- Value types are allocated on the stack, reference types on the heap
- Value types cannot be extended and cannot extend any type other than ValueType
- Assignment of value type results in field by field copy.
- Assignment of reference type results in copy of memory address
- Multiple reference type variables can refer to the same instance
- Value type variables always have their own state
- Value types have two representations boxed and unboxed

## Predefined Types

.NET provides a set of predefined types (also known as built in types). The predefined types are directly supported by the compiler which maps keywords to FCL types.




The following provides the mapping between predefined types and their corresponding FCL type. All of the predefined types highlighted in blue are considered primitive types, meaning they map very closely to the underlying machines instruction set.

| | | |
|---|---|---|
| sbyte | System.Sbyte | Signed 8 bit integer |
| byte | System.Byte | Unsigned 8 bit integer |
| short | System.Int16 | Signed 16 bit integer |
| ushort | System.UInt16 | Unsigned 16 bit integer |
| int | System.Int32 | Signed 32 bit Integer |
| uint | System.UInt32 | Unsigned 32 bit integer |
| long | System.Int64 | Signed 64 bit integer |
| ulong | System.UInt64 | Unsigned 64 bit integer |
| float | System.Single | 32 bit float |
| double | System.Double | 64 biut float |
| char | System.Char | character |
| bool | System.Boolean | boolean |
| decimal | System.Decimal | |
| string | System.String | |
| object | System.Object | |
| Dynamic | System.Object | |

**PREDEFINED TYPES**

- Supported by the compiler
- Referred to as built-in types
- Value types are object and string
- Reference types are the  numeric types, bool and char

**PRIMITIVE TYPES**

- Prefefined types that map closely to the underlying instruction set

# Functions

In .Net all of the following language features are implemented as functions

- Constructors
- Properties
- Events
- Indexers
- Methods
- Finalizers

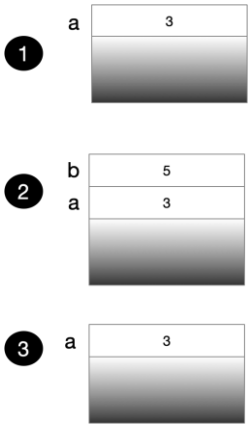
A Function consists of local variables, formal parameters and a return value. We consider each in turn. Functions can call other functions via a well defined protocol that defines the semantics. The calling and the called function communicate information via an activation frame. The caller supplies a this pointer, any actual parameters (arguments) and a return address. When it completes, the called function gives back a return value and returns to the callers address.

## Local variables

### Scope

Local variables are allocated on the stack. Code blocks can be nested to create different scopes. Variables in any given code block come into scope when the flow of execution crosses its opening curly brace. At this stage space is allocated for the variables on the stack. When the flow of execution moves out a code block by exiting its closing curly brace the variables go out of scope by popping them from the stack.

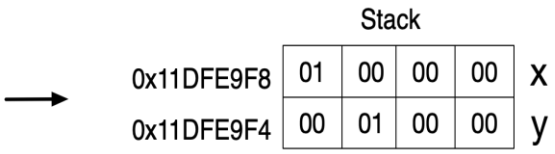
```
void Main()
{
    int a = 3; ❶
    {
        int b =5 ;❷
    }❸
}
```




## Declaring and Initialising variables

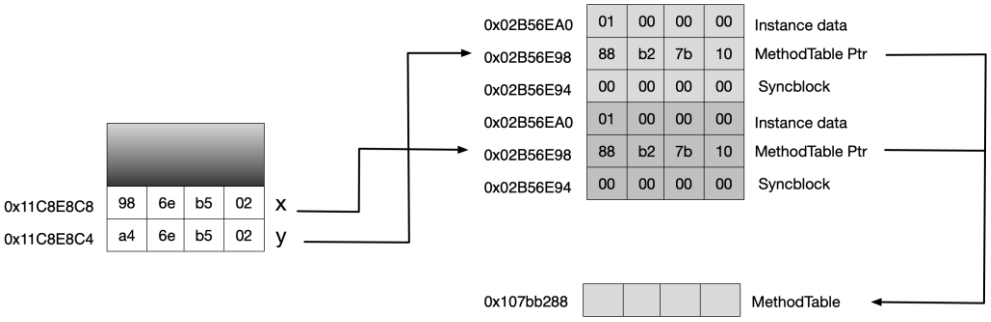
A variable name is a synonym for a stack address. If a variable's type extends System.ValueType then its stack address directly holds the bytes that constitute the variables value. The number of bytes allocated on the stack for a single value type variable depends on the variables type. In this way the type of a stack variable is implicit and the compiler knows how to treat the bytes there. As value types are implicity sealed there is no virtual dispatch to worry about.

```
public void ValueTypeOnStack()
{
 uint x = 01;
 uint y = 256;
}
```




If a variables type is of a reference type then it's stack address holds a pointer to the location on the managed heap where the raw bytes live. This extra level of indirection is very powerful and allows us to implement features such as polymorphism. It also, however, necessitates increased memory usage and pointer indirection

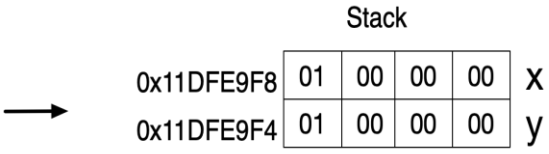
```
public void ReferenceTypeOnStack()
{
  MyType x = new MyType();
  MyType y = new MyType();
}

class MyType
{
  private uint _val = 1;
}
```




## COPY BY VALUE

By default, asigning one stack variable to another makes a copy of the variables bytes on the stack. For variables of value type this has the effect of making a copy of the variables value.

```
public void ValueTypeOnStack()
{
 uint x = 01;
 uint y = x;
}
```




For rerence types copying has the effect of making a copy of the pointer such that we now have two references to the same heap object

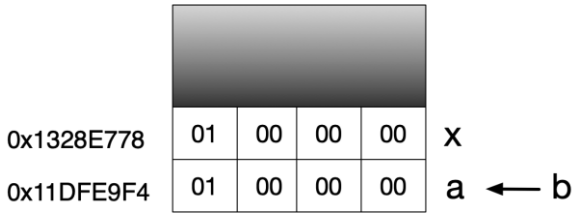
```
public void CopyingReferenceStackVariabes()
{
    MyType x = new MyType();
    MyType y = x;
}
```

### COPY BY REFERENCE

C#7.0 introduced reference variables. This enables us to asign one stack variable to another such that both variables are synonyms for the same stack address.

**Figure 1 Copy By Reference**

```
byte x = 1;
byte a = 5;

ref byte b = ref a;

b = 4;
Console.WriteLine(a);
```




## Formal parameters and arguments

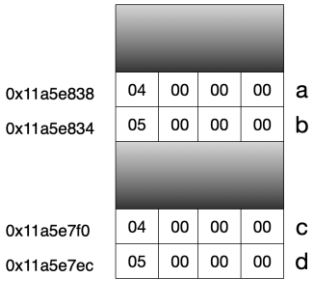
A method signature declaration specifies a sequence of formal parameters that determine the arguments (actual parameters) that must be passed to the method by calling code. If the formal parameters are not marked with any specific modifiers then the default behaviour is pass by value. The ref and out modifiers can be used to specify pass by reference behaviour. We consider each in turn now.

### PASS BY VALUE FOR VALUE TYPES

When using pass by value semantics space is allocated on the stack for the formal parameters and the actual parameters are then copied into these locations. If the type of the formal parameter is a struct then the actual value of the object itself is copied from actual to formal parameter. If the type of the formal parameter is a class then a reference to the object is copied from the actual to the formal parameter. The following code highlight pass by value for value types.

**Figure 2 Pass by value for value types**

```
public void Foo()
{
    uint a = 4;
    uint b = 5;
    A(a,b);
}

public static void A(uint c, uint d)
{
  B(c, d);
}
```

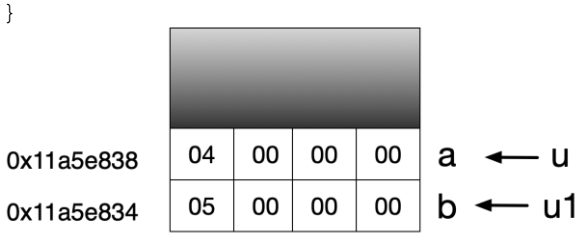



### PASS BY REFERENCE FOR VALUE TYPES

If we mark a method's parameters with the ref modifier then the calling convention is no longer pass by value. The formal parameters are just synonyms for the caller's arguments. The function can modify the calling code's variables. With ref parameters the caller must initialise the arguments before calling the functions. We can also mark a method's parameters with the out modifier. This enables a function to pass out values and doesn't require that variables be initialised before the method is called.

**Listing 1 Pass by reference for value types**

```
public void PassbyRef()
{
  uint a = 4;
  uint b = 5;
  C(ref a, out b);
}

private void C(ref uint u, out uint u1)
{
  u1 = 10;

}
```



## Return values

By default a methods return values are passed by value and as with parameters a copy is made. From C#7.0 onwards, support is provided for reference return values. The following code would output John to the console. The method has to return a reference to something whose scope outlives that method, typically a field on a class or a variable passed as an argument to the function. The functions local varaiables cannot be returned by value.

**Figure 3 Reference return values**

```
void Main()
{
    Person p =new Person() {_name="Kenny"};
    ref string b = ref GetString(p);
    b = "John";
    Console.WriteLine(p);
}

public ref String GetString(Person p)
{
    return ref p._name;
}

public class Person
{
    public String _name ;
    public override string ToString() => _name;
}
```

## Method Dispatch

The CLR provides two call instruction which vary in the way they find the address of the JITed native code to be invoked.

- call
- callvirt

When the compiler compiles a method it inserts flags into the assembly's method definition table (instance, static, virtual). When a calling method is compiled it examines these flags to determine which of the two IL method call instruction to use.

Methods executing virtually look at the method table of the object on which the method is dispatching in order to determine the method table slot (address) to use (runtime). Non-virtual methods enable the JIT to burn the address of the target method table address because it knows the location at compile time. Confusingly call is sometimes used to call virtual methods non-virtually and callvirt is sometimes used to call non-virtual methods.

### Call

Call is used to invoke static and non-virtual methods and even sometimes virtual methods non-virtually. When calling instance methods, call assumes the reference is not null – no checking is performed. Call can be used in special cases to call virtual methods non-virtually

- If a method calls base.<method> to prevent recursive overflow
- Calling value type methods which are implicitly sealed

### Callvirt

Callvirt is used to invoke virtual methods. The methods are called polymorphically.

### NON-VIRTUAL METHOD

Method dispatch in C# is non-virtual by default. Consider the following code. As the method is non-virtual the method invoked is determined at compile time based on the compile time type of the variable. As the subclass C does not have an implementation of the method DoSomething then the method from its immediate parent in the hierarchy is used.

```
void Main()
{
    C c = new C();
    B b = c;
    A a = b;

    c.DoSomething();
    b.DoSomething();
    a.DoSomething();
}

public class A
{
    public void DoSomething() => Console.WriteLine("A:DoSomething");
}

public class B : A
{
    public void DoSomething() => Console.WriteLine("B:DoSomething");
}

public class C : B {}
```

The output is then

```
B:DoSomething
B:DoSomething
A:DoSomething
```

> This code generates a compiler warning telling us we should use the new keyword to explicitly show that the subclasses method will hide the base class method

### VIRTUAL METHODS

Where the base class method is defined as virtual it supports polymorphism. In addition a subclass needs to explicitly mark its implementation with the override keyword in order to facilitate virtual dispatch

```
void Main()
{
    SubClass c = new SubClass();
    BaseClass bref = c;

    // As the method is defined as virtual in the base class
    // and overriden in the subclass all calls are polymorphic.
    // The run-time type of the actual object, and not the
    // compile-time type of the reference, determines which method
    // is called

    bref.VirtualMethod();
}

class BaseClass
{
    public BaseClass()
    {
          Console.WriteLine("BaseClass()");
    }

    public virtual void VirtualMethod()
    {
          Console.WriteLine("BaseClass.VirtualMethod()");
    }
}

class SubClass : BaseClass
{
    public SubClass() { Console.WriteLine("SubClass()"); }


    public override void VirtualMethod()
    {
          Console.WriteLine("SubClass.VirtualMethod()");
    }
}
```

> **DISPATCH IS NOT VIRTUAL BY DEFAULT**
>
> If we mark the base class method as `virtual` and do not mark the subclass method as `override` then the subclass method will **hide** the base class method. The compiler will warn us of this.

### VIRTUAL METHODS AND THE NEW KEYWORD

Even if the base class method is virtual when we use the new keyword in the subclass we hide the base implementation and prevent polymorphism taking place. The type of the method invoked is then again determined by compile time type of reference.

```
void Main()
{
    SubClass c = new SubClass();
    BaseClass bref = c;

    // Even though the base class method is virtual, because we
    // didnt override it and instead used the new keyword, once
    // again the method invoked is based on the compile time type

    // Call SubClass.VirtualMethod
    c.VirtualMethod();

    // Call BaseClass.VirtualMethod
    bref.VirtualMethod();
}

class BaseClass
{
    public BaseClass()
    {
          Console.WriteLine("BaseClass()");
    }

    public virtual void VirtualMethod()
    {
          Console.WriteLine("BaseClass.VirtualMethod()");
    }
}

class SubClass : BaseClass
{
    public SubClass() { Console.WriteLine("SubClass()"); }


    public new void VirtualMethod()
    {
          Console.WriteLine("SubClass.VirtualMethod()");
    }
}
```

### OVERLOADING METHODS AND INHERITANCE

The overloaded method selected in the below piece of code is statically evaluated at compile time.

```
void Main()
{
    Base b = new Sub();
    DoIt(b);
}

public void DoIt(Base b) => WriteLine("DoIt(Base)");
public void DoIt(Sub b) => WriteLine("DoIt(Sub)");

public class Base { }
public class Sub : Base { }
```

Even though the runtime type type is Sub we actually see the output

```
DoIt(Base)
```

### STATIC METHODS AND METHOD RESOLUTIONS

In the following code it is of interest which method is actually called. In actual fact even though the runtime type of c is SubClass the method is resolved statically at compile time and chooses the overload taking an instance of the BaseClass

```
void Main()
{
    MainClass.MainMethod
}


class BaseClass { }

class SubClass : BaseClass { }

class MainClass
{
    public static void AMethod(BaseClass br) { }
    public static void AMethod(SubClass br) { }

    public static void MainMethod()
    {
          BaseClass c = new SubClass();

          AMethod(c);
    }
}
```

### SEALED CLASSES AND METHODS

- An overridden function member can seal its implementation preventing further sub-classing
- A sealed member can be implemented non-polymorphically in a subclass using new
- A class can be sealed preventing all sub-classing

## Delegates

A delegate enables one to capture behaviour inside an object. The behaviour does not need to be executed at the point the delegate is created. The object can be passed around and stored for future use. At the point where the behaviour is required it is executed by calling the invoke method on the delegate object which in turn executes the encapsulated behaviour.

It is important to be able to distinguish between a delegate type and a delegate instance. The following code listing show how we declare a delegate type.

```
public delegate void Strdel( string str );
```

This code will cause the compiler to create a subclass of System.MulticastDelegate. The delegate type defines the signature of the encapsulated method. If we want to use the delegate type we need to create an instance of it.

```
Strdel delA = new Strdel(this.SomeInstanceMethod);
```

This provides for the delegate instance both the method to be executed and the target object to execute is on (this). Where the delegate is being instantiated in the same object as the target the this can be ommited.

```
Strdel delC = new Strdel(SomeInstanceMethod);
```

this code can be further shorted using C# 2.0 and later via a mechanism known as method group conversion. The compiler will allocate the correct target to the delegate based on the delegate type.

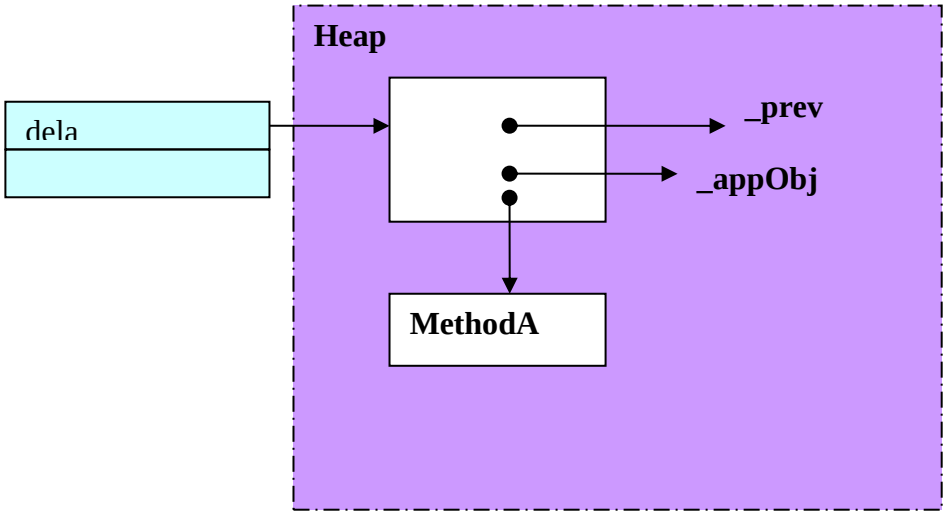
```
Strdel delD = SomeInstanceMethod;
```

### ADVANTAGES

- Provide a level of indirection between caller and method implementation
- Dynamically wire up method caller and target method
- Delegate type provides a protocol to which the caller and targer conform to
- Delegate instances refer to one or more target methods conforming to the protocol
- Caller invokes a delegate and the delegate inokes the method
- Similar to C function pointers

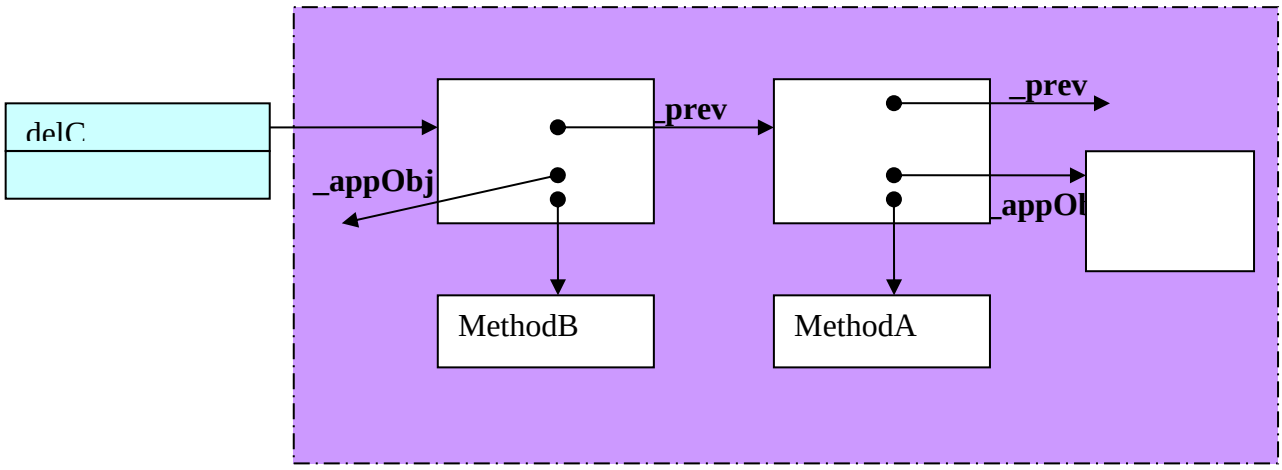
## SINGLE MULTICAST DELEGATE

```
public delegate void strdel( string str );
strdel delA = new strdel(this.SomeInstanceMethod);
```




## CHAINING

```
strdel delB = new strdel(SomeClass.SomeStatic);
strdel delC = delA += delB;
```

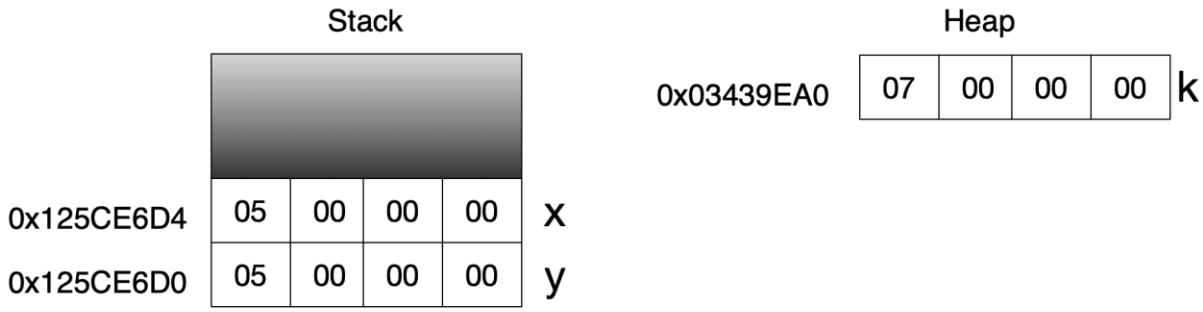


### CAPTURED VARIABLES

Any local variables and parameters of the anonymous functions enclosing scope are called outer variables and they are available to the anonymous delegates. We say a reference to the outer variable is captured when the delegate is created

```
public  void GetAction()
{
    uint i = 5;
    uint j = 6;
    uint k = 7;

    Action a = () => Console.WriteLine(k);
}
```

*Figure 4Captured Variables - 1*




## Lambdas

## Local Methods

Locals methods have the following advantages over lambdas

- Don’t require the creation of a delegate type and instance
- Support recursion
- Don’t require same indirection as delegate so more efficient
- Can access variables of containing method more efficiently

# Predefined Types

## Char

To understand C# characters in any depth we need to be able to distinguish between a character set and an encode.

A character set maps numeric values to alphanumeric characters. For example in Unicode the finnish character Ö is mapped as follows

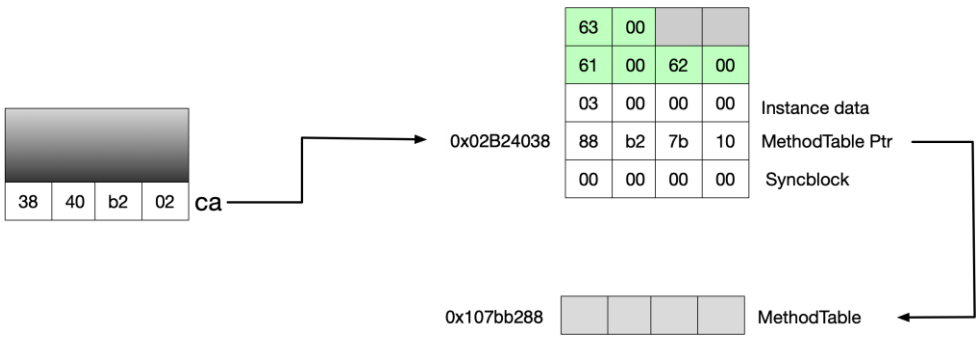
```
d6 -> Ö
```

In the early years of computing the ASCII character set was prevalent. ASCII supports 128 characters. C# characters use the Unicode character set which has about 1million slots of which about 100,000 are currently allocated.

An encoding defines how the numeric codes for each character are converted to binary. At runtime the .NET runtime encodes characters using UTF-16 which allocates 16 bits to each character. 16 bits support 65536 characters which in theory is not enough to cover every Unicode code point. In practice for most situations it is sufficient as it covers all the code point in a subset of Unicode known as Basic Multilingual plane.

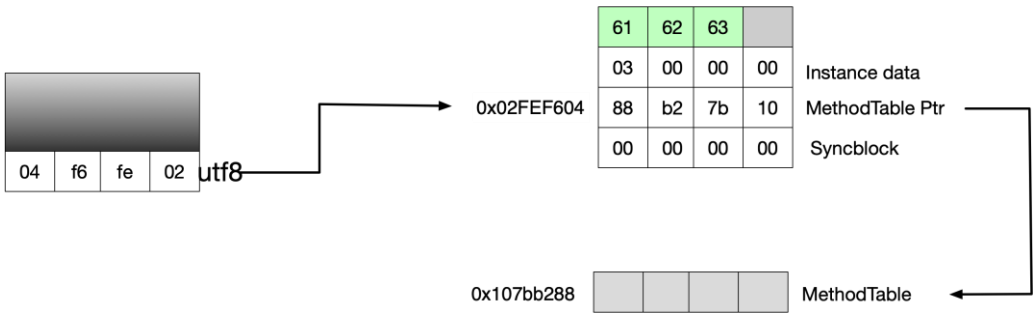
Consider the in-memory layout for a char array of three characters using the UTF-16 encoding.

```
char[] ab = {'a', 'b', 'c'};
```



Now if we encode to UTF-8 which is the default for .NET streaming we see the simple characters require only one byte each

```
byte[] utf8 = Encoding.UTF8.GetBytes(ab);
```




## Strings

A string is an immutable sequence of Unicode characters. Because strings are immutable concatenating long sequences together is inefficient. In such cases use of a StringBuilder is expedient. The < and > operators are not supported for strings. Instead the Compare method can be used

## Nullable types

`T?` is mapped to the type `System.Nullable<T>` which is a lightweight value type with logic to support being considered null. The compiler has language support for nullable types.

```
int? nullable = null;
// Equivalent
bool isNull = nullable == null;
bool isNull2 = !nullable.HasValue;
```

Boxing is clever and boxes the actual value and not the nullable wrapper as boxed types can already be null by virtue of being reference types.

```
object o = nullable;
```

Unboxing is also supported.

```
int? b = o as int?;
WriteLine(b);
```

The compiler lifts operators from the basic value type to the nullable. The following two methods can be considered equal.

```
int? x = 5;
```

```
int? y = 5;
bool eq1 = x == y;
bool eq2 = (x.HasValue && y.HasValue)
```

## Anonymous Types

In the following snippet the variable place must be off type var because it is anonymous. Its actually type is generated by the compiler.

```
var place = new { Street = "Worple Road", Number = 20 };
WriteLine(place.Street);
```

Equality and hashing are overridden to use value equality

```
var a = new { Street = "Worple Road", Number = 20 };
var b = new { Street = "Worple Road", Number = 20 };
WriteLine(a.Equals(b)); // true
WriteLine( a == b); // false

var aa = new Dictionary<Object,string>();
aa[a] = "hello";
WriteLine(aa.ContainsKey(b));  // true
```

The compiler is smart enough to generate only one type in the following code

```
var c = new { Street = "Worple Road", Number = 20 };
var d = new { Street = "Worple Road", Number = 20 };
WriteLine(c.GetType() == d.GetType()); // true
```

We can create arrays of anonymous type objects

```
var ar = new[]
{
      new { Street = "Worple Road", Number = 20 },
      new { Street = "Worple Road", Number = 25 }
}
```

We cannot, however, return anonymous types from methods.

## Tuples

Tuples allow one to return multiple values from a method without using out values. The compiler generates types of System.ValueTuple when it sees the following syntax.

```
var myAddress = (23,"Worple Road");
```

As with all value types assigning makes a copy.

```
var a1 = (23,"Worple Road");
var a2 = a1;
a2.Item1 = 46;
WriteLine(a1); // (23, "Worple Road");
```

In order to return a tuple from a function we use an explicit tuple type.

```
(int, string) GetAddress() => (23, "Worple Road");
(int, string) address = GetAddress();
WriteLine(address);
```

We can use tuples with generics

```
IList<(int,string)> l = new List<(int, string)>()
{
      (23,"Worple Road"),
      (46,"Worple Road")
};
```

We can name tuple members

```
var a3 = (Number:1,Road:"Worple Road");
WriteLine( a3.Number); // 1
(int Number, string Road) a4 = (Number:1,Road:"Worple Road");
(int Number, string Road) a5 = (2,"Worple Road");
```

We can deconstruct tuples

```
(int Number, string Road) a6 = (2,"Worple Road");
(int num, string rd) = a5;
WriteLine(rd);
```

As with anonymous types equality is overridden to do value equality

```
// Tuples use value equality on both operator and method
(int Number, string Road) a7 = (2,"Worple Road");
(int Number, string Road) a8 = (2,"Worple Road");
WriteLine(Object.ReferenceEquals(a7,a8)); // false
WriteLine(a7 == a8); // true
WriteLine(a7.Equals(a8)); // true
```

## Boolean

The conditional operators && and || short circuit

The conditional operator & and | do not short circuit. These are not biwise operators on bool in Csharp ???

# Numeric types

### INTEGERS

.NET provides 8, 16, 32 and 64 bit integral types in both signed and unsigned versions. When the compiler sees an integral literal it chooses the first integral type in the list int, uint, long, ulong that accommodates the literal.

On any integral arithmetic overflow is silent and causes wraparound by default. Division on integral types always causes truncation of fractional parts.

### FLOATING POINT

.Net provides float, double and decimal floating point types of size 32, 64 and 128 bits respectively. Literals with no suffix are assumed to be doubles. The following list gives the properties of floating point numbers.

- Float and double internally represent in base 2
- For this reason only numbers expressible in base 2 are precisely represented
- Most literals with a fractional component in base 10 will not be precisely represented
- Decimal works in base 10 and can represent number is base 10 accurately
- Decimal can also represent number expressible in bases which are factors of 10 accurately
- Neither double or Decimal can accurately represent a fractional number whose base 10 representation is repeating
- Double is base 2, moderate precision with large range and high performance
- Decimal is base 10, high precision with moderate range and low performance

The following lists the special results of floating point operations

- When using floating point types .NET doest throw DivideByZero exception
- Not throwing exception is mathematically correct behaviour
- 1.0 / 0.0 = Infinity
- -1.0 / 0.0 = NegativeInfinity

- 1.0 / -0.0 = Negative Infinity
- -1.0 / -0.0 = Infinity
- 1 / 4 = Throw DivideByZeroException
- Dividing, multiplying or subtraction expressions involving NaN result in Nan
- Dividing, multiplying or subtraction expressions involving Infinity result in NaN

### Numeric Literals

By default the compiler infers a numeric literal to be of either floating point or numeric type according to the following rules

1. If the literal contains a decimal point or the exponentiation symbol E it is assumed to be a double
2. Otherwise it is assumed to be the first integral type in the following list big enough to hold it int, uint, ulong ???

The suffixes F,D,U,L can explicitly define a literal as Float, Double, Unsigned or Long

# Numeric Types

### Arithmetic Operators

When performing arithmetic operators on integral types the CLR performs overflow which guarantees wraparound. One can use the checked keyword on an expression or statement block to cause the runtime to throw an overflow exception.

Division operators on integers always truncate the result.

# Type conversion and casting

In .NET the C# cast operator can perform the following operations.

- Numeric conversion
- Reference conversion
- Boxing/Unboxing conversion
- Custom conversion/Operator overloads

An implicit conversion is allowed when no loss of data is possible. An explicit cast is requited where there is a potential to lose data.

## Numeric Conversion and arithmetic operators

For numeric types implicit conversion is allowed where there is no possible loss of data. If there is a possibility that data will be lost an explicit cast will be needed.

By default overflow checking is turned off but the compiler can be instructed to turn it on. Checked an unchecked operators can be used

## Conversion

| Source | Destination | Implicit/Cast Needed | Notes |
|---|---|---|---|
| Any integral type | Any floating point type | Implicit | Magnitude is maintained but precision may be lost |
| Any floating point type | Any integral type | Cast needed | Fractional part truncated. Use system.Convert to round |

How overflow works?

How checking works?

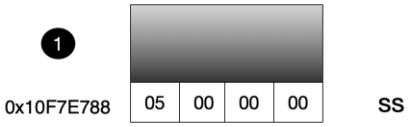
## Boxing and unboxing conversion

Often we need to treat a value type as a reference type. In such case the compiler will generate code to copy a value type value onto the heap. In order to highlight how value types work we will make sure of the following simple value type which implements an interface

```
public interface Iset {void SetValue(double val_);}

public struct SimpleStruct : ISet
{
  public double X;

  public void SetValue(double val_) => X = val_;
}
```

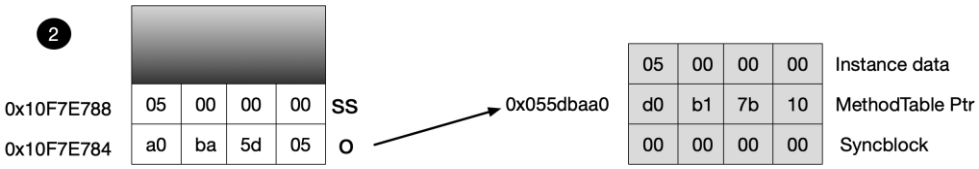
Now we perform some simple operations with instances of SimpleStruct. First we create an instance of SimpleStruct on the stack and set the value of its single field. At this point all storage is on the stack.

```
SimpleStruct ss;
ss.X = 5;
```



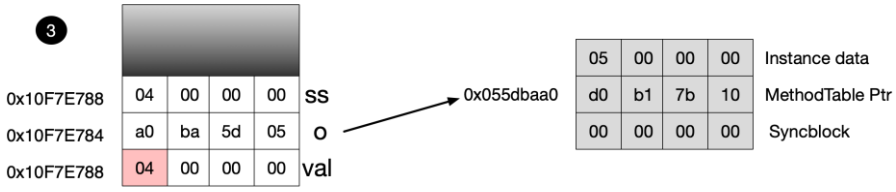

Now we asign our value type of a variable of type Object. This implicit conversion causes a boxing operation to take place. Space is allocated on the heap and the value types bits are copied onto the heap location.

```
object so = ss;
```




Now we use our setter method to change the value of our value type field. Note that our boxed heap object is not changed by any change to the original stack object as the stack values were simply copied to the heap.

```
ss.X = 4;
uint val = ss.X;
```




The result of the next piece of code is somewhat surprising. Casting our object to an instance of SimpleStruct causes an unboxing operation which copies the heap values and then changes them. So in fact the heap object is not changed at all.

```
((SimpleStruct) so).SetValue(3);
val = ((SimpleStruct) so).X;
```


Finally we show how to use an interface to change the value on the heap instance.

```
((ISet) so).SetValue(3);
val = ((SimpleStruct) so).X;
```


In general, boxing is less necessary since the introduction of generics. Boxing involves allocating enough memory on the heap to encapsulate the value type plus a type pointer and a sync block. Unboxing involves copying the instance data from the refeence object on the heap into a variable stored on the stack. Both operations incur a performane overhead. When unboxing we must use the exact type that was boxed.

# Equality

## Equality Operators (==, !=)

### Reference Types use Reference equality

```
public class MyReference { public int Value;}

…
MyReference a = new MyReference {Value = 1};
MyReference b = new MyReference {Value = 1};
WriteLine(a == b); // False
```

The equality operators `==` and `!=` are statically resolved like all operators. As such they are fast and efficient. `Object` implements the operators to perform referential equality

### Overriding The Equality Operator for Reference Types

```
public class MyReference
{
    public int Value;

    public static bool operator==(MyReference a, MyReference b)
          => a.Value == b.Value;
    public static bool operator!=(MyReference a, MyReference b)
          => a.Value == b.Value;
}
…
MyReference a = new MyReference {Value = 1};
MyReference b = new MyReference {Value = 1};
WriteLine(a == b); // true
```

Here we overridden the equality operators to carry out value equality rather than the referential equality performed by Object's implementation

### Equality Operators are Statically Resolved

```
WriteLine(a == (object)b); // false
WriteLine((object)a == b); // false
```

Because operators are statically resolved asigning a value of a more specific type to a variable of type Object cause Object operator to be used.

### Value Types Have No Default Equality Operator ImplementaTion

```
public struct MyStruct
{
    public int Value;
    public static bool operator ==(MyStruct a, MyStruct b)
          => a.Value == b.Value;
    public static bool operator !=(MyStruct a, MyStruct b)
```

```
            => a.Value == b.Value;
}

MyStruct c = new MyStruct { Value = 1 };
MyStruct d = new MyStruct { Value = 1 };
WriteLine(c == d); // true
```

## Object.Equals instance method

The object equals method provides a virtual method for carrying out equality functionality. The default implementation for objects carrying out referential equality

```
void Main()
{
    MyReference a = new MyReference {Value = 1};
    MyReference b = new MyReference {Value = 1};
    WriteLine(a.Equals(b)); // false
}

public class MyReference { public int Value;}
```

The default implementation for value types carries out value equality

```
public struct MyStruct { public int Value;}
…
MyStruct a = new MyStruct {Value = 1};
MyStruct b = new MyStruct {Value = 1};
WriteLine(a.Equals(b)); // true
```

This default implementation is however ineffient in two ways; It uses reflection for the field by field comparison and it uses boxing which is several times more expensive than the actual comparisonThe first of these problems can be overcome  by providing our own value type implementation of the Equals method that does not use reflection

```
public struct MyStruct
{
    public int Value;

    public override bool Equals(object obj)
          => Value == (obj as MyStruct?)?.Value;
}
…
MyStruct a = new MyStruct {Value = 1};
MyStruct b = new MyStruct {Value = 1};
WriteLine(a.Equals(b)); // true
```

If we want to eliminater boxing we can implement the interface IEquatable

```
public struct MyStruct : IEquatable<MyStruct>
{
    public int Value;

    public bool Equals(MyStruct other)
          => Value == other.Value;
```

```
}
```

## Object.Equals Static method

Object also provides a static equals method that provides a null checking, unified type equality comparison.

## Pluggable Equality

Consider the following type which does not itself provide any sensible implementation of GetHashCode() and Equals

```
public class Point
{
    public int X;
    public int Y;
}
```

If we try and use objects of this type as hash keys we see the following behaviour.

```
IDictionary<Point,string> a1 = new Dictionary<UserQuery.Point, string>();
a1[new Point() {X=1,Y=1}] = "hello";
WriteLine(a1.ContainsKey(new Point() {X=1,Y=1})); // False
```

We can create our our pluggable Equality comparer

```
public class PointComparer : EqualityComparer<Point>
{
    public override bool Equals(Point x, Point y)
          => x.X == y.X && x.Y == y.Y;

    public override int GetHashCode(Point obj)
          => obj.X ^ obj.Y;
}
```

And we can use it as follows which

```
var pc = new PointComparer()
var a2 = new Dictionary<UserQuery.Point, string>(pc);
a2[new Point() { X = 1, Y = 1 }] = "hello";
WriteLine(a2.ContainsKey(new Point() { X = 1, Y = 1 }));//true
```

## Structural Equality

```
int[] x = {3,4,5};

int[] y = {3,4,5};

((IStructuralEquatable)x)

      .Equals(y,EqualityComparer<int>.Default);
```

## Implementing Equality Sensibly

By default most reference types use reference equality and value types use value equality with the equality operators and equal methods being consistent. There are however some exceptions in the framework. Despite being a reference type String overrides Equals and == to use value equality

```
string a = new string('*',4);
string b = new string('*',4);
string c = a;

WriteLine(Object.ReferenceEquals(a,b)); // false
WriteLine(a==b); // true
WriteLine(a.Equals(b)); // true
```

StringBuilder uses referential equality for == and value equality for Equals

```
StringBuilder a =  new StringBuilder("Hello");
StringBuilder b =  new StringBuilder("Hello");

WriteLine(Object.ReferenceEquals(a,b)); // false
WriteLine(a==b); // false
WriteLine(a.Equals(b)); // true
```

## Equality in user defined types

Equality operators should respect the following

- Reflexive: x.Equals(x) is true
- Symmetric/Commutative: x.Equals(y) gives same result as y.Equals(y)
- Transitive: x.Equals(y) and y.Equals(z) implies x.Equals(z)
- Consistent
- Any object cannot equal null

Consider the following struct implementation

```
public struct Point: IEquatable<Point>
{
    public int X;
    public int Y;

    // 1. Implement type safe equals defined in IEquatable
    public bool Equals(Point other) => X == other.X && Y == other.Y;

    // 2. Override Equals to perform null and type check before
    // calling IEquatable method
    public override bool Equals(object obj)
    {
          if (!(obj is Point)) return false;

          return Equals((Point)obj);
    }

    // 3. Implement operators
    public static bool operator==(Point a, Point b) => a.Equals(b);
    public static bool operator!=(Point a, Point b) => !a.Equals(b);

    // 4. override hash code
    public override int GetHashCode()
    {
          return X * 31 + Y;
    }
}
```

The following guidelines can be used when implementing equality

- Return false if the single argument is null
- Return false if the two objects are of different type
- Compare each field and return false if any fields are different
- Call the base class Equality if it isn't the identity version defined in object
- Implement IEquatable<T> to define a type safe interface
- Overload == and != to internally call the type safe equals method

## Hashing

Why do we do prime messing when creating hashing functions?

## Comparisons

The basic comparison interfaces are `IComparable` and `IComparable<T>`. If comparing two objects for equality, via their equality operators or their Equals method, gives true then comparing them for ordinality should return zero. The converse, however, is not true. We could compare two strings using cases insensitive arguments. In this case comparison is zero and equality is false. Consider the following

```
string a = "hello";
string b = "Hello";

WriteLine(a.Equals(b)); // false
WriteLine(string.Compare(a,b,StringComparison.InvariantCultureIgnoreCase)
); // 0
```

As with equality we can plug in our own algoritms

```
public class Point
{
    public int X;
    public int Y;
}

public class PointComparer : Comparer<Point>
{
    public override int Compare(Point x, Point y)
    {
          if (object.Equals(x,y)) return 0;

          return x.X.CompareTo(y.X);
    }
}
```

Which we use as follows

```
Point[] a = new Point[]
{
    new Point() {X=3,Y=1},
    new Point() {X=1,Y=1},
    new Point() {X=5,Y=1}
};

Array.Sort(a,new PointComparer());
WriteLine(a);
```

## Expressions

The ?? operator returns the left hand operand if it is non-null otherwise it returns the right operand

```
string b = a ?? "empty";
```

The ?. operator will

# User Defined Types

User defined types can be any of the following.

- Class types
- Struct types
- Enum types
- Array types
- Delegate types
- Interface types

Any type defined with file scope can be either public or internal. If no visibility modifier is specfied the default visibility is internal. Any assembly can define other assemblies which it considers friends. Any such friend assemblies can see an assemblies internal types

## Members

- Fields
- Properties
- Methods
- Events
- Constructors
- Deconstructors (C#7)

### Member Visibility

In addition to the following for a member to be visible its owning type must also be visible to the caller.

| Modifier | Visibility |
|---|---|
| **Private** | Accessible by methods in the defining type and in any nested type?? |
| **Protected** | Accessible by methods in defining type, nested type or derived type |
| **Internal** | Accessible by methods in the same assembly |
| **Internal protected** | Accessible by any type in the defining type, nested type, derived type or any method in the defining assembly |

| | |
|---|---|
| **Public** | Accessible to any method in any assembly |

## CONSTRUCTORS

A constructor performs initialization logic on a class or struct. Overloaded constructors can call other constructors using the this construct. Such a construct cannot use the this pointer, however they can call static mathods. If no constructor is explicitly defined the compiler will implicitly generate a parameterless constructor.

> **STRUCTS AND PARAMETERLESS CONSTRUCTORS**
>
> A paramterless constructor is an intrinsic part of a struct which initializes each field with default values. For this reason one cannot create ones own parameterless structor constructor

*Figure 5Constructor overloading*

```
// Two overloaded constructors.
// Second constructor calls the first using this keyword
public MethodCallSemantics() { }
public MethodCallSemantics(int a_) : this() { }

// Two overloaded constructors.
// First constructor calls the second using this keyword
// and passes in an expression that calls a static method
public MethodCallSemantics() : this ( ClassMethod() ) { }
public MethodCallSemantics(int a_) { }

// Two overloaded constructors. Invalid defintions
// First constructor calls the second using this keyword
// and passes in an expression that calls a instnace method
// Wont compile due to use of this pointer inside this expression
// Compiler says Error  7     Keyword 'this' is not available in
// the current context
public MethodCallSemantics() : this ( this.InstanceMethod() ) { }
public MethodCallSemantics(int a_) { }
```

Private constructors are often used in conjuction with static factory methods

```
    // We have a private constructor
    // Only was clients can obtain an instance of the
    // type is through the static method. This gives us
    // control of when and how objects of this type are created
    private ObjectFactory() { }
    public static ObjectFactory CreateObject()
    {
      return new ObjectFactory();
}
```

.NET 3.0 provides object initialization lists for accessible fields or properties

```
public static void MainMethod()
{
  Constructors myObj =
    new Constructors() { Height = 10.0, Width = 2.0 };
}
public double Height { set { } }

public double Width { set { } }

public Constructors() { }
```

### Order of Execution

- From sub-class to base class field are initialized and then constructor args evaluated
- From base class to sub-class constructor bodies are executed

### Static Constructors

- Executes once per type before any instances of the type are created
- Cannot have access modifiers
- Static field initialization occurs before static constructor in order of declaration
- Static constructor is invoked by the runtime and cannot be explicitly invoked
- When runtime invokes static constructors is non-deterministic
- A subclasses static constructor can execute before or after its base class

## PROPERTIES

Look to clients like fields but give the type implementer the control of a method. They promote encapsulation, allow derived field etc. Properties can be read-only when they provide only an accessor. Properties can be write-only when they provide only an mutator (uncommon) Csharp 3.0 allows the specification of automatic properties where the compiler generates implemenation

```
// The compiler will generate a private
// backing field. Set accessor can be
// marked as private if you want a read-only
// behaviour. You cannot just leave out the
// set part
public double AutomaticProperty { get; private set; }
```

The compiler will generate methods called get_XXX and set_XXX for properties. Non-virtual property accessor are inlined by JIT eliminating performance overhead of property versus field

### FIELDS

A field is a member of a class or struct that requires storage. Field initialization occurs before the constructor is invoked in the order of declaration. Any unitialized fields take a default value of zero for numeric values, null for reference types, false for bools. If a field is marked with the readonly modifiers it cannot be modified after an object of that type is constucted. Any fields not explicitly initalized take an implicit value of zero for numeric types, null for reference types and false for bools. Every instance of a struct or class has a special field called this which refeences itself.

### METHODS

A methods signature consists of its name, parameter types but not its return type. Method signatures must be unique within a type. Overloaded methods have the same name but different signatures. Whether a paramater is pass by value or pass by reference forms part of the signature however overloaded methods cannot differ only on ref versus out

```
// Valid as both methods have differnt signatures
// Whether a param is reference forms part of the
// signature
public static void SingleParamMethod(int a) { }
public static void SingleParamMethod(ref int a) { }

// InValid as both methods have same signatures
// cannot define overloaded methods that differ only on ref and out
public static void SingleParamMethod(out int a) { }
public static void SingleParamMethod(ref int a) { }
```

### INDEXERS

Provide access to elements within a classes internal set. The index can be keyed by one or more arguments of any type. A type can have multiple indexers with different argument types

```
public int this[string a, Object b]
 {
   get
   {
     /* Do some lookup based on the composite key {a,b}*/
   }
 }
```

Compiler generate get_Item and set_Item methods

## DECONSTRUCTORS

```
void Main()

{

      Point p = new Point(1.0,2.0);


      var (a,b) = p;

}


public class Point

{

      private double x;

      private double y;


      public Point(double x, double y)

      {

            this.x = x;

            this.y = y;

      }


      public void Deconstruct(out double x, out double y)

      {

            x = this.x;

            y = this.y;

      }
```

```
}
```

## Constants and read-only fields

A consrtant is a field whose value can never change. It is evaluated statically at compile time and substituted wherever used. Only the built in types can be used as constants and they must be initialized with a value.

Readonly fields however are initialiazed at runtime.

To see the different between the two consider and assembly B which exposes two value; one a const and one a static readonly field. Assembly A is then compiled against assembly B. If before assembly A runs assembly B is recomiled with new value for both fields only the static readonly field change will be picked up. This is because the B's constant fields value was burned into A at compile time.

## Classes

User defined types defined using the class keyword are reference types. If a class is defined as static it can only contain static methods and cannot be subclassed. A class can contain fields, properties, methods, events,  constructors and deconstructors

### STATIC CLASSES

- Allow set of related methods to be grouped together in type instantiated type
- E.g. Console, Math, Environment, ThreadPool
- Must directly inherit from object
- Must not implement any interfaces
- Must define only static members
- Class cannot be used as field, method parameter or local variable

## Interfaces

## Enums

```
[Flags]
public enum Location { None = 0, Left = 1, Right = 2, Top = 4, Bottom = 8
}

void Main()
{
    Location vertical = Location.Top | Location.Bottom;
    Location horizontal = Location.Left| Location.Right;
```

```
    // Because we use the Flags attribute this give us "Top, Bottom"
    // Without the Flags enum we would get the integer 12
    WriteLine(vertical);

    // Both output true
    WriteLine((Location.Top & vertical) != 0);
    WriteLine((Location.Bottom & vertical) != 0);

    WriteLine(vertical ^ Location.Top);
    WriteLine(horizontal ^ Location.Left);
}
```

## Nested Types

```
// Classes and structs can have nested classes, structs,
// interfaces, delegates and enums
public class OuterMost
{
    private double x = 4.0;

    // Default visibility is private so not accessible outside
    enum Location { None, Left };

    // Make the innter struct visibile to the outside
    public struct InnerStruct {};
}
```

## Events

Delegates can be used to implement producer consumer behavior. Consider the following producer consumer logic

**Figure 6 Producer Consumer with Delegates**

```
void Main()
{
    Producer producer = new Producer();

    // Register First consumer
    producer.ValueChanged += x => WriteLine($"First {x}");

    // Register Second consumer
    producer.ValueChanged += x => WriteLine($"Second {x}");

    // Producer does something
    producer.Fire();
}


public  delegate void ValueChangedEventHandler(double newValue);

public class Producer
{
    public ValueChangedEventHandler ValueChanged;

    protected void OnValueChanged(double newValue)
    {
          ValueChanged?.Invoke(newValue);
    }

    public void Fire() => OnValueChanged(3.4);
}
```

There are, however, shortcomings with this approach. Firstly consumers can interfere with each other. Secondly consumers can invoke the delegate which is the responsibility of the producer

**Figure 7 Problems with Delegate Approach**

```
    // 1. Second consumer blows away first
    producer.ValueChanged = x => WriteLine($"Second {x}");

    //2. Consumer forcer producer to raise events
    producer.ValueChanged.Invoke(6.7);
```

Events solve these problems with compiler support and a standard pattern. Consider the following code implementation

```
// 1. Create a subclass of EventArgs with a public
//    readonly field for the new value
public class ValueChangedEventArgs : EventArgs
{
     public readonly double NewValue;

     public ValueChangedEventArgs(double newValue)
     {
            NewValue = newValue;
     }
}

public class Producer
{
     // 2. Add an event using the generic EventHandler delegate
     //    type
     public event EventHandler<ValueChangedEventArgs> ValueChanged;

     // 3. Add a method called OnValueChanged that raised the event
     protected void OnValueChanged(ValueChangedEventArgs args)
     {
            ValueChanged?.Invoke(this,args);
     }

     public void Fire(double newValue)
     {
            OnValueChanged(new ValueChangedEventArgs(newValue));
     }
}
```

By using the event member we get language support for separating the producer and consumer behaviour. Consumers can no longer interfere with each other or cause the underlying delegate to fire. We can write our own logic for registering and deregisting with the delegate as follows. The following is an explicity implementation of the previous code

```
public class Producer
{
    private EventHandler<ValueChangedEventArgs> _delegate;

    public event EventHandler<ValueChangedEventArgs> ValueChanged
    {
          add {_delegate += value;}
          remove {_delegate -= value;}
    }

    // 3. Add a method called OnValueChanged that raised the event
    protected void OnValueChanged(ValueChangedEventArgs args)
    {
          _delegate?.Invoke(this,args);
    }

    public void Fire(double newValue)
    {
          OnValueChanged(new ValueChangedEventArgs(newValue));
    }
}
```

While the compiler generates code similar to our explicit add and remove it does so in a thread save manner.

# Generics

## Overview

A generic type declaration has arguments that must be supplied by the user of the type. Generics provide a number of benefits.

#### BENEFITS OF GENERICS

- Increase type safety
- Reduce the number of casts needed
- Reduce the amount of boxing/unboxing required
- Reduces duplicate code
- Enable us to create collections where the element type is parameterised by the user
- Combines the benefit of reusable collection of object with static type safety

### Where we can introduce generic type parameters

```
// Generic class
public class GenericClass<T> { }

// Generic interface
public interface IGenericInterface<T> { }

// Generic struct
public struct GenericStruct<T> { }

// Generic delegate
public delegate TOut GenericDelegate<TIn,TOut>(TIn param);

public class ClassWithGenericMethod
{
    // Generic method
    public void Swap<T>(T a, T b) { }
}
```

Generic type parameters cannot be introduced on constructors, properties, indexers, operators or fields although any of these can use generic type parameters introduced by the enclosing type.

## Open and Closed types

In the below code fragment, I have highlighted the type arguments in bold red. We say the generic type is an **open type**.

**Figure 8Generic Declaration**

```
public class List<T>
{
  public void Add(T element) {_storage[_nextFreeSlot++] = element;}
  public T Get(int idx) {return _storage[idx];}

  private readonly T[] _storage = new T[10];
  private int _nextFreeSlot = 0;
}
```

The user of the generic type specifies the type argument at the point of variable declaration and definition. The generic instance with its type parameter provided is known as a **closed type.** The runtime creates the closed type the first time it sees a request for a particular set of type arguments. The CLR effectively creates a new type.

**Figure 9Generic Use**

```
List<int> listOfInt = new List<int>();
listOfInt.Add(4);
```

## Generic Methods

The method signature contains **type parameters**. We can generically write a swap method to swap two elements of any type, both reference and value types, without requiring casting or boxing/unboxing

```
public static void Swap<T>(ref T a, ref T b)
{
T temp = a;
       a = b;
       b = temp;
}
public void TestSwap()
{
      Object stringA = "Kenny";
       Object stringB = "Wilson";
       Swap<object>(ref stringA, ref stringB);

       int intA = 4;
       int intB = 5;
       Swap<int>(ref intA, ref intB);
}
```

In many cases the compiler can infer the type to be used so we don't even need to provide the type parameter when we use the method. Where there is ambiguity we can provide the compiler with type parameters explicitly.

## Generic Constraints

```
public interface IInterface {}
public class MyClass {}

// 1. Base class constraint. T must be decendent of MyClass
public class BaseClassConstraint<T> where T : MyClass {}

// 2. Interface constraint. T must implement IInterface
public class InterfaceConstraint<T> where T : IInterface {}

// 3. Reference type constraint
public class ReferenceConstraint<T> where T : class {}

// 4. Value type constraint
public class ValueConstraint<T> where T : struct { }

// 5. Parameterless constructor constraint
public class ConstructorConstraint<T> where T : new() { }


public class Naked<T>
{
    // 6. Naked constraint. One generic param must implement
    //    another
    public void NakedMethod<U>(U input) where U : T
    {

    }
```

```
}
```

## Basic conversions

```
public class ConversionExample1
{
    public String Convert<T>(T arg)
    {
          // Does not compile are compiler not sure
          // whether we are performing a custom
          // conversion or not
          if ( arg is String)
                return (String)arg;

          return "Not a string"
    }
}

public class ConversionExample2
{
    public String Convert<T>(T arg)
    {
          // Compiles because the compiler considers this
          // convertion to object and from object to be either
          // boxing or unboxing conversions
          if (arg is String)
                return (string)(Object)arg;

          return "Not a string"
    }
}
```

# Variance

Variance is used to describe how sub-typing in complex type constructions relate to sub typing in the basic polymorphic case. In practice, for C# developers this mean relating subyping in generic interfaces and delegates to subtyping in basic polymorphic situations. This is because in .NET variant type parameters are restricted to generic interfaces and delegate types. Variance is also only applicable to reference types.

## COVARIANCE

Covariance enables one to asign an object whose generic type parameter is of a more specific type to a variable whose generic type parameter is of a more generic type. This looks much like polymorphism and feels intuitive.

```
IEnumerable<Cat> cats = new List<Cat>();

IEnumerable<Animal> animals = cats;
```

In order to allow this the IEnumerable interface specifies it type paramer as `out`

```
public interface IEnumerable<out T> : IEnumerable
```

we can do the same thing with delegates,

```
Func<Cat> f = () => new Cat();

Func<Animal> g = f;
```

## CONTRAVARIANCE

Contravariance enables one to asign an object whose generic type parameter is of a more generic type to a variable whose generic type parameter is of a more specific type. Contravariance looks counterintuitive from the perspecive of basic polymorhism

In order to allow this the IObserver interface is markd as `in`

```
public interface IObserver<in T>
```

We can do the same thing with delegates.

```
void Main()
{
    ContravariantDelegate<object> a = DoSomething;

    // Looks a bit strange but it type safe
    ContravariantDelegate<string> b = a;
    a("A String");
}

public void DoSomething(Object obj) {}
```

```
delegate void ContravariantDelegate<in TIn>(TIn inputType);
```

### HIGHER ORDER FUNCTIONS

The previous examples on covariance and contravariance were with first order functions. Higher order functions are functions that either take functions as arguments or return functions as results. Higher order function add complexity, especially where contravariance is involved. Consider the following second order function

```
void SecondOrderFunction(Action<Derivative> derivative)

{

  derivative(new Derivative());

}
```

As it invokes the passed in delegate with an instance of Derivative it cannot handle functions that require a subclass of Derivative in a type safe manner. However, it can handle base classes. Contravariance has been reversed in the second order function.

```
Action<Action<Product>> g = SecondOrderFunction;

g(Console.WriteLine);
```

The basic rule is an even number of contravariance conversions will cause the final result to be covariant and an odd number will cause the resut to be contravariant.

One classic example of a higher order function and contravariance become covariance is with the `IObervable<T>` interface from Reactive Extensions.

```
  public interface IObservable<out T>

  {

    IDisposable Subscribe(IObserver<T> observer);

  }
```

Superficially we might expect the type parameter to be marked as in because it is used with a method parameter. Because the parameter is itself a generic type the contravariance is converted to variance. We can use it like this.

```
IObservable<Derivative> o1 = new Subject<Derivative>();

IObservable<Product> o2 = o1;
```

Any observers of products will be quite happy to receive derivatives.

# Questions

## Overview of Type

**In .NET every data item has a type. What does that type define?**

*How much memory needs to be set aside for that item*

*Set of all possible values an instance of that type can take*

*The operations that can be carried out on that type*

**What is meant by statically typed?**

*All variables are of a particular type which the compiler is aware of. The compiler will only allow operations relevant to a variables type*

# CATEGORISING TYPE

## VALUE TYPES

**What is a value type?**

*Any type that inherits from System.ValueType*

**What is a valuetype also known as?**

*A struct*

**What are the characteristics of a ValueType?**

1. *When used as local variable or formal parameter its fields are stored directly on the stack*
2. *No pointer dereference needed to access its fields*
3. *Assignment always results in a field by field copy*
4. *Implicitly sealed*

**When would you chose to use a value type?**

5. *Behaves like a primitive and is immutable*
6. *Doesn't need to inherit from another type*
7. *Wont need to be extended*
8. *Wont be passed as value parameter to functions*

**What are the performance advantages of value types?**

*No pointer derefernce needed to access fields*

*Stack allocations reduce the number of garbage collections*

*Value type methods can be invoked non-virtually*

**Can value types be sub-classed?**

*No they are implicity sealed*

**How much memory does a value type occupy?**

*The sum of the memory occupied by its fields*

**What is the default implementation of hash-code and equality for value types?**

*Equality if all instance fields match*

*Hashing algorithm based on values in instance fields*

**Why should you override Equality when implementing value types?**

*Default implementation uses reflection which is inefficient*

**What happens if you call a method defined in Object and not overriden in ValueType?**

*The value type is boxed in order to call the method expecting the value type*

**What happens if you call a method defined in a ValueType or you subclass?**

*As value types cannot be subclassed the compiler will use the call rather than callvirt instruction*

**What is the default implementatin of hash code and equality for value types?**

*Equality if all fields are binary equal*

*Hashing based on the value in instance fields*

**Can you use a value type as a lock? Explain your answer**

*No – Value types do not contain the sync block and cannot be used for locking*

*What are predefined types?*

*Types supported by the compiler*

### Reference Types

**What is a reference type?**

*Any type that does no inherit from System.ValueType*

**Where are reference types allocated?**

*Always on the heap*

**Which types in C# are reference types?**

*All class, array, delegate and instance types*

**What memory is required for an instance of a reference type?**

*Heap memory for all instance fields plus extra memory for lock, class pointer*

*Each reference variable on stack need additional memory*

**How much memory is required for a reference variable on the heap?**

*Depends on whether the operating system is 32bit or 64 bit*

*32bit OS requires four bytes, 64 bit OS requires 8 bytes*

**What are the characteristics of reference types?**

*Allocated from managed heap using new*

*Types derived from System.Object type using the class keyword*

**What are the main disadvantages of reference types?**

*Memory must be allocated off the managed heap which can force GC*

*Pointer dereference cause performance overhead*

**How much memory does a reference type occupy?**

*Sum of memory occupied by its fields plus*

*Extra overhead for key to objects type, lock state*

*Each reference requires an additional four to eight bytes*

*What are the base classes of value and reference types?*

*Value types extend ValueType using the struct keyword*

*Reference type extend Object using the class keyword*

### Comparing Reference and value Types Types

**Where is memory allocated for reference and value type?**

*Reference types are always allocated on the heap*

*Value types are usually allocated on the stack*

**Compare and contract assignment of reference and value types?**

*Assignment of value type results in copy of fields*

*Assignment of reference type simply results in copy of address of object on heap*

### Categorising Type – Predefined and Primitive Types

**What is meant by a predefined type?**

*A type supported by the compiler*

**What are predefined types also known as?**

*Built in types*

**Are predefined types reference or value types?**

*Can be either*

**Give another name for predefined types**

*Built in types*

**What are the predefined value types?**

*The numeric types plus bool and char*

**What are the predefined reference types?**

*String and decimal*

**What are primitive types?**

*All the predefined value types other than decimal. We expect primitive type to be directly supported by the undelying native instruction set and CPU*

## Functions

**Which language features are implemented as functions**

*Constructors*

*Properties*

*Methods*

*Events*

*Indexers*

*Finalizers*

**Is there any point to the initialization of a in the below?**

```
void Main()
{
	int a = 4;
	ProvideOut2(out a);
}

public void ProvideOut2(out int x)
{
	x = 3;
}
```

*No. Because the compiler requires that out variables are initialized inside a method before they are used.*

**What happens when one tries to compile the below code?**

```
void Main()
{
	int x;
	DoRef(ref x);
}

public void DoRef(ref int a)
{
	Console.WriteLine(a);
	a = 4;
}
```

*The compiler gives an error complaining about use of unassigned local variable x*

**Compare out and ref formal parameters?**

*Using out the actual parameters do not need to be initialized before the method call and the method cannot access them until they are initialized. With ref the actual parameters must be inititialzed before being passed to the method and the method can access them without initializing them.*

**What are reference variables?**

*Csharp7 feature enabling one variable to be synonym for another. Chaning one variable changes both.*

**What is the default means of parameter passing?**

*Pass by value. References are copied*

**What are reference return values?**

*C#7 feature enabling values to return by reference*

**Are there any restriction on reference return values?**

*The method has to return a reference to something whose scope outlives that method, typically a field on a class or a variable passed as an argument to the function. The functions local varaiables cannot be returned by value.*

**When would a compiler generate a call instruction?**

*Methods defined on a value type (implicitly sealed so no inheritance)*

**When would a compiler generate a non-virtual call of a virtual method and why?**

*When a method call base.method the CLR will call it non-virtually to prevent an infinite recursion*

**How can one specifiy pass by reference semantics?**

*By using the out or ref keywords*

**How do ref and out keywords differ?**

*Ref parameters must be initialized by the caller*

*Out parameters must be initialized by the callee*

**Are the ref and out parameters specified by the function or caller?**

*For clarity both*

### DISPATCH

**What are the C# call instructions?**

*Call and callvirt*

**Compare the two?**

*Methods executing virtually look at the method table of the object on which the method is dispatching in order to determine the method table slot (address) to use (runtime).*

*Non-virtual methods enable the JIT to burn the address of the target method table address because it knows the location at compile time*

**When is call used?**

*To call static and non-virtual instance methods*

*Sometimes even to call virtual methods non-virtually*

*Calling value type methods which are implicitly sealed*

**When is call used to invoke virtual methods non-virtually?**

*When invoking base.method to prevent stack overflow*

**Describe callvirt?**

*Used to invoked methods polymorphically.*

**In .NET what is the default dispatch mechanism?**

*By default dispatch is non-virtual*

**What is the output of the following code and why?**

```
void Main()
{
    C c = new C();
    B b = c;
    A a = b;

    c.DoSomething();
    b.DoSomething();
    a.DoSomething();
}

public class A
{
    public void DoSomething() => Console.WriteLine("A:DoSomething");
}

public class B : A
{
    public void DoSomething() => Console.WriteLine("B:DoSomething");
}

public class C : B {}
```

*B:DoSomething*

*B:DoSomething*

*A:DoSomething*

*Dispatch is non-virtual by default so the method invoked is based on the compile time time of the variable.*

**What is the output of the following code and why?**

```
void Main()
{
    SubClass c = new SubClass();
    BaseClass bref = c;

    // As the method is defined as virtual in the base class
    // and overriden in the subclass all calls are polymorphic.
    // The run-time type of the actual object, and not the
    // compile-time type of the reference, determines which method
    // is called

    bref.VirtualMethod();
}

class BaseClass
{
    public BaseClass()
    {
          Console.WriteLine("BaseClass()");
    }

    public virtual void VirtualMethod()
    {
          Console.WriteLine("BaseClass.VirtualMethod()");
    }
}

class SubClass : BaseClass
{
    public SubClass() { Console.WriteLine("SubClass()"); }


    public override void VirtualMethod()
    {
          Console.WriteLine("SubClass.VirtualMethod()");
    }
}
```

*SubClass.VirtualMethod()*

*The method in the base class is defined as virtual and the subclass overrides it using the override keyword so dispatch is virtual using the runtime type of the object item on the heap.*

**What is the output of the following code and why?**

```
void Main()
{
    SubClass c = new SubClass();
    BaseClass bref = c;

    // Even though the base class method is virtual, because we
    // didnt override it and instead used the new keyword, once
    // again the method invoked is based on the compile time type

    // Call SubClass.VirtualMethod
    c.VirtualMethod();

    // Call BaseClass.VirtualMethod
    bref.VirtualMethod();
}

class BaseClass
{
    public BaseClass()
    {
          Console.WriteLine("BaseClass()");
    }

    public virtual void VirtualMethod()
    {
          Console.WriteLine("BaseClass.VirtualMethod()");
    }
}

class SubClass : BaseClass
{
    public SubClass() { Console.WriteLine("SubClass()"); }


    public new void VirtualMethod()
    {
          Console.WriteLine("SubClass.VirtualMethod()");
    }
}
```

*SubClass.VirtualMethod()*

*BaseClass.VirtualMethod()*

*Even if the base class method is virtual when we use the new keyword in the subclass we hide the base implementation and prevent polymorphism taking place. The type of the method invoked is then again determined by compile time type of reference.*

**What is the output of the following code and why?**

```
void Main()
{
    Base b = new Sub();
    DoIt(b);
}

public void DoIt(Base b) => WriteLine("DoIt(Base)");
public void DoIt(Sub b) => WriteLine("DoIt(Sub)");

public class Base { }
public class Sub : Base { }
```

*"DoIt(Base)"*

*Which overloaded method to call is evaluated statically using the compile type of the variable not the runtime type of the item.*

## DELEGATES

**What is a delegate?**

*An object that encapsulates a method call*

*Self-describing type-safe function pointer*

**What do all delegates in C# contain?**

*Target object which is null if method is static*

*Method Pointer*

*Previous pointer allowing delegates to be chained*

**Which design pattern do C# delegates implement?**

*Observer pattern*

**What is the observer pattern also known as?**

*Also known as publish/subscribe*

*Notifications of single events broadcast to multiple subscribers*

**What are the main advantages of delegates?**

*Provide a level of indirection between caller and method implementation*

*Dynamically wire up method caller and target method*

*Delegate type provides a protocol to which the caller and targer conform to*

*Delegate instances refer to one or more target methods conforming to the protocol*

- Caller invokes a delegate and the delegate inokes the method
- Similar to C function pointers

**What are the uses of a delegate?**

*Passing in comparison method argument to a sort algorithm*

*Enabling asynchronous notification*

**What is MulticastDelegate?**

*Forms base class for all delegates in C#*

*Each delegate has three properties*

**How do the MulticastDelegate type check for equality?**

*Examines the target object and the method pointer fields*

**What happens if you try to explicitly inherit from MulticastDelegate?**

*Compiler will generate an error*

**What will happen when a compiler compiles the following code?**

```
public delegate void strdel( string str );
```

*A new class that derives from MulticastDelegate will be created*

*Implementation of the Observer pattern*

**What is the effect of the += operator on a delegate?**

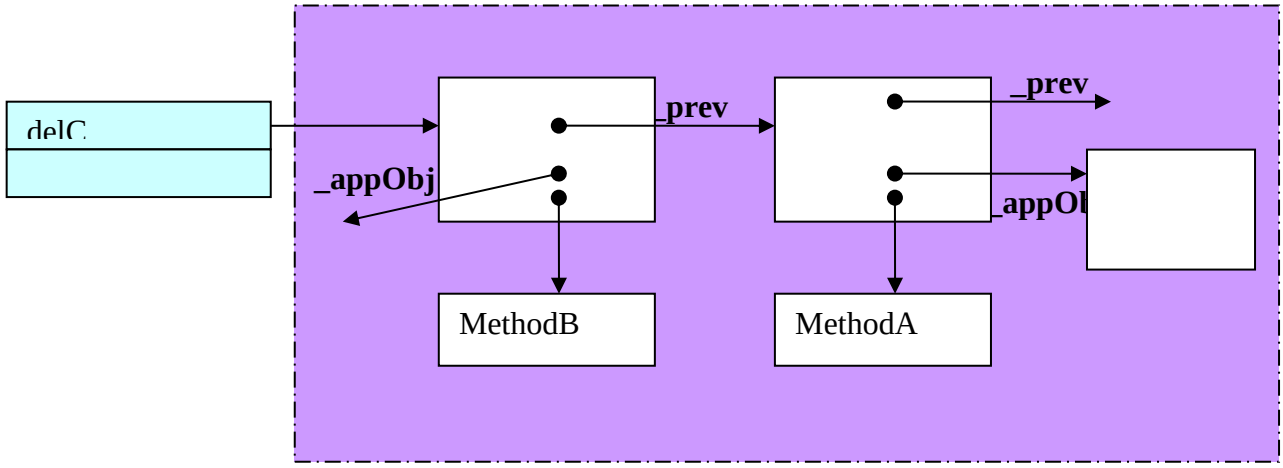
*The same as the static combine method on the delegate*

*Creates a chain with the second argument as the head*

*Invoke calls previous invoke first so delegates generally invoked in order added*

**What will happen when the delegate delC is invoked?**

```
strdel delB = new strdel(SomeClass.SomeStatic);
strdel delC = delA += delB;
```




---

- *When delegate a is invoked it will first check to see if its previous pointer is set and if it is then it will call that first so that methods are invoked in the order they are added.*

**What is the effect of the -= operator on a delegate?**

*Remove (-=)*

*removes a delegate from the chain and returns head*

**How do events differ from delegates?**

*Events are members of a type*

*Events are implemented using delegates*

*Only the containing type can invoke the delegate and trigger event notification*

*Assignment not supported outside of containing class*

**By convention types which provide event information are derived from?**

*System.EventArgs*

**What else is needed?**

- *A method responsible for notifying registered listeners of the event*

**Define an event?**

```
public delegate void strdel(string str);

public class SomeClass
{
    public event strdel MyEvent;
}
```

**Show what the compiler will generate behind the scenes????**

```
public class SomeClass2
{
    public strdel MyEvent;

    public void add_MyEvent(strdel del) =>
          MyEvent = (strdel)Delegate.Combine(MyEvent, del);

    public void remove_MyEvent(strdel del) =>
          MyEvent = (strdel)Delegate.Remove(MyEvent, del);
}
```

**What else is needed?**

*A method resposible for notifying registed listeners*

**Write such code?**

```
{

    strdel delA = new strdel(this.SomeInstanceMethod);

    Delegate[] dels = delA.GetInvocationList();


    foreach (strdel del in dels)

    {

      try

      {

        del("Test");
```

```
        }
        catch (Exception ex) { }
    }
}
```

## CLOSURES

**What is the output of the following code and why?**

```
Action[] actions = new Action[3];

for(int i = 0; i < 3; i++)
{
      actions[i] = () => Console.Write(i);
}

foreach (var action in actions)
{
      action();
}
```

*333 as it is the variable itself that is captures and not the value of it at the time the action is created.*

**What is the output from the following code and why?**

```
Action[] actions = new Action[2];

int outer = 0;
for (int j = 0; j<2;j++)
{
      int inner = 0;
      actions[j] = () => Console.WriteLine($"{outer++},{inner++}");
}

actions[0]();
actions[0]();
actions[0]();

actions[1]();
```

*0,0*

*1,1*

*2,2*

*3,0*

*4,1*

*The reason being that both delegates share a reference to the heap stored variable outer whereas each delegate has its own heap stored variable to hold the captured variable inner.*

**What is the output of the following code and why?**

```
void Main()
{
    Action a = GetAction();
    a();
    a();
}

public static Action GetAction()
{
    int i = 5;
    return () => Console.WriteLine(++i);
}
```

*6*

*7*

*The captured variables are stored on the heap and get extended scope. They stay alive until all delegated holding references to them are collected*

**What is the point of captured variables?**

*They prevent the need to write custom classes jus to pass in information from the calling scope into a delegate.*

## LOCAL METHODS

*What are the advantages of local methods over lambda?*

*Don't require the creation of a delegate type and instance*

*Support recursion*

*Don't require same indirection as delegate so more efficient*

*Can access variables of containing method more efficiently*

## Class Hierarchies and Function Call

**What is meant by polymorphism?**

*A reference to a base class can refer to an instance of a subclass*

**Define an interface with one field and one method?**

```
interface Interface1
{
  string Property { get; }

  string Method();
}
```

**Create an abstract implementation that implements one of the fields?**

```
abstract class Abstract : Interface1
  {
    private string _property ;

    public Abstract(string property_)
    {
      _property = property_;
    }

    public string Property
    {
      get { return _property; }
    }

    public abstract  string Method();
  }
```

**Now create a concrete subclass of the abstract class**

```
class Concrete : Abstract
{
  public Concrete(string aString)
    : base(aString)
  {
  }

  public override string Method()
  {
    return null;
  }
}
```

**Finally add a second constructor that take two arguments and calls the existing constructor?**

```
class Concrete : Abstract
{
  public Concrete(string aString)
    : base(aString)
  {
  }

  public override string Method()
  {
    return null;
  }
}
```

**What is output from the following code?**

```
public class BaseClass
  {
    public virtual void Foo() { Console.WriteLine("BaseClass.Foo"); }
  }

  public class Overrider : BaseClass
  {
    public override void Foo() { Console.WriteLine("Overrider.Foo"); }
  }

  public class Hider : Overrider
  {
    public new void Foo() { Console.WriteLine("Hider.Foo"); }
  }

  public class MainClass
  {
    public static void MainMethod()
    {
      Overrider over = new Overrider();
      BaseClass b1 = over;

      over.Foo();
      b1.Foo();

      Hider h = new Hider();
      BaseClass b2 = h;
      h.Foo();
      b2.Foo();
    }
  }
```

**What are the two uses of the base keyword?**

*Accessing a overridden function member from the subclass*

*Calling a base class constructor*

**What happens if a sub-class constructor omits the base keyword?**

*The base types parameterless constructor will be called*

**What happens if the base class doesn't have a parameterless constructor?**

*The subclass constructor is forced use the base keyword explicitly to choose a base constructor*

**What are object initializers?**

*Accessible fields or properties of an object can be initialized in single statement directly after construction*

*C# 3.0 Feature*

```
ObjectInitializers ob = new ObjectInitializers {
SomeField = "Kenn" };
```

**Label each line in the following code with the order in which it is executed?**

```
  public class BaseClass
  {
    public int x = 0;

    public BaseClass()
    {
      int a = x;
    }
  }

  public class SubClass
  {
    public int y = 1.0;

    public SubClass()
    {
      int z = y;
    }
 }
The answer is

  public class BaseClass
  {
    public int x = 0;  // Executes third

    public BaseClass()  // Executes fourth
    {
      int a = x;        // Executes Fifth
    }
  }

  public class SubClass
  {
    public int y = 1.0; // Executes first

    public SubClass()   // Executes second
    {
      int z = y;        // Executes Sixth
    }
  }
```

## Predefined Types

### Char

**Describe the difference between a character set and an encode?**

*A character set maps numeric values to alphanumeric characters*

*An encoding defines how the numeric codes for each character are converted to binary.*

**What encode does .NET use at runtime?**

*UTF-16 which allocates 16 bits to each character*

*Covers all points in the Multilingual plane subset of Unicode*

**What code is used for .NET streaming ?**

*UTF-8 which is the default for .NET streaming*

### Nullable Types

**What are nullable types?**

*lightweight value types with logic to support being considered null*

**How does one declare an instance of a nullable type?**

*Using* `T? syntax`

**What does the compile map T? to**

*To the type* `System.Nullable<T>`

**What happens when one boxes a nullable type?**

*The compiler is clever and boxes the actual value and not the nullable type as boxed types are already reference types and hence can be null*

### Anonymous Types

**ToDo: Add questions on anonymous types**

### Tuples

**ToDo: Add questions on tuples**

## Numeric Types

**What size is an int in Csharp?**

*32 bits*

**What is the range of int?**

*-2^31 to 2^(31)-1*

**What other integral types are there?**

*Sbyte, byte, short, ushort, uint, long, ulong*

**What size is a float inCsharp?**

*32 bits*

**What size is a double in csharp?**

*64 bits*

**How does the compiler infer the type of a numeric literal?**

*If it containes . or E it is assumed to be a double*

*Otherwise first int in the list int,uint, long, ulong that can accommodate the literal*

**What suffixes can be applied to a literal?**

*L,F,F,U,M*

**What is the M suffix?**

*decimal*

## NUMERIC TYPE CONVERSION

**When is an implicit cast allowed?**

*No possible loss of data*

**When is an explicit cast needed?**

*Possible loss of date*

**When is default overflow behaviour?**

*No overflow checking with wrapping*

**How can overflow checking be turned on?**

*Use of checked keyword on block or expression*

*Compiler flag*

**How can overflow checking be turned off if the compiler turned it on?**

*Using the unchecked keyword around a block or expression*

**What happens when you cast from a floating point to integral type?**

*The fractional part is truncated*

**How can you obtain rounding behaviour when converting?**

*The static class Convert provides methods for rounding*

**What happens when you implicitly convert a large integer to a floating point?**

*Keep the magnitude but loose the precision*

**What happened when you use the division operator on integers?**

*Remainder is always truncated*

**What happens if arithmetic operations on integral types overflow?**

*CLR guaranteed wraparound*

**Why wont the following code compile?**

```
static void Add()
{
  short a = 4;
  short b = 8;

  short c = a + b;
}
```

*Because the 8 and 16 bit integral types lack their own arithmetic operators*

*C# will implicitly convert them to int causing an error assigning result back to short*

## ARITHMETIC OPERATORS

**Implement a method to compare two numbers for equality without using any equality operators?**

```
public static bool Equals(int a, int b)
{
  return (!(a < b) && !(a > b));
}
```

**Nice. Do it now without use of the comparison operators?**

```
public static bool Equals2(int a, int b)
{
  int c = a - b;
  bool equals = true;

  try
  {
    object[] arr = new object[1];
    object res = arr[c];
  }
  catch (Exception ex)
  {
    equals = false;
  }
  return equals;
}
```

**Imagine truth is defined in the same way as C. Implement the comparison method using bitwise operators?**

```
public static bool Equals2(int a, int b)
{
  return (!(a ^ b));
}
```

**Implement a comparison operator for two integers?**

**How are floats and doubles internally represented?**

*As base 2*

**What problems can this introduce?**

*Only numbers expressible in base 2 are represented precisely*

*Literals with a fractional component will not be represented precisely*

## User Defined Types

**What visibility modifiers are applicable to a file scoped type?**

*Public or internal*

**What is the default visibility if no explicit type modifier is provided?**

*Internal*

**What is the purpose of a static class?**

*Allow related methods to be grouped together*

**What restrictions apply to static classes?**

*Must directly implement object*

*Cant implement interfacces*

*Must define only static members*

*Cannt be used as field, method parameter or local variable*

**What visibility modifiers can be applied to a member?**

*Private – only accessed by methods in the defining type or nested type*

*Protected – only accessed by methods in defining type, subtype or nested type*

*Internal – can only be accessed by methods in defiining assembly*

*Internal protected – can be accessed by defining type, nested type, sub-type or any method in defining assembly*

*Public – accessible to any method in any assembly*

**Describe sealed classed and methods**

- An overridden function member can seal its implementation preventing further sub-classing
- A sealed member can be implemented non-polymorphically in a subclass using new
- A class can be sealed preventing all sub-classing

## Type Conversion and Casting

### Numeric Conversion and Arithmetic Operators

**In general, when is implicit conversion allowed for numeric types?**

*When there will be no loss of data*

**Describe the conversion from integral types to floating point types?**

*Implicit*

*Magnitude maintained but precision may be lost*

**Describe the conversion from floating point types to integral types?**

*Cast needed*

*Fractional part truncated*

*Use System.Convert for rounding*

### Boxing

**What happens if you cast your value type to an interface it implemements?**

*It will be boxed*

**What is boxing?**

*Allocating memory from the heap for the value type fields plus a type pointer and a sync block*

*Copy the value types fields into the heap*

*Return a refernce*

**What is the advantage of boxing?**

*Allows a unified view of the type system*

**What is entailed by boxing?**

*Allocating enough heap memory for a value type plus object overhead*

*Value type bits are copied onto the heap*

*The address of the reference type is returned*

**What is entailed by un-boxing?**

*A pointer to the value type contained inside the object is returned*

*The value type that this pointer refers to does not include the usual overhead associated with a true object: a pointer to a virtual method table and a sync block*

**When should you manually box value types?**

*If you know that the code you're writing is going to cause the compiler to generate a lot of boxing code, you will get smaller and faster code if you manually box value types*

**When should you be very careful of boxing?**

*Boxing and un-boxing in a loop can seriously degrade performance and memory usage*

**Why is boxing less prevalent since .NET 3.0?**

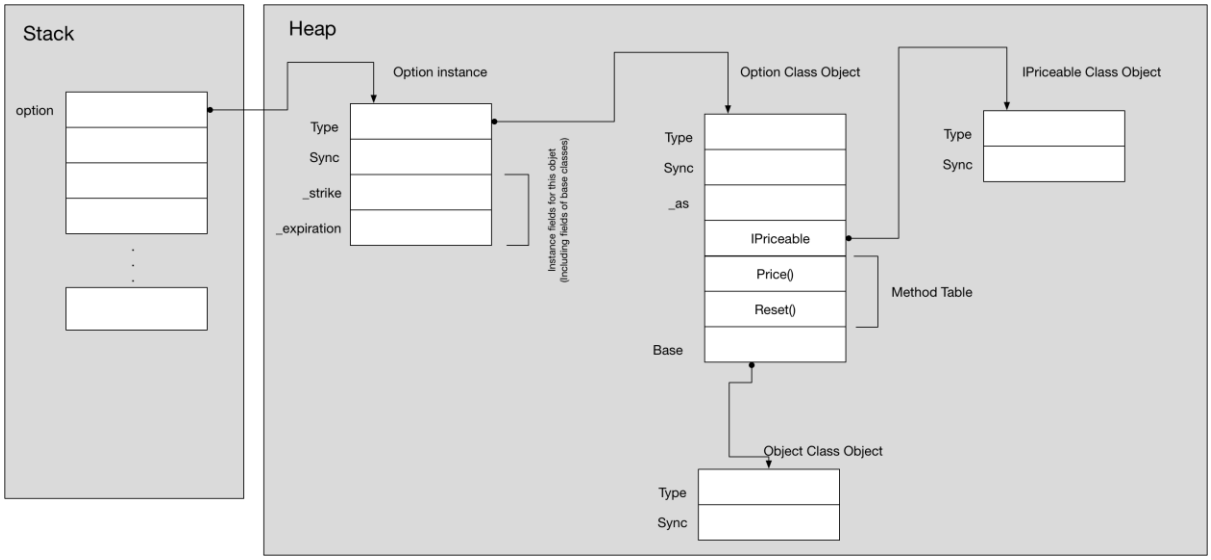
*The introduction of generics makes boxing less necessary*

**What do you need to ensure when unboxing?**

*You need to ensure the type you un-box to is exactly same type as the original value type*

## EQUALITY

## CONTRAVARIANCE


Stack
option
Heap
Option instance
Type
Sync
_strike
_expiration
Instance fields for this objet
(Including fields of base classes)
Option Class Object
Type
Sync
_as
IPriceable
Price()
Reset()
Base
Method Table
IPriceable Class Object
Type
Sync
Object Class Object
Type
Sync