.net core

Shalabayev Yerassyl Vladimirovich

yerassyl.shalabayev@outlook.com

Overview of lecture

- Types
- Casting
- Process and threads
- Stack and heap
- Primitive types
- Value and reference types
- Boxing unboxing
- Classes, fields and parameters

Types

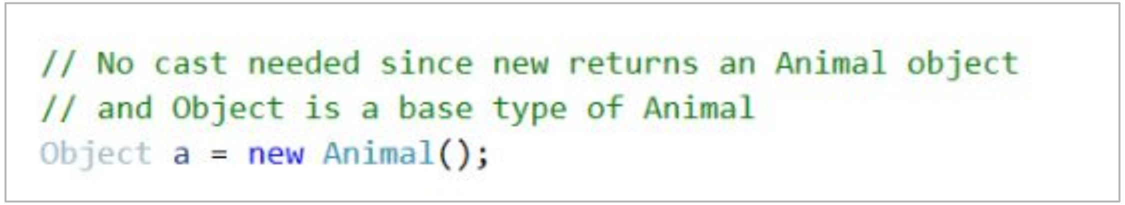
- Every type is derived from System.Object
    - Equals
    - GetHashCode
    - ToString
    - GetType
    - MemberwiseClone
    - Finalize
    - Every type has to be created by the **new** operator
        - *Animal a =* ***new*** *Animal ("Dog");*

New operator

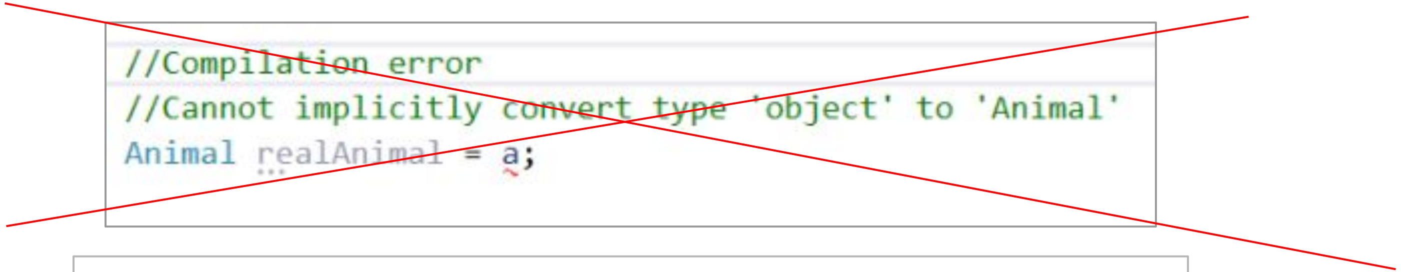
1. Calculates the number of bytes
2. Allocates memory for the object from the managed heap
3. The type’s instance constructor is called
4. Returns a reference (or pointer) to the newly created object

The **new** operator has no complementary **delete** operator. That is, there is no way to explicitly free the memory allocated for an object

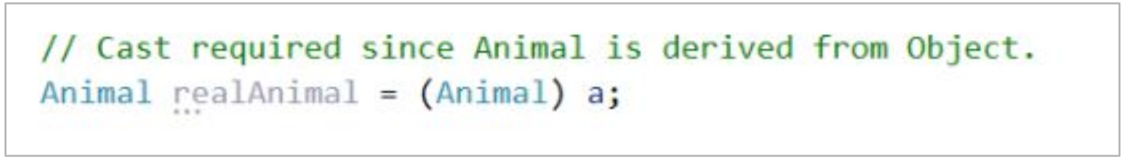
Casting



```
// No cast needed since new returns an Animal object
// and Object is a base type of Animal
Object a = new Animal();
```



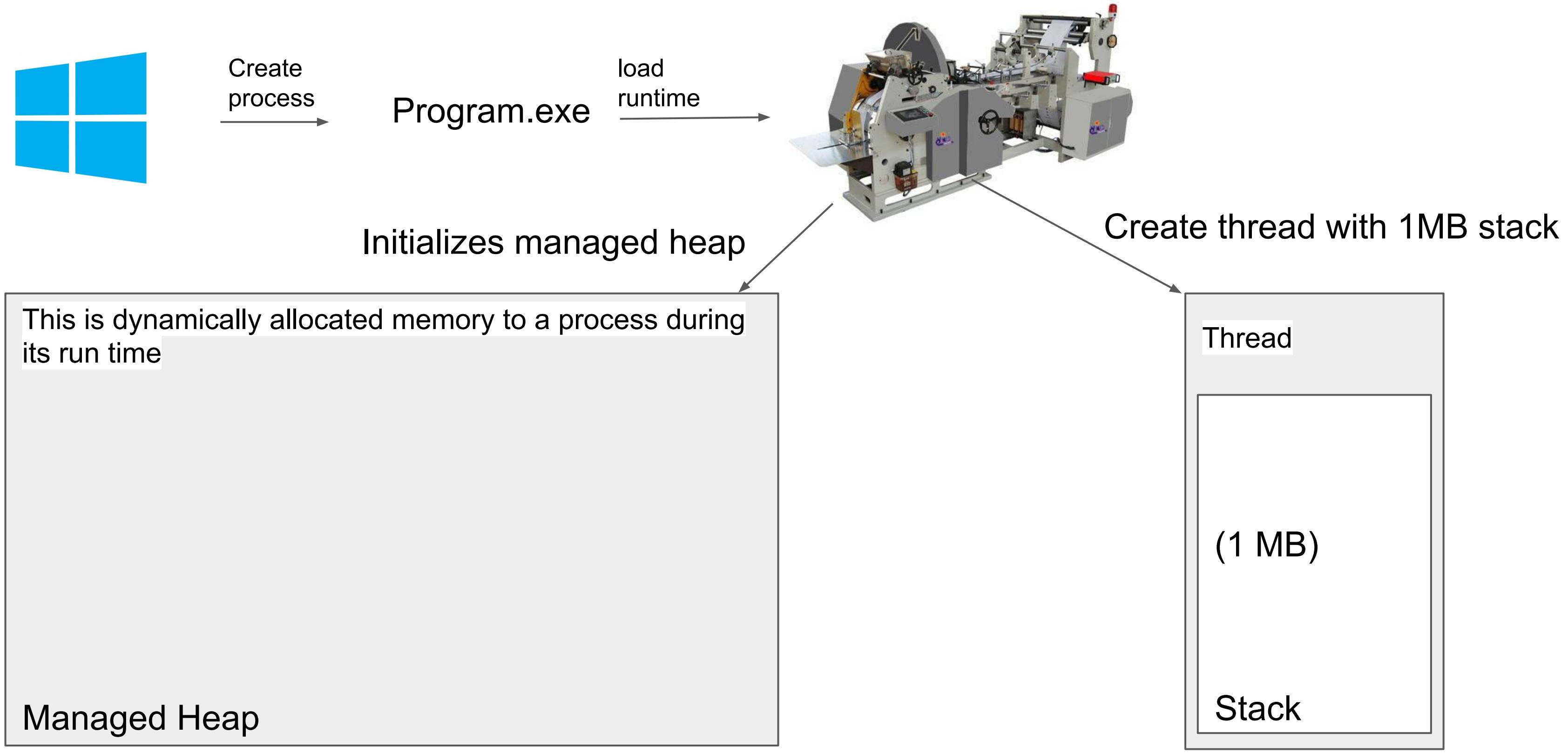
```
~~//Compilation error~~
~~//Cannot implicitly convert type 'object' to 'Animal'~~
~~Animal realAnimal = a;~~
```



```
// Cast required since Animal is derived from Object.
Animal realAnimal = (Animal) a;
```

At runtime, the CLR always knows what type an object is

How os run .net app



This is dynamically allocated memory to a process during its run time

Process and threads

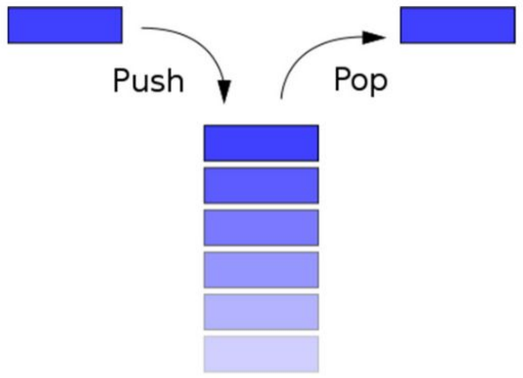
A process is defined as an entity which represents the basic unit of work to be implemented in the system.

In the world of computer programming, a process is an instance or execution of a program

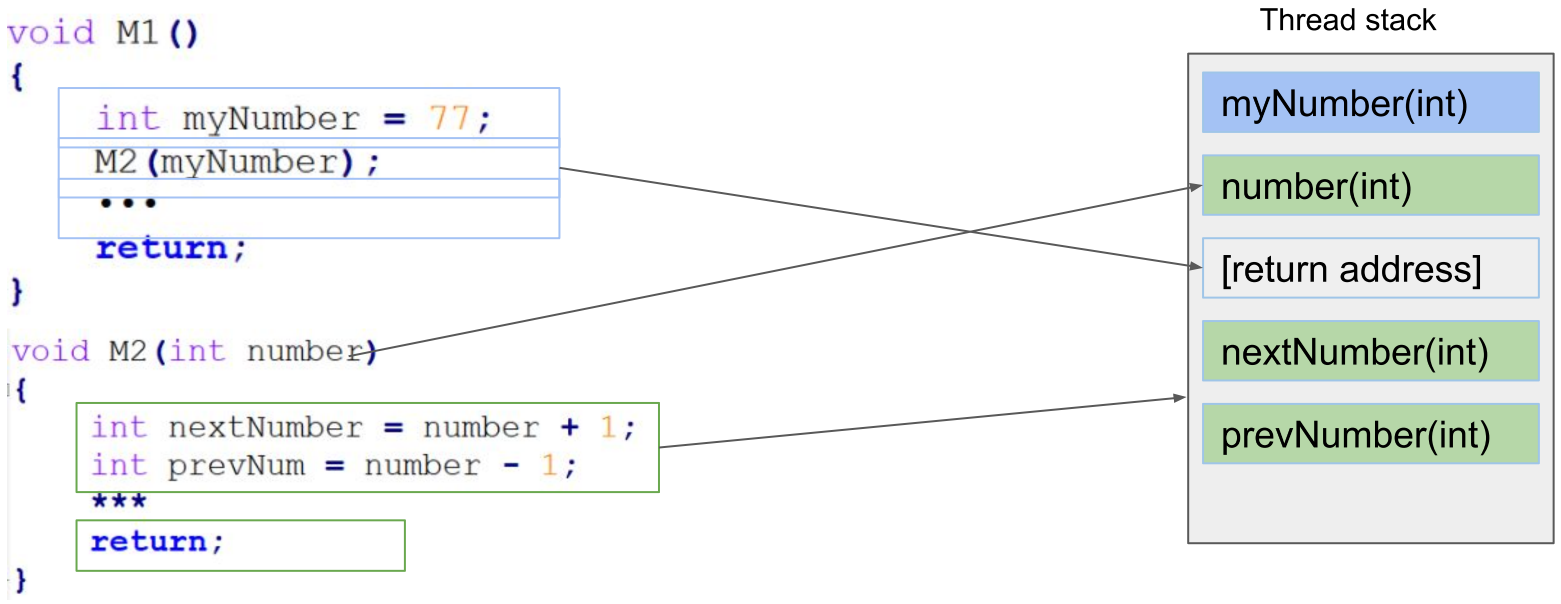
1. A process can contain more than one thread.
2. A process is considered as “heavyweight” while a thread is deemed as “lightweight”.
3. Processes are heavily dependent on system resources available while threads require minimal amounts of resource.
4. Modifying a main thread may affect subsequent threads while changes on a parent process will not necessarily affect child processes.
5. Threads within a process communicate directly while processes do not communicate so easily.
6. Threads are easy to create while processes are not that straightforward.

Stack

- Data structure
- LIFO



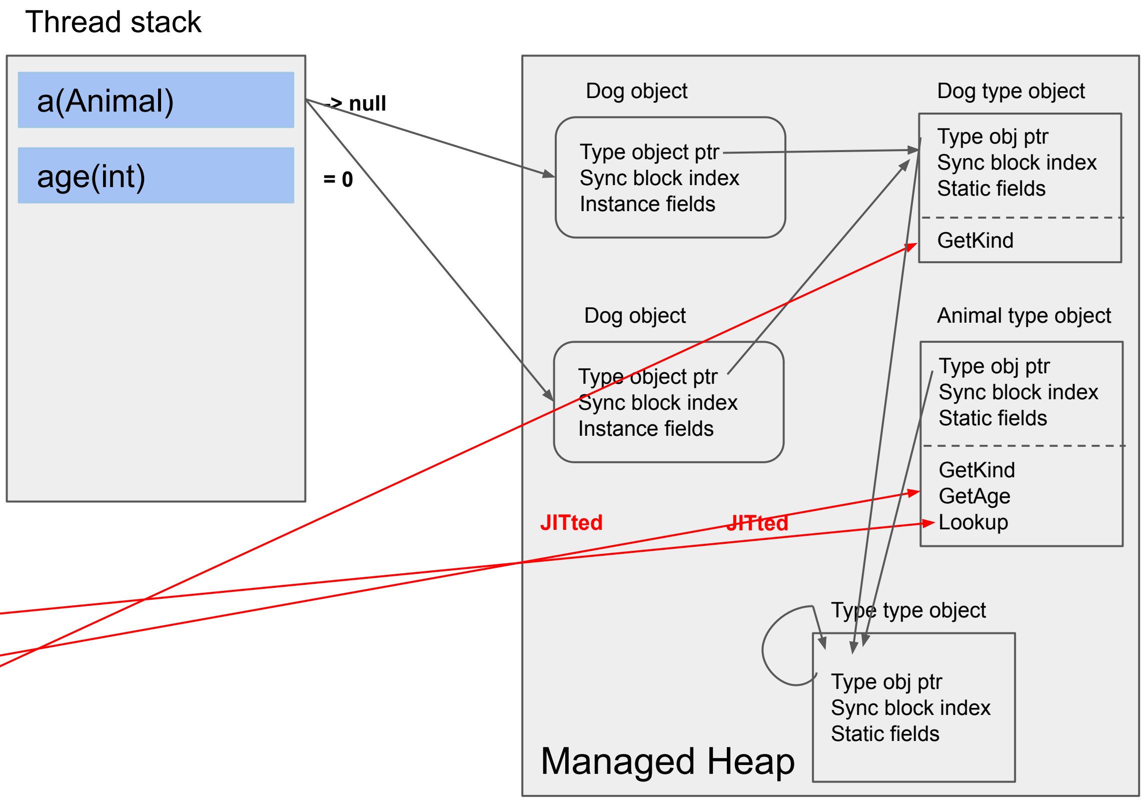
Stack



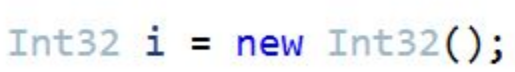
```
void M1()
{
    int myNumber = 77;
    M2(myNumber);
    ...
    return;
}

void M2(int number)
{
    int nextNumber = number + 1;
    int prevNum = number - 1;
    ***
    return;
}
```

Heap

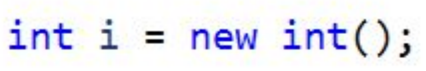
```
class Animal
{
    public int GetAge() { *** }
    public virtual string GetKind() { *** }
    public static Animal Lookup() { *** }
}

class Dog
{
    public override string GetKind(){ *** }
}

void M3(){
    Animal a;
    int age;

    a = new Dog();
    a = Animal.Lookup("Dog")
    age = a.GetAge();
    a.GetKind();
}
```



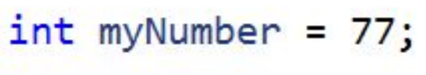
Primitive types



```
Int32 i = new Int32();
```



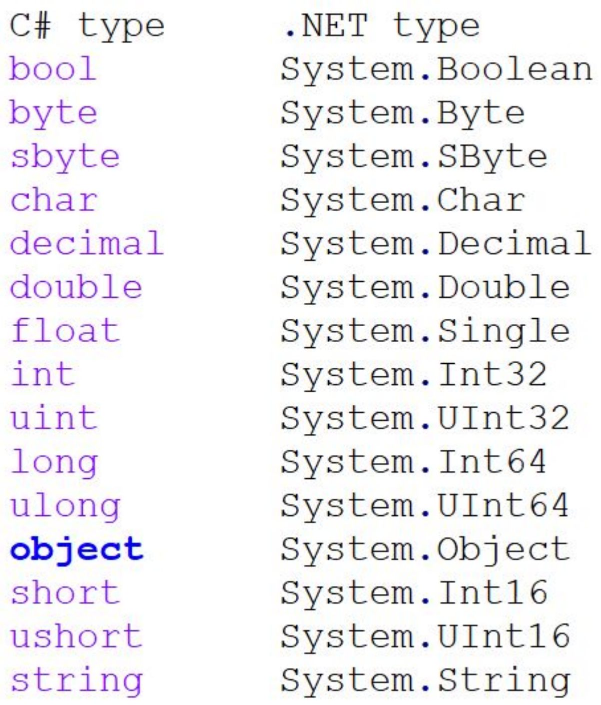
```
int i = new int();
```



```
int myNumber = 77;
```

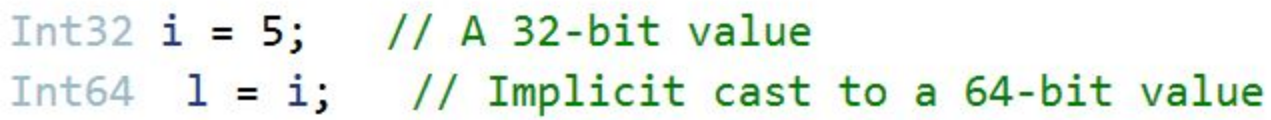
Any data types the compiler directly supports are called primitive types. Primitive types map directly to types existing in the Framework Class Library (FCL)

Primitive types

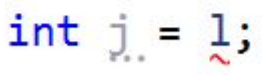


C# type	.NET type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

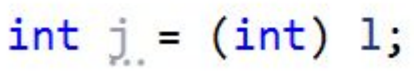
Primitive types casting



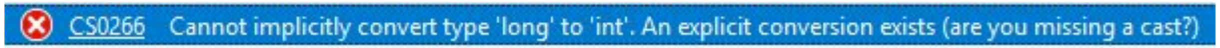
```
Int32 i = 5;   // A 32-bit value
Int64  l = i;   // Implicit cast to a 64-bit value
```



```
int j = l;
```



```
int j = (int) l;
```



CS0266 Cannot implicitly convert type 'long' to 'int'. An explicit conversion exists (are you missing a cast?)

Reference Types and Value Types

The CLR supports two kinds of types: reference types and value types. While most types in the FCL are reference types, the types that programmers use most often are value types. Reference types are always allocated from the managed heap, and the C# new operator returns the memory address of the object—the memory address refers to the object’s bits. You need to bear in mind some performance considerations when you’re working with reference types. First, consider these facts:

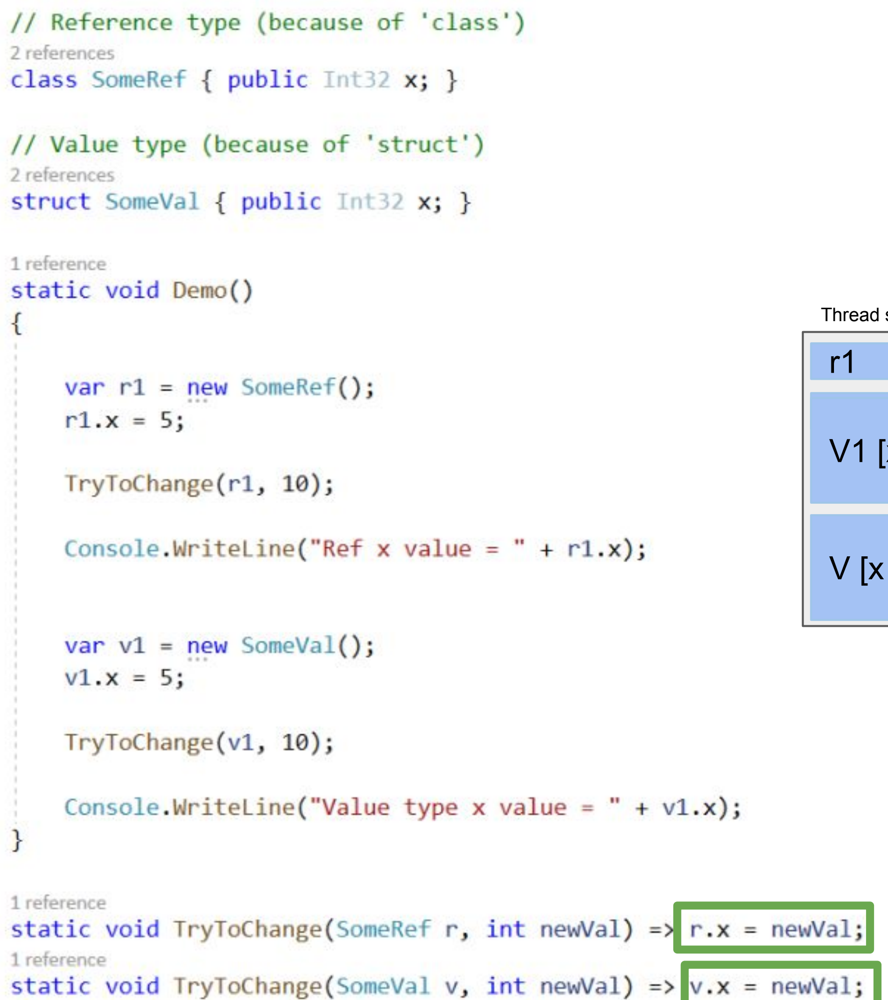
Reference Types

- The memory must be allocated from the managed heap.
- Each object allocated on the heap has some additional overhead members associated with it that must be initialized.
- The other bytes in the object (for the fields) are always set to zero.
- Allocating an object from the managed heap could force a garbage collection to occur.

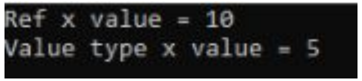
Value types

- All value types are derived implicitly from the System.ValueType
- Unlike with reference types, you cannot derive a new type from a value type. However, like reference types, structs can implement interfaces.
- Value type variables cannot be null by default

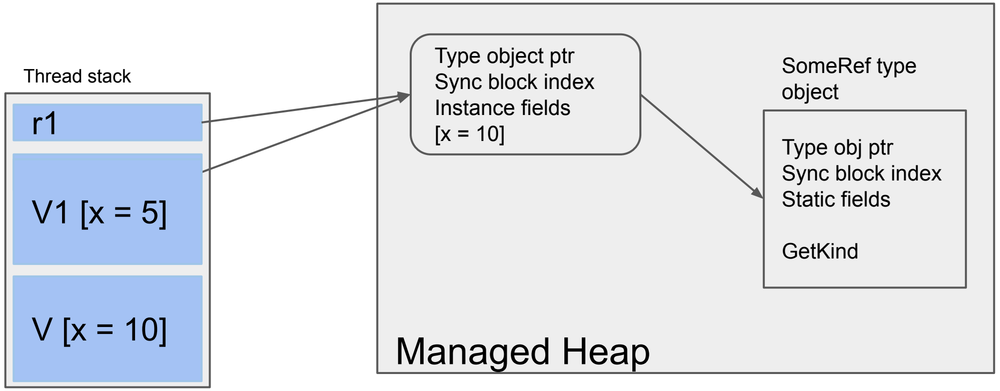
Value types are lighter weight than reference types because they are not allocated as objects in the managed heap, not garbage collected, and not referred to by pointers

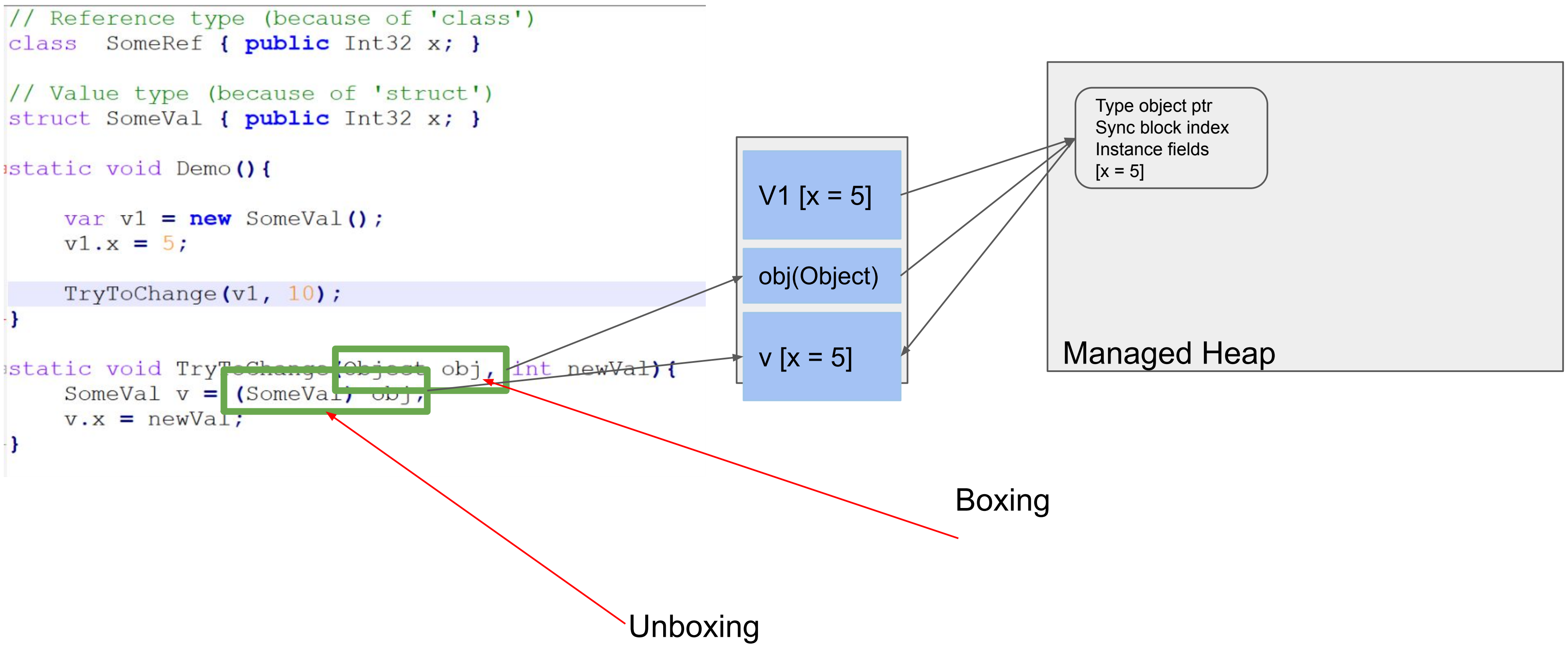


```csharp
// Reference type (because of 'class')
2 references
class SomeRef { public Int32 x; }

// Value type (because of 'struct')
2 references
struct SomeVal { public Int32 x; }

1 reference
static void Demo()
{

    var r1 = new SomeRef();
    r1.x = 5;

    TryToChange(r1, 10);

    Console.WriteLine("Ref x value = " + r1.x);


    var v1 = new SomeVal();
    v1.x = 5;

    TryToChange(v1, 10);

    Console.WriteLine("Value type x value = " + v1.x);
}

1 reference
static void TryToChange(SomeRef r, int newVal) => r.x = newVal;
1 reference
static void TryToChange(SomeVal v, int newVal) => v.x = newVal;
```



```
Ref x value = 10
Value type x value = 5
```





```
// Reference type (because of 'class')
class  SomeRef { public Int32 x; }

// Value type (because of 'struct')
struct SomeVal { public Int32 x; }

static void Demo(){

    var v1 = new SomeVal();
    v1.x = 5;

    TryToChange(v1, 10);
}

static void TryToChange(Object obj, int newVal){
    SomeVal v = (SomeVal) obj;
    v.x = newVal;
}
```

Static classes

There are certain classes that are never intended to be instantiated, such as Console, Math, Environment, and ThreadPool. These classes have only static members and, in fact, the classes exist simply as a way to group a set of related members together. For example, the Math class defines a bunch of methods that do math-related operations. C# allows you to define non-instantiable classes by using the C# static keyword.

Static Classes

• The class must be derived directly from System.Object because deriving from any other base class makes no sense since inheritance applies only to objects, and you cannot create an instance of a static class.

• The class must not implement any interfaces since interface methods are callable only when using an instance of a class.

• The class must define only static members (fields, methods, properties, and events). Any instance members cause the compiler to generate an error.

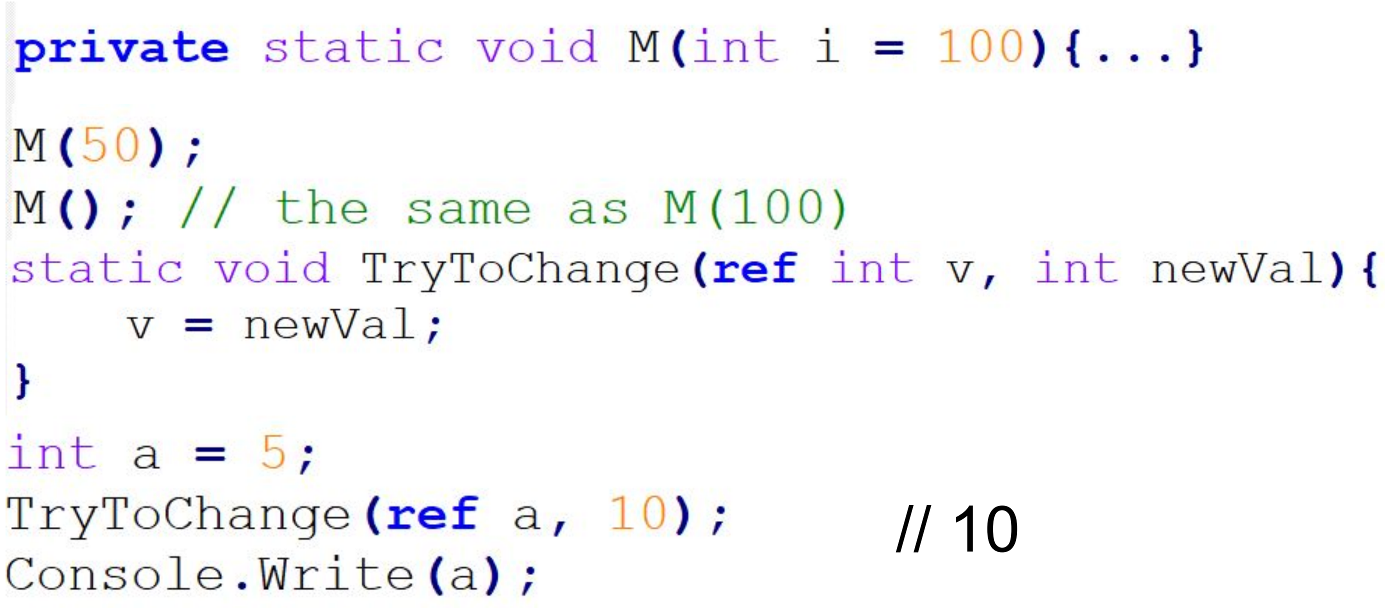
• The class cannot be used as a field, method parameter, or local variable because all of these would indicate a variable that refers to an instance, and this is not allowed. If the compiler detects any of these uses, the compiler issues an error.

c# keywords	Class	Method/Property/Event	Constant/Field
abstract	Indicates that no instances of the type can be constructed	Indicates that the derived type mustoverride and implement this member before instances of the derived type can be constructed	—
virtual	-	Indicates that this member can be overridden by a derived type	-
override	-	Indicates that the derived type is overriding the base type's member	-
sealed	Indicates that the type cannot be used as a base type	Indicates that the member cannot be overridden by a derived type. This keyword can be applied only to a method that is overriding a virtual method.	-
new	When applied to a nested type, method, property, event, constant, or field, indicates that the member has no relationship to a similar member that may exist in the base class		

Fields

c# term	Description
static	The field is part of the type’s state, as opposed to being part of an object’s state.
(default)	The field is associated with an instance of the type, not the type itself.
readonly	The field can be written to only by code contained in a constructor method.
volatile	keyword indicates that a field might be modified by multiple threads that are executing at the same time

Ways of passing parameters to methods



```
private static void M(int i = 100){...}

M(50);
M(); // the same as M(100)
static void TryToChange(ref int v, int newVal){
    v = newVal;
}
int a = 5;
TryToChange(ref a, 10);
Console.Write(a);
```

// 10

```
public sealed class Program {
   public static void Main() {
      Int32 x;                  // x is uninitialized
      GetVal(out x);            // x doesn’t have to be initialized.
      Console.WriteLine(x);  // Displays "10"
   }

   private static void GetVal(out Int32 v) {
      v = 10;  // This method must initialize v.
   }
}
```

Passing a Variable Number of Arguments to a Method

```
// User defined function
public void Show(params int[] val) // Params Paramater
{
    for (int i=0; i<val.Length; i++)
    {
        Console.WriteLine(val[i]);
    }
}
// Main function, execution entry point of the program
static void Main(string[] args)
{
    Program program = new Program(); // Creating Object
    program.Show(2,4,6,8,10,12,14); // Passing arguments of variable length
}
```