Module 2

Types basics

Value vs reference type

"value types are allocated on the stack and reference types are allocated on the heap"

"value types are allocated on the stack and reference types are allocated on the heap"

"value types are allocated on the stack and reference types are allocated on the heap"

"value types are allocated on the stack and reference types are allocated on the heap"

- this IS NOT true

"value types are allocated on the stack and reference types are allocated on the heap"

- this IS NOT true
- this IS NOT even a good simplification

"value types are allocated on the stack and reference types are allocated on the heap"

- this IS NOT true
- this IS NOT even a good simplification
- this IS an implementation detail:

"value types are allocated on the stack and reference types are allocated on the heap"

- this IS NOT true
- this IS NOT even a good simplification
- this IS an implementation detail:
  - stack vs heap - describes the concept from **the implementation** point of view

"value types are allocated on the stack and reference types are allocated on the heap"

- this IS NOT true
- this IS NOT even a good simplification
- this IS an implementation detail:
  - stack vs heap - describes the concept from **the implementation** point of view
  - does NOT explain the true difference behind those two categories

    (although it may be useful in memory-aware code, as we will see...)

"value types are allocated on the stack and reference types are allocated on the heap"

Value type vs reference type

ECMA-335:

Value type vs reference type

ECMA-335:

- *"type, value: A type such that an instance of it **directly contains all its data**."*

Value type vs reference type

ECMA-335:

- *"type, value: A type such that an instance of it **directly contains all its data**."* (...) *"The values described by a value type are **self-contained** (each can be understood without reference to other values)."*
- *"type, reference: A type such that an instance of it **contains a reference to its data**."*

Value type vs reference type

ECMA-335:

- *"type, value: A type such that an instance of it **directly contains all its data**."* (...) *"The values described by a value type are **self-contained** (each can be understood without reference to other values)."*
- *"type, reference: A type such that an instance of it **contains a reference to its data**."* (...) *"A value described by a reference type **denotes the location of another value**."*

Value type vs reference type

ECMA-335:

- *"type, value: A type such that an instance of it **directly contains all its data**."* (...) *"The values described by a value type are **self-contained** (each can be understood without reference to other values)."*
- *"type, reference: A type such that an instance of it **contains a reference to its data**."* (...) *"A value described by a reference type **denotes the location of another value**."*

So, we can say the true difference is the instance "shape" and bahaviour:

	Shape	Sharing	Lifetime	Identity
Value type	Only data	Pass-by-value - cannot be shared (we can only send **a copy** of it*)	As long as the instance itself	Does not have (instances are equal if all bits are equal**)
Reference type	Reference and data	Pass-by-reference - can be shared (many references to the same data)	Needs to be discovered (multiple instances to shared data)	Location-based (the same if and only if their locations are the same**)

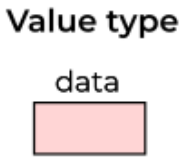
* although you can use so-called *managed pointer* to pass a pointer to it, ** although you can override it in user-defined types

Value type vs reference type

```
A(x)
{
    var y = x;
    return y;
}
```

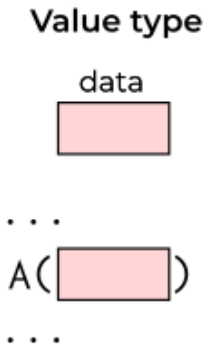
Value type vs reference type

```
A(x)
{
    var y = x;
    return y;
}
```



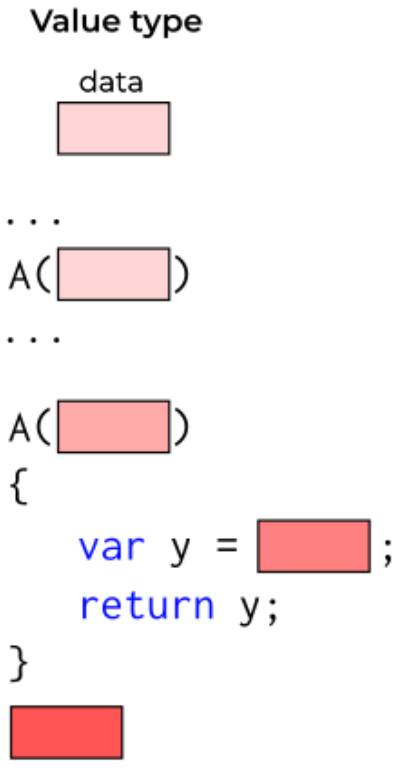
Value type vs reference type

```
A(x)
{
    var y = x;
    return y;
}
```



Value type vs reference type

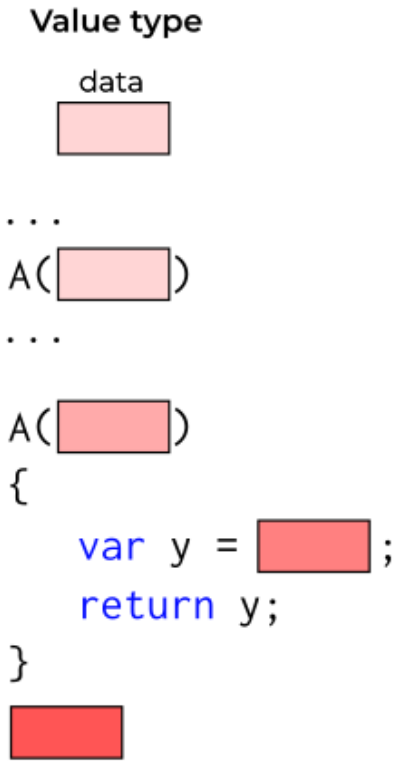
```
A(x)
{
    var y = x;
    return y;
}
```



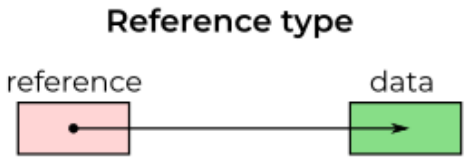
pass-by-value semantics

Value type vs reference type

```
A(x)
{
    var y = x;
    return y;
}
```

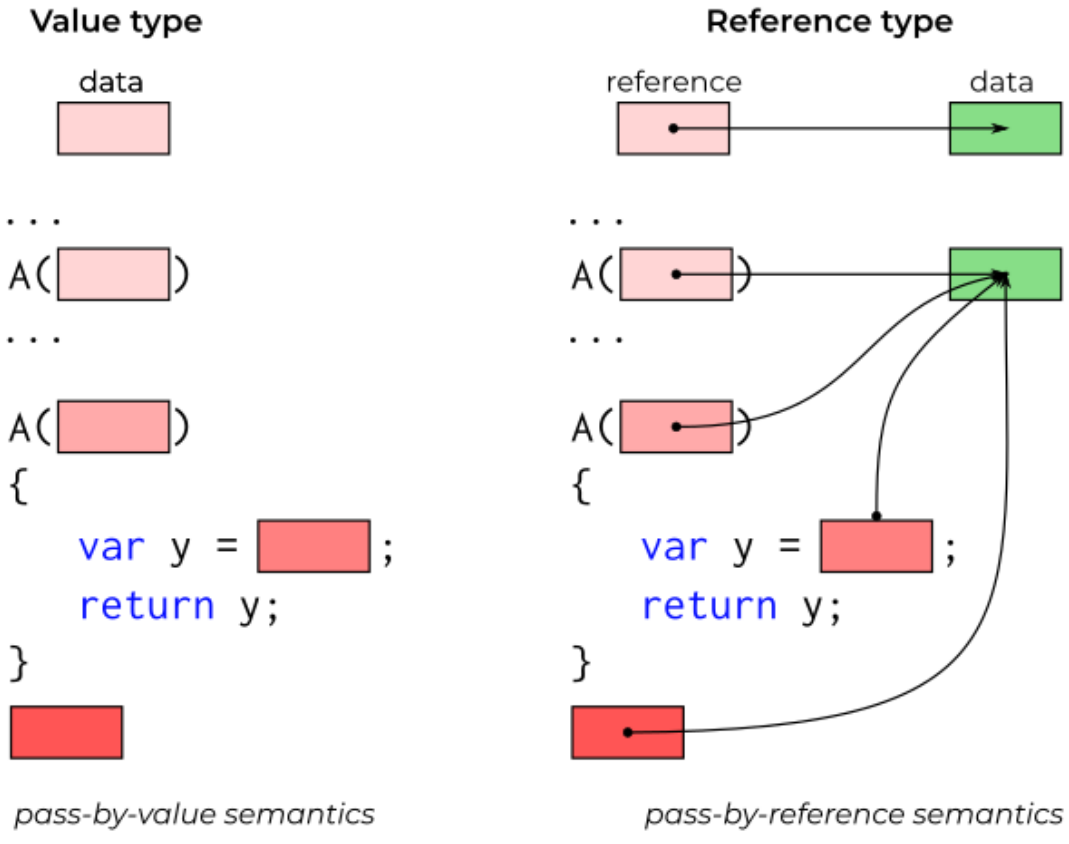


pass-by-value semantics



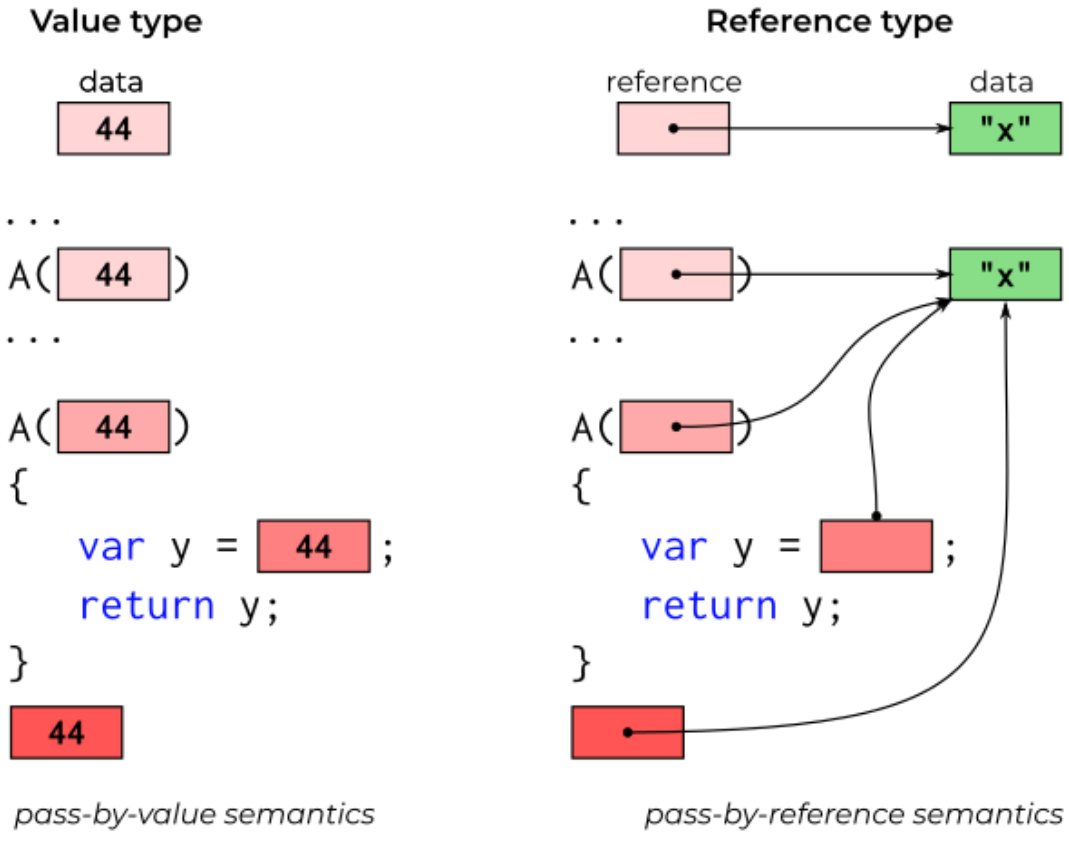
Value type vs reference type

```
A(x)
{
    var y = x;
    return y;
}
```



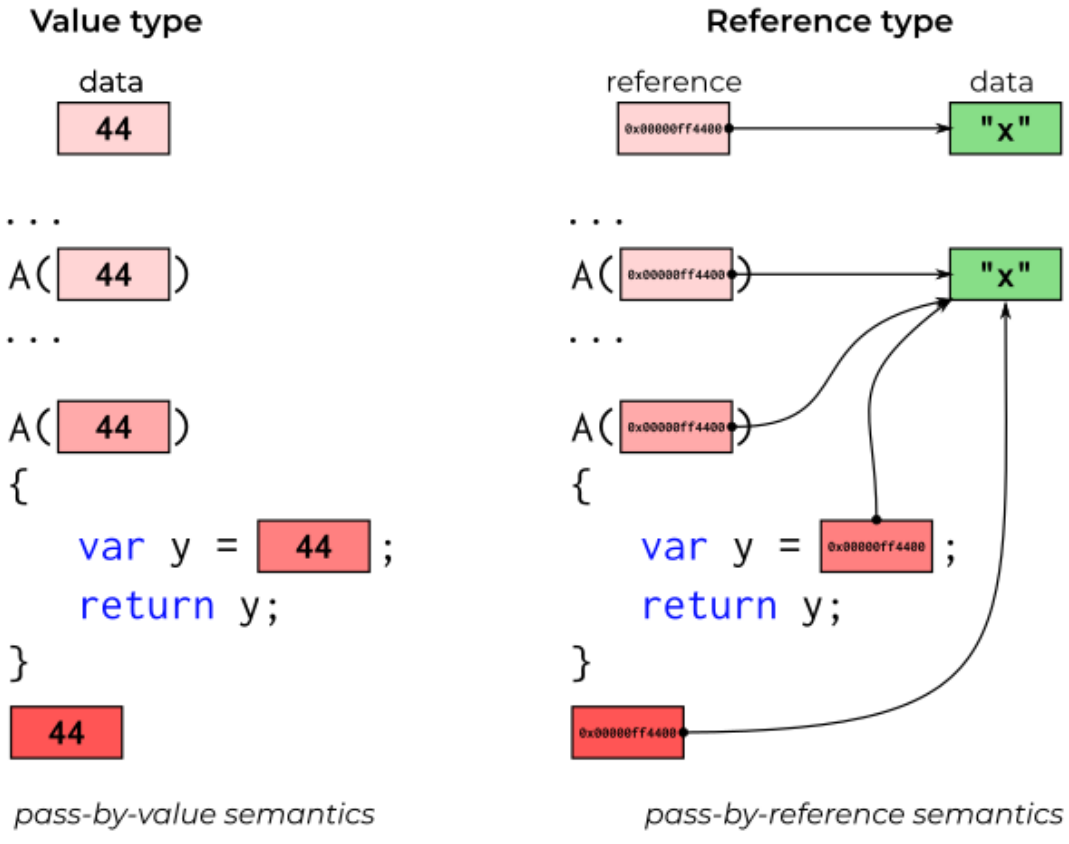
Value type vs reference type

```
A(x)
{
    var y = x;
    return y;
}
```



Value type vs reference type

```
A(x)
{
    var y = x;
    return y;
}
```



Value type vs reference type

structs are value types (having *pass-by-value* semantics):

```
var x = new SomeStruct() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeStruct M(SomeStruct s)
{
    s.Field++;
    return s;
}

public struct SomeStruct
{
    public int Field;
}
```

*class*es are reference types (having *pass-by-reference* semantics):

```
var x = new SomeClass() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeClass M(SomeClass s)
{
    s.Field++;
    return s;
}

public class SomeClass
{
    public int Field;
}
```

Value type vs reference type

structs are value types (having *pass-by-value* semantics):

```
var x = new SomeStruct() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeStruct M(SomeStruct s)
{
    s.Field++;
    return s;
}

public struct SomeStruct
{
    public int Field;
}
```

*class*es are reference types (having *pass-by-reference* semantics):

```
var x = new SomeClass() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeClass M(SomeClass s)
{
    s.Field++;
    return s;
}

public class SomeClass
{
    public int Field;
}
```

What is the result?!

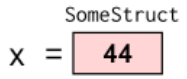
Value type vs reference type

structs are value types (having *pass-by-value* semantics):

```
var x = new SomeStruct() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeStruct M(SomeStruct s)
{
    s.Field++;
    return s;
}

public struct SomeStruct
{
    public int Field;
}
```



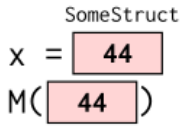
Value type vs reference type

structs are value types (having *pass-by-value* semantics):

```
var x = new SomeStruct() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeStruct M(SomeStruct s)
{
    s.Field++;
    return s;
}

public struct SomeStruct
{
    public int Field;
}
```



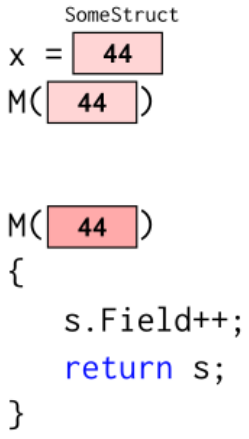
Value type vs reference type

structs are value types (having *pass-by-value* semantics):

```
var x = new SomeStruct() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeStruct M(SomeStruct s)
{
    s.Field++;
    return s;
}

public struct SomeStruct
{
    public int Field;
}
```



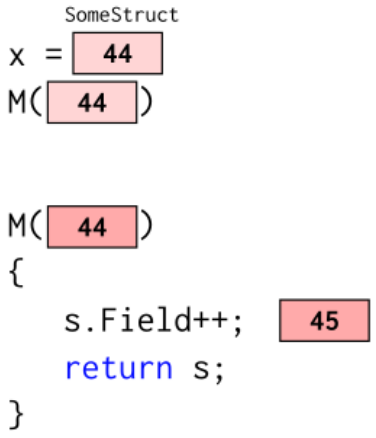
Value type vs reference type

structs are value types (having *pass-by-value* semantics):

```
var x = new SomeStruct() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeStruct M(SomeStruct s)
{
    s.Field++;
    return s;
}

public struct SomeStruct
{
    public int Field;
}
```



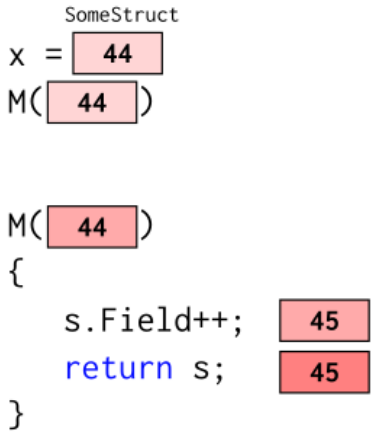
Value type vs reference type

structs are value types (having *pass-by-value* semantics):

```
var x = new SomeStruct() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeStruct M(SomeStruct s)
{
    s.Field++;
    return s;
}

public struct SomeStruct
{
    public int Field;
}
```



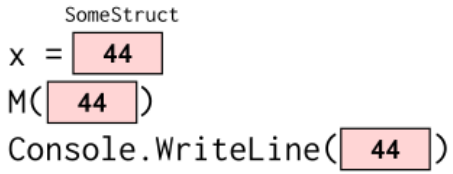
Value type vs reference type

structs are value types (having *pass-by-value* semantics):

```
var x = new SomeStruct() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeStruct M(SomeStruct s)
{
    s.Field++;
    return s;
}

public struct SomeStruct
{
    public int Field;
}
```



Value type vs reference type

*class*es are reference types (having *pass-by-reference* semantics):

```
var x = new SomeClass() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeClass M(SomeClass s)
{
    s.Field++;
    return s;
}

public class SomeClass
{
    public int Field;
}
```

Value type vs reference type

*class*es are reference types (having *pass-by-reference* semantics):

```
var x = new SomeClass() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeClass M(SomeClass s)
{
    s.Field++;
    return s;
}

public class SomeClass
{
    public int Field;
}
```

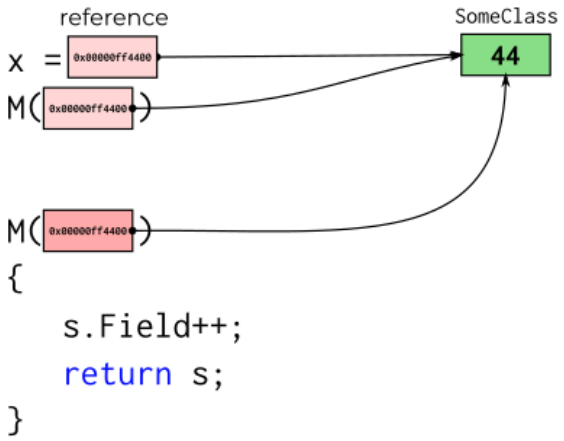
Value type vs reference type

*class*es are reference types (having *pass-by-reference* semantics):

```
var x = new SomeClass() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeClass M(SomeClass s)
{
    s.Field++;
    return s;
}

public class SomeClass
{
    public int Field;
}
```



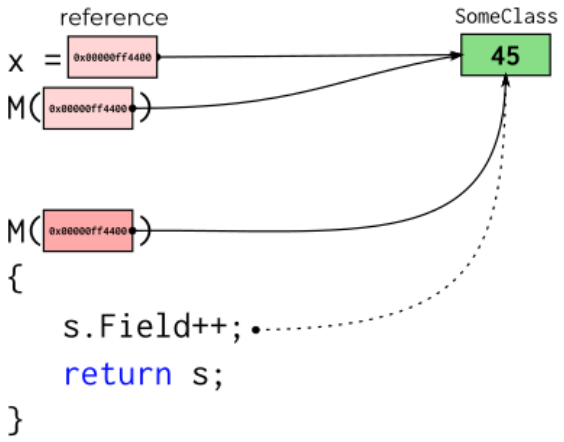
Value type vs reference type

*class*es are reference types (having *pass-by-reference* semantics):

```
var x = new SomeClass() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeClass M(SomeClass s)
{
    s.Field++;
    return s;
}

public class SomeClass
{
    public int Field;
}
```



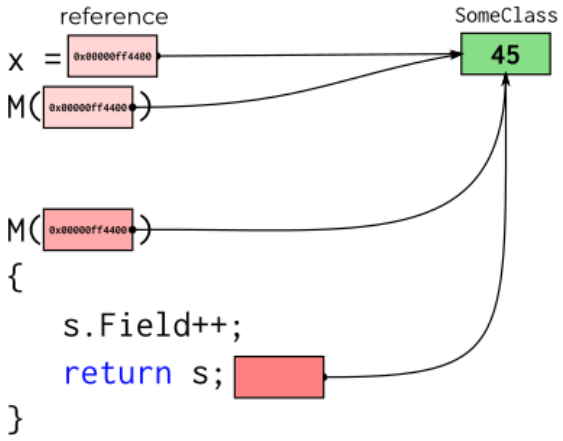
Value type vs reference type

*class*es are reference types (having *pass-by-reference* semantics):

```
var x = new SomeClass() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeClass M(SomeClass s)
{
    s.Field++;
    return s;
}

public class SomeClass
{
    public int Field;
}
```



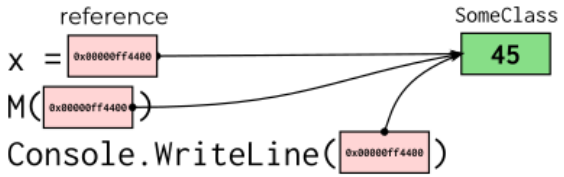
Value type vs reference type

*class*es are reference types (having *pass-by-reference* semantics):

```
var x = new SomeClass() { Field = 44 };
C.M(x);
Console.WriteLine(x.Field);

static public SomeClass M(SomeClass s)
{
    s.Field++;
    return s;
}

public class SomeClass
{
    public int Field;
}
```



Value type vs reference type

Value types:

- built-in types: **`bool`**, **`char`**, **`float`**, **`double`**, **`int`** variations
- enums
- user-defined **`struct`**

Reference types:

- user-defined **`class`**
- built-in types: **`string`**, **`object`**, arrays
- boxed types
- delegates
- interface type
- pointer type

Value type vs reference type

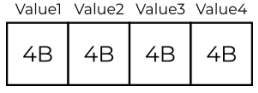
```
public struct SomeStruct
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
}
```

Value1	Value2	Value3	Value4
4B	4B	4B	4B

- only fields, no additional data (*"The values described by a value type are **self-contained**"*)
- automatic layout*

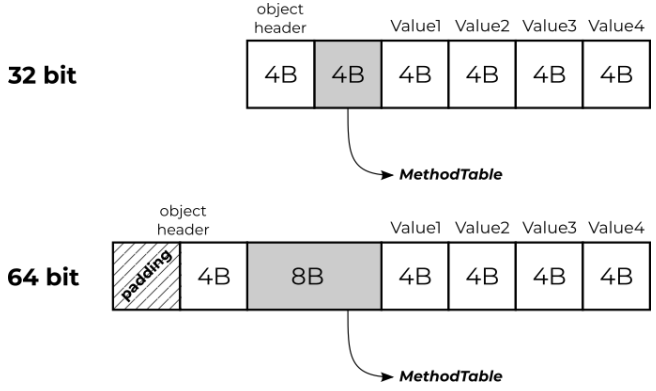
Value type vs reference type

```
public struct SomeStruct
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
}
```



- only fields, no additional data (*"The values described by a value type are **self-contained**"*)
- automatic layout*

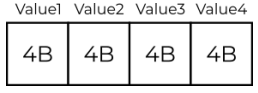
```
public class SomeClass
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
}
```



- object header
- ***MethodTable*** pointer to the type description
- fields with automatic layout*

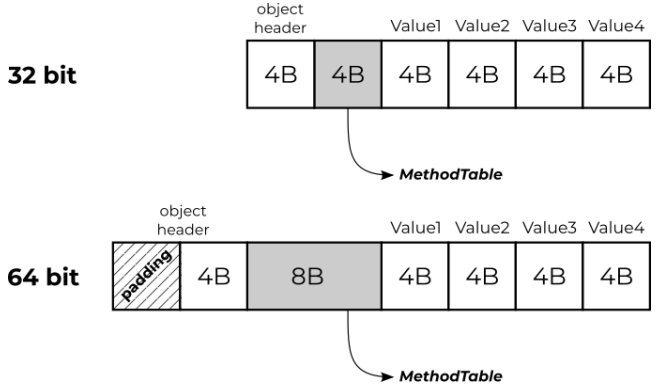
Value type vs reference type

```
public struct SomeStruct
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
}
```



- only fields, no additional data (*"The values described by a value type are **self-contained**"*)
- automatic layout*

```
public class SomeClass
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
}
```



- object header
- ***MethodTable*** pointer to the type description
- fields with automatic layout*

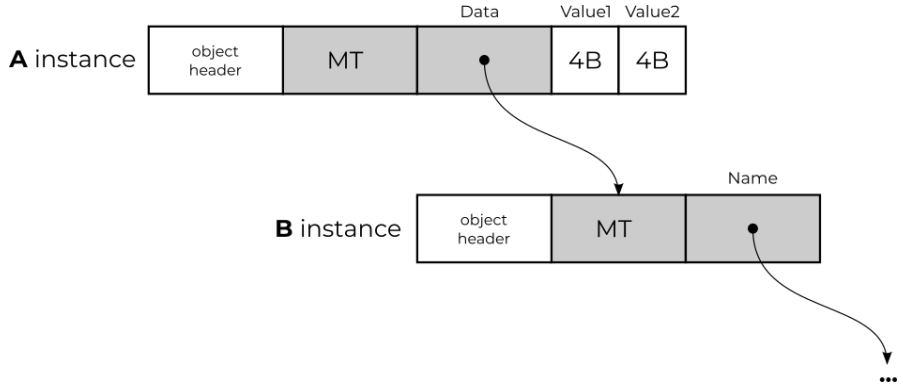
* we will cover controlling layout by the end of the course

Reference types

```
class A
{
  public B Data;
  public int Value1;
  public int Value2;
}

class B
{
  public string Name;
}
```

References (values of reference types) point to the ***MethodTable***:



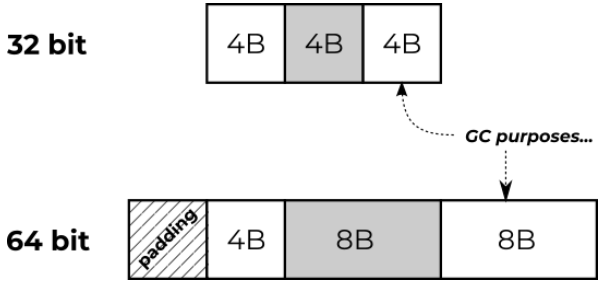
References may have also **`null`** value - we'll explain it in one of the next lessons.

Minimum object size

```
struct S
{
}
```

Is *one byte*. Period.

```
class C
{
}
```



Is:

- 12 bytes on 32-bit runtime
- 24 bytes on 64-bit runtime

Maximum object size

- in general - **2 GB**. Period.

Maximum object size

- in general - **2 GB**. Period.
- but... since .NET 4.5 - `<gcAllowVeryLargeObjects>` flag for 64-bit runtime
  - size fitting 64-bit signed long value - huuuge,
  - but... the maximum number of elements in an array is `UInt32.MaxValue` (4,294,967,295)
  - the maximum size for strings and other non-array objects is unchanged.

DEMO