7SENG011W Object Oriented Programming

Namespaces and Assemblies, Static Attributes and Static Methods

Dr Francesco Tusa

Readings

The topics we will discuss today can be found in the books

- Hands-On Object-Oriented Programming with C#
  - Chapter: Implementation of OOP in C# (Properties and Access Modifiers only)
- Programming C# 10
  - Chapter 3: Types

Online

- C# Properties
- C# Program Structure Overview and Namespaces
- Assemblies in .NET
- Static Keyword
- Static Classes and Static Class Members

Outline

- Review of Encapsulation, C# Properties
- Namespaces and Assemblies
- Static
  - Attributes
  - Methods

Outline

- Review of Encapsulation, C# Properties
- Namespaces and Assemblies
- Static
  - Attributes
  - Methods

Information Hiding: Implementation Hiding

```
class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private string address;

    public Person(string n, string s, int yob)
    { ... }

    // other methods...

    public void SetAddress(string addr) {
        address = addr;
    }

    public string GetAddress() {
        return address;
    }
}
```

different implementation

different implementation

```
class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private Address address;

    public Person(string n, string s, int yob)
    { ... }

    // other methods...

    public void SetAddress(string addr) {
        address = new Address(addr);
        // also checks address is valid
    }

    public string GetAddress() {
        return address.ToString();
    }
}
```

Information Hiding: Implementation Hiding

```
class Program
{
   public static void Main(string args)
   {
      Person  tom = new Person("Tom", "Jones", 1940);

      tom.SetAddress("30 Hampstead Ln; London; N6 4NX");
      // ...
      Console.WriteLine($"{tom.GetName()} lives at {tom.GetAddress()}");

}
```

Should **all** the classes that use *Person* be **modified**?

No! This class and all other (possibly hundreds of) classes that already use *Person* **will continue to work without the need of any changes**

Get and Set Methods: C# properties

- C# also supports an additional way to define attributes with associated *getter* and *setter* methods

```
class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;

    private string address;

    public void SetAddress(string addr) {
        address = addr;
    }

    public string GetAddress() {
        return address;
    }

    public Person(string n, string s, int yob) { … }
}
```

```
class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    // public way to access the property
    public string address
    {
      get;
      set;
    } // public getter/setter methods

    public Person(string n, string s, int yob) { … }
}
```

Get and Set Methods: C# properties

- C# also supports an additional way to define attributes with associated *getter* and *setter* methods

```
class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;

    private Address address;

    public void SetAddress(string addr) {
        address = new Address(addr);
    }

    public string GetAddress() {
        return address.ToString();
    }

    public Person(string n, string s, int yob) { ... }
}
```

```
class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;

    private Address _address;

    public string address
    {
        set { _address = new Address(value); }

        get { return _address.ToString() }
    }

    public Person(string n, string s, int yob) { ... }
}
```

keyword that contains the received value

Information Hiding: Implementation Hiding

```
class Program
{
    public static void Main(string args)
    {
        Person  tom = new Person("Tom", "Jones", 1940);

        // tom.SetAddress("30 Hampstead Ln, London N6 4NX");
        tom.address = "30 Hampstead Ln; London; N6 4NX";

        // Console.WriteLine($"{tom.GetName()} lives at {tom.GetAddress()}");
        Console.WriteLine($"{tom.GetName()} lives at {tom.address}");

}
```

Properties can be used as if they were public data attributes, but they are special methods called *accessors*.

They expose a public way of *getting* and *setting* values, while **hiding** implementation or verification code.

Information Hiding: Implementation Hiding

```
class Program
{
    public static void Main(string args)
    {
        Person  tom = new Person("Tom", "Jones", 1940);

        // tom.SetAddress("30 Hampstead Ln, London N6 4NX");
        tom.address = "30 Hampstead Ln; London; N6 4NX";

        // Console.WriteLine($"{tom.GetName()} lives at {tom.GetAddress()}");
        Console.WriteLine($"{tom.GetName()} lives at {tom.address}");

}
```

does not use the *address* attribute directly, instead causes the *associated set method to be called*

Information Hiding: Implementation Hiding

```
class Program
{
    public static void Main(string args)
    {
        Person  tom = new Person("Tom", "Jones", 1940);

        // tom.SetAddress("30 Hampstead Ln, London N6 4NX");
        tom.address = "30 Hampstead Ln; London; N6 4NX";

        // Console.WriteLine($"{tom.GetName()} lives at {tom.GetAddress()}");
        Console.WriteLine($"{tom.GetName()} lives at {tom.address}");

}
```

does not use the *address* attribute directly, instead causes the *associated get method to be called*

Outline

- Review of Encapsulation, C# Properties
- Namespaces and Assemblies
- Static
  - Attributes
  - Methods

Namespaces

```
class Point { ... }
```

```
class Segment { ... }
```

```
class Circle { ... }
```

```
class XXX { ... }
```

```
class Program
{
    static void Main(string[] args)
    {
        Point p1 = new Point(6, 4);
        Point p2 = new Point(8, 2);
        Circle c1 = new Circle(p1, 4)
        Segment s1 = new Segment(p1, p2)
        p2.Display();
        c1.Area();
        s1.Length();
    }
}
```

Namespaces

```
class Point { ... }
```

Point.cs

```
class Circle { ... }
```

Circle.cs

```
class Segment { ... }
```

Segment.cs

```
class XXX { ... }
```

XXX.cs

```
class Program
{
    static void Main(string[] args)
    {
        Point p1 = new Point(6, 4);
        Point p2 = new Point(8, 2);
        Circle c1 = new Circle(p1, 4)
        Segment s1 = new Segment(p1, p2)
        p2.Display();
        c1.Area();
        s1.Length();
    }
}
```

Program.cs

Namespaces

```
class Point { ... }
```

Point.cs

```
class Circle { ... }
```

Circle.cs

```
class Segment { ... }
```

Segment.cs

```
class XXX { ... }
```

XXX.cs

```
class Program
{
    static void Main(string[] args)
    {
        Point p1 = new Point(6, 4);
        Point p2 = new Point(8, 2);
        Circle c1 = new Circle(p1, 4)
        Segment s1 = new Segment(p1, p2)
        p2.Display();
        c1.Area();
        s1.Length();
    }
}
```

Program.cs

namespace Shapes

Question

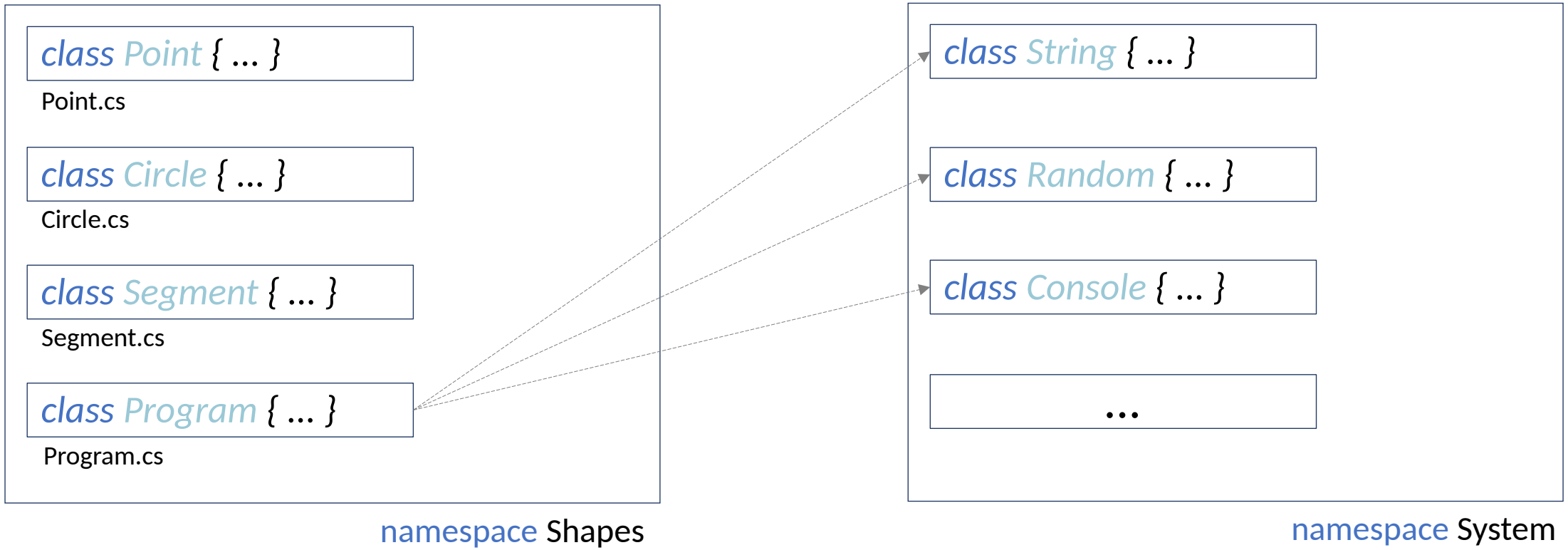
What is a namespace?

Question

What is a namespace?

- A way to organise classes in large programming projects
- A way to control how class and method names are exposed to other programs

Namespaces: System Library



Namespaces: System Library

```
namespace Shapes
{
    class Program
    {
        static void Main(string[] args)
        {
        ...
        Random r = new Random();
        ...
        }
    }
}
```

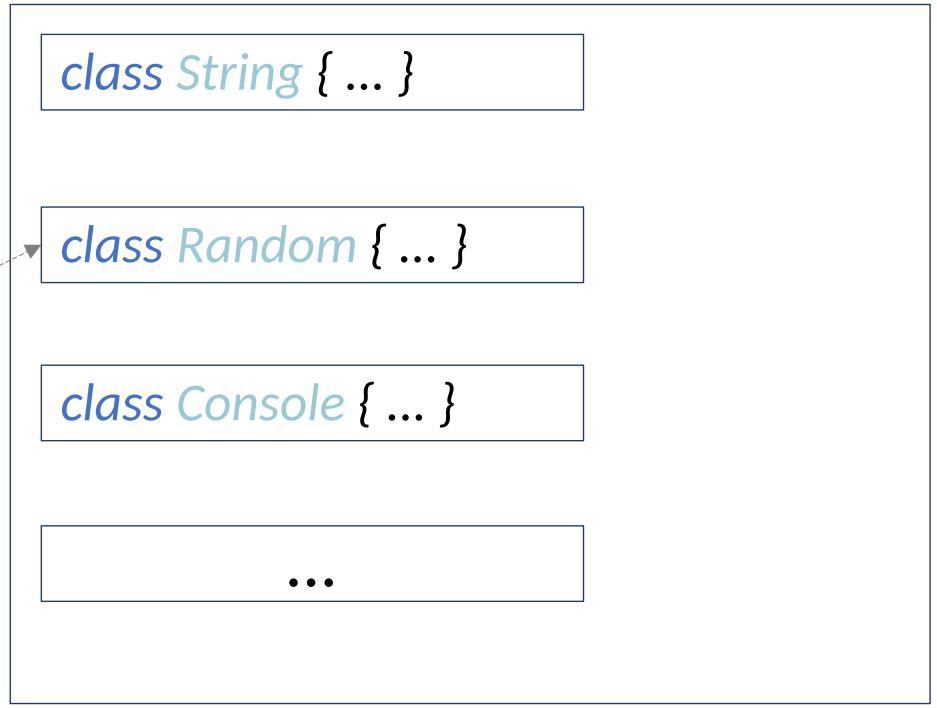
Where is *Random* defined?
The compiler would look
inside *Shapes*

```
class String { ... }

class Random { ... }

class Console { ... }

...
```

namespace System

Namespaces: the using keyword

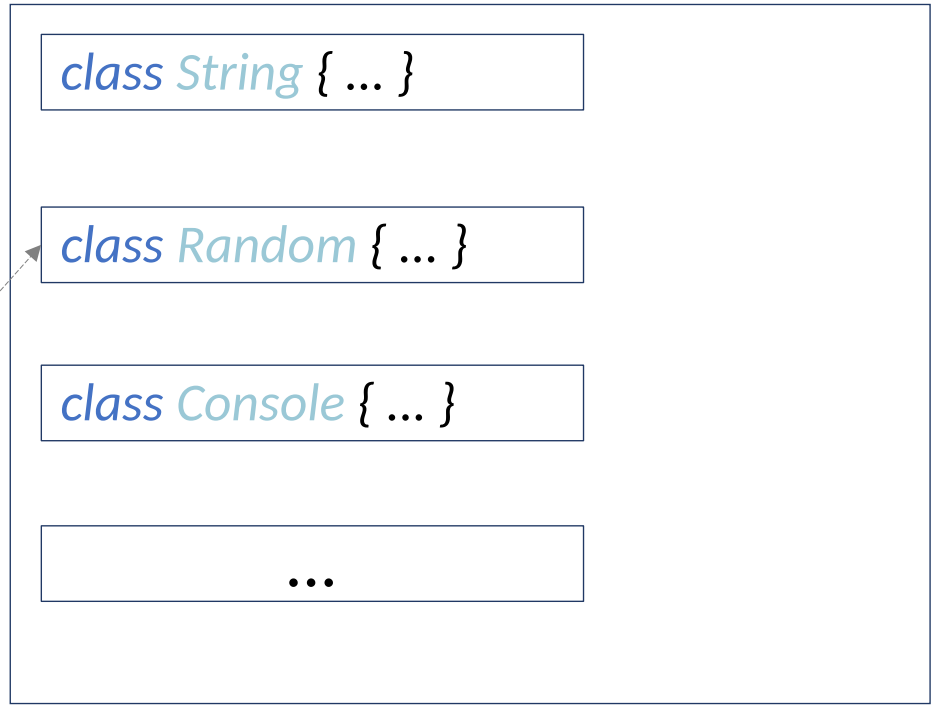
```
using System;
namespace Shapes
{
    class Program
    {
        static void Main(string[] args)
        {
        ...
        Random r = new Random();
        ...
        }
    }
}
```



namespace System

Namespaces: the using keyword

```
// using System;
namespace Shapes
{
    class Program
    {
        static void Main(string[] args)
        {
        ...
        System.Random r = new System.Random();
        ...
        }
    }
}
```



namespace System

Question

What are program ***assemblies*** in C#?

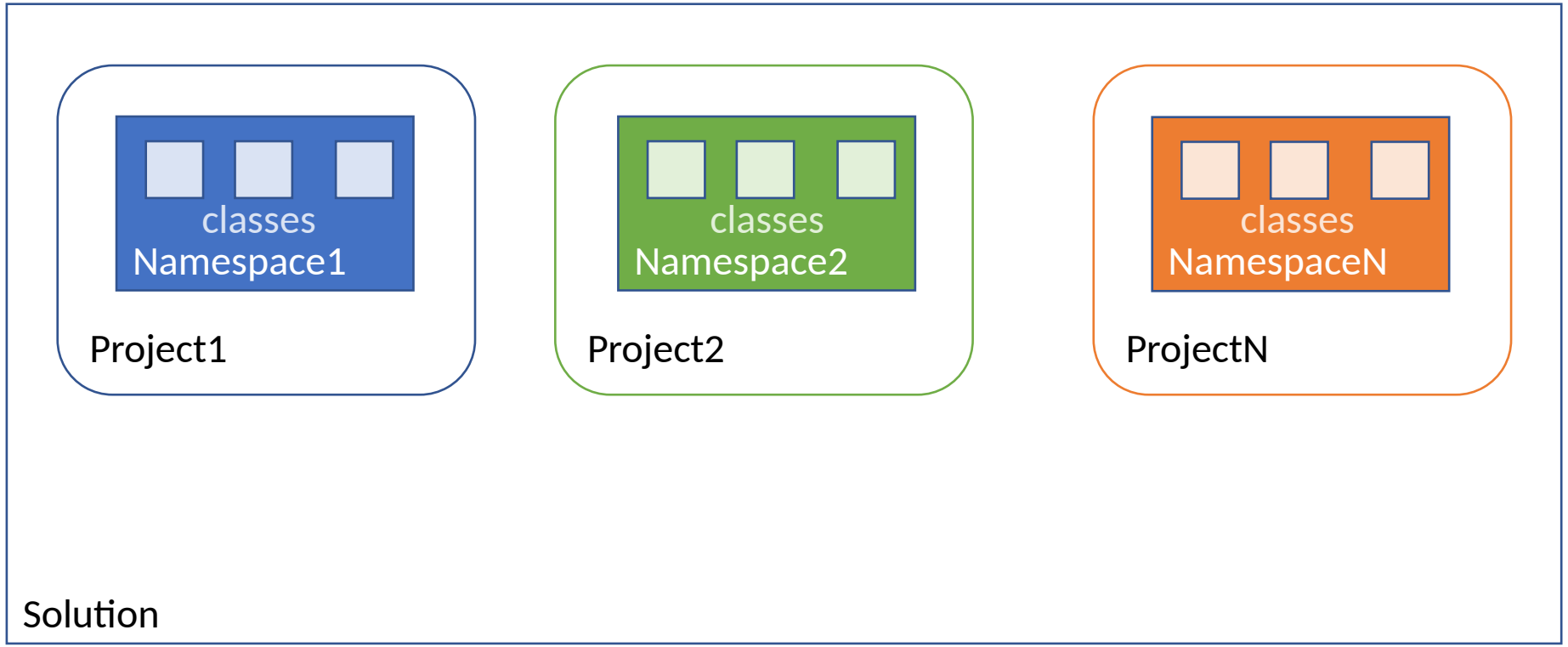
Question

What are program ***assemblies*** in C#?

- **Building blocks** of .NET applications—*collections of types* that form a *logical unit of functionality*

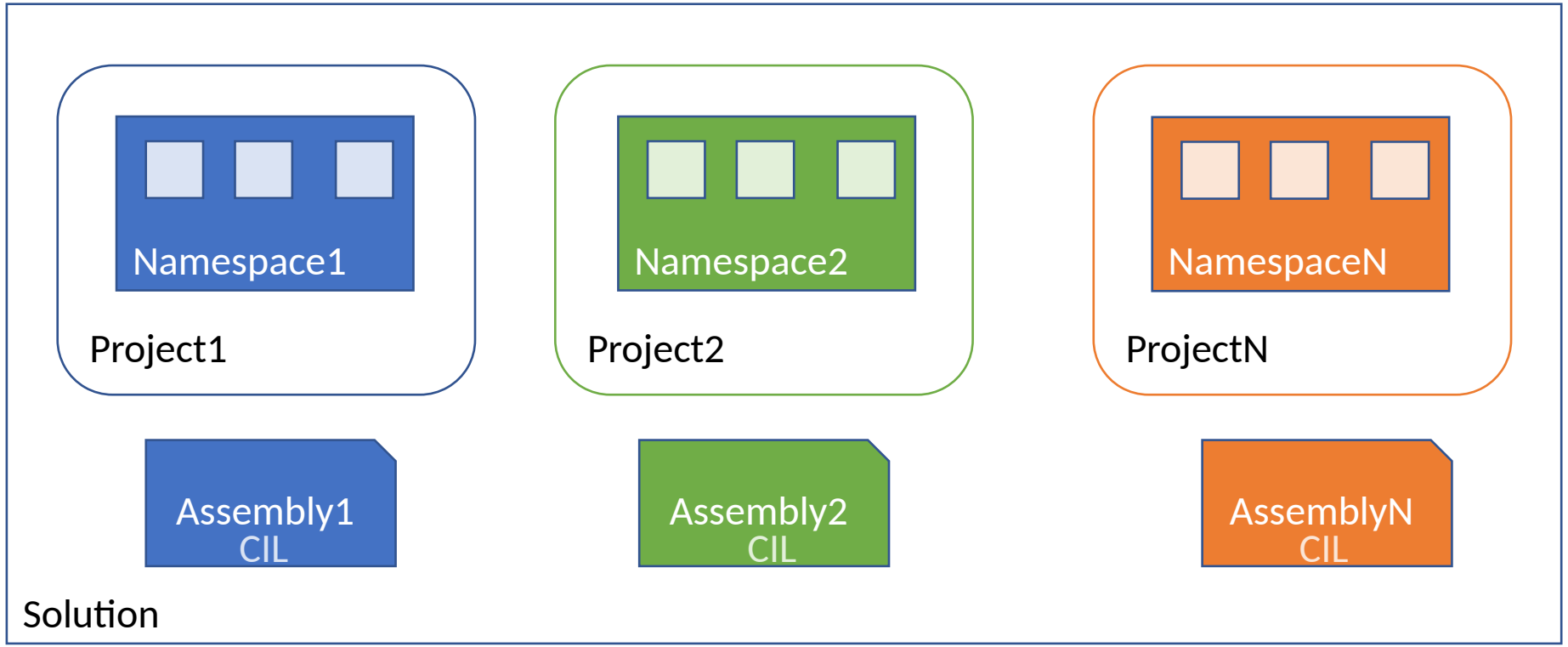
- Contain the *Common Intermediate Language (CIL)* code generated when a *.NET project* is built—*executable* (.exe) or *dynamic link library* (.dll)

Assemblies: .NET projects



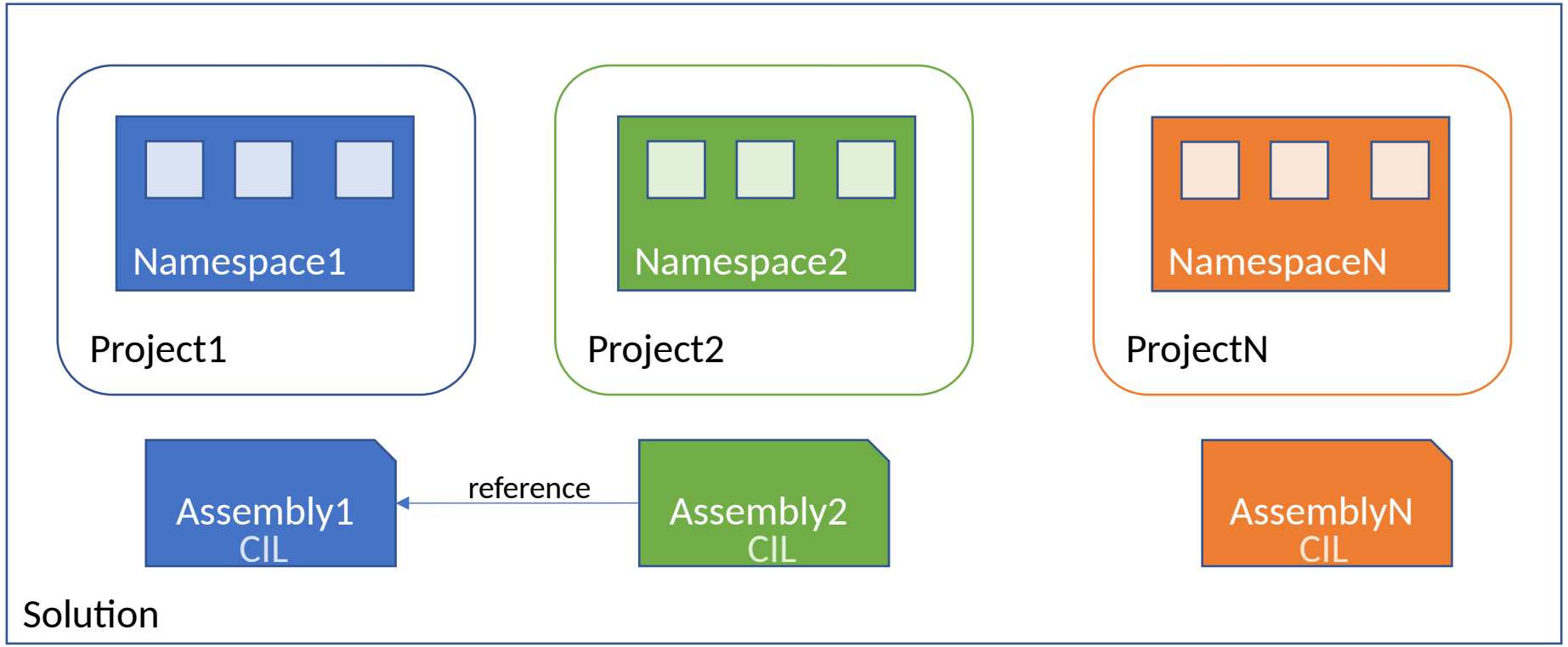
We know that a *.NET Solution* can be a container for multiple *Projects*
Different people can work on different Projects that will become a single *.NET Application*

Assemblies: references



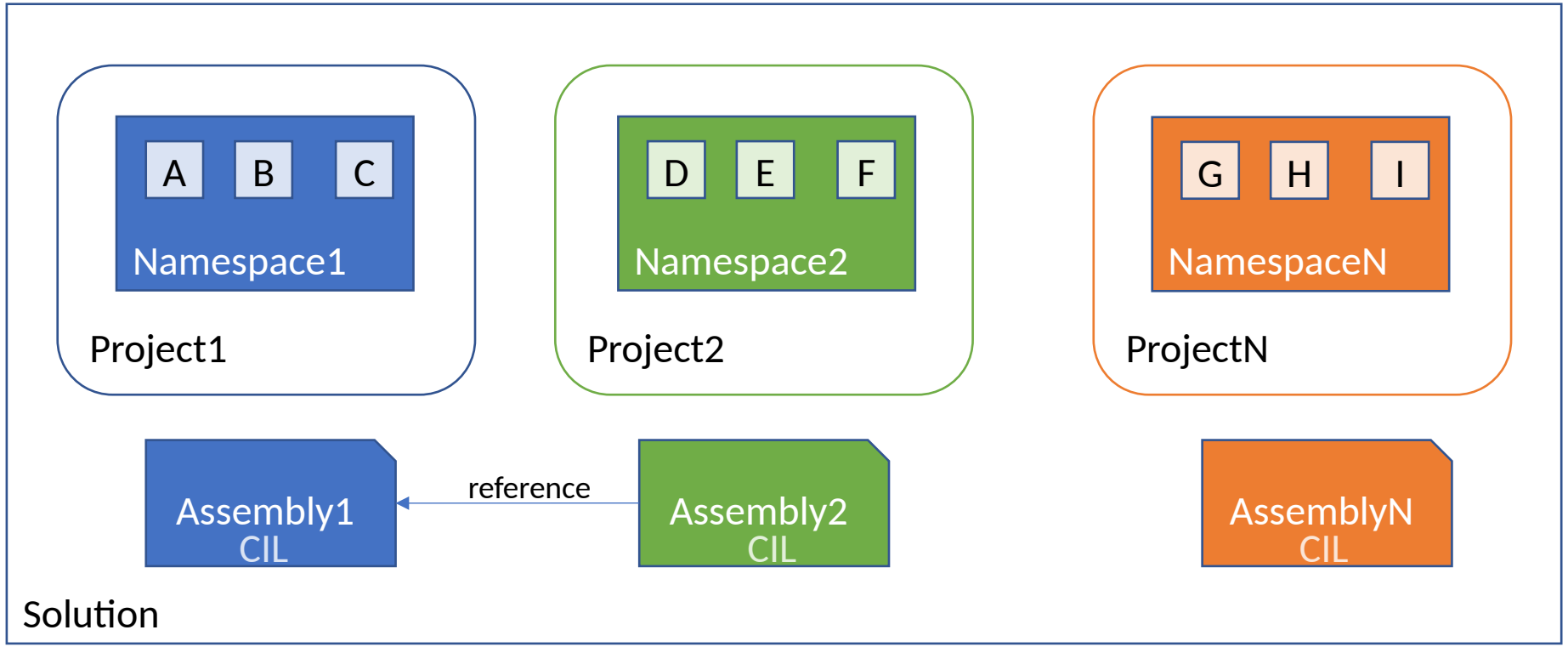
When each project is built, an *assembly* with the associated *CIL code* is generated

Assemblies: references



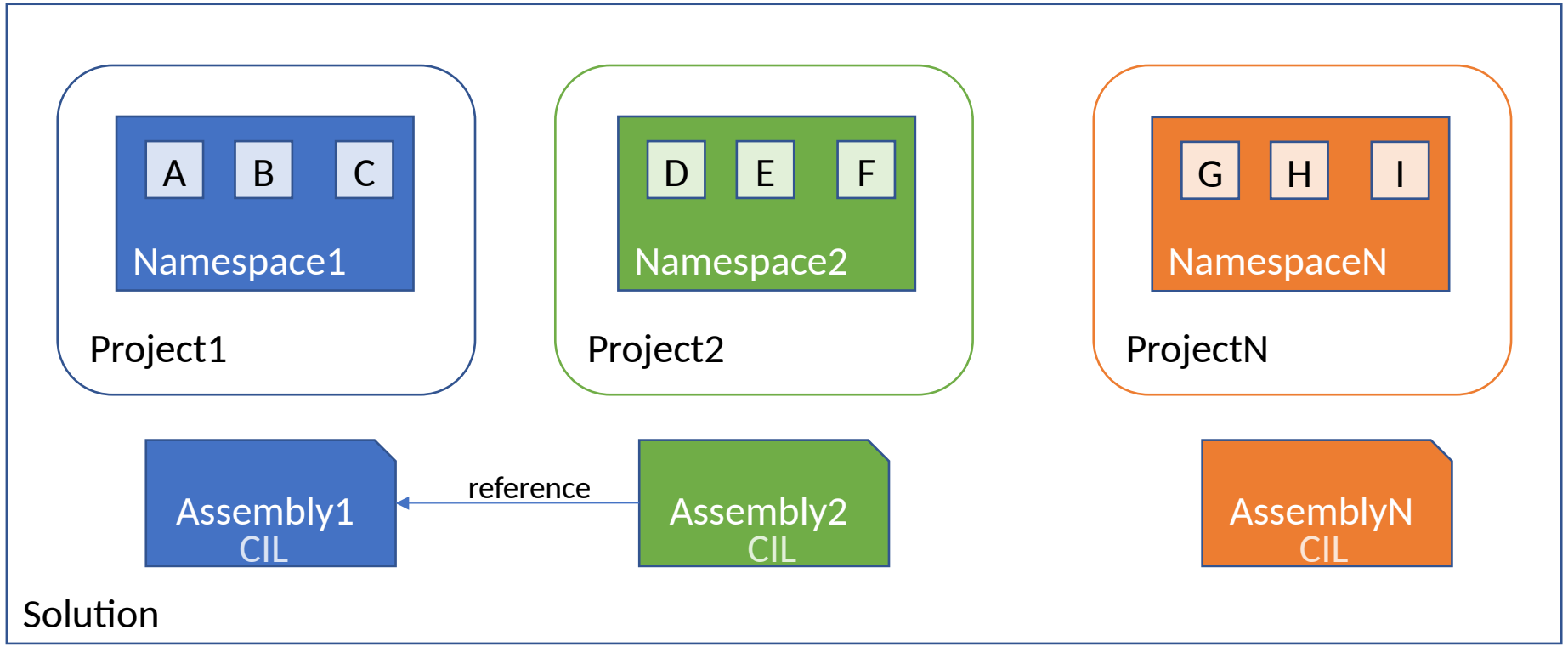
An *assembly* can reference another *assembly* to build the final *.NET Application*

Assemblies: references



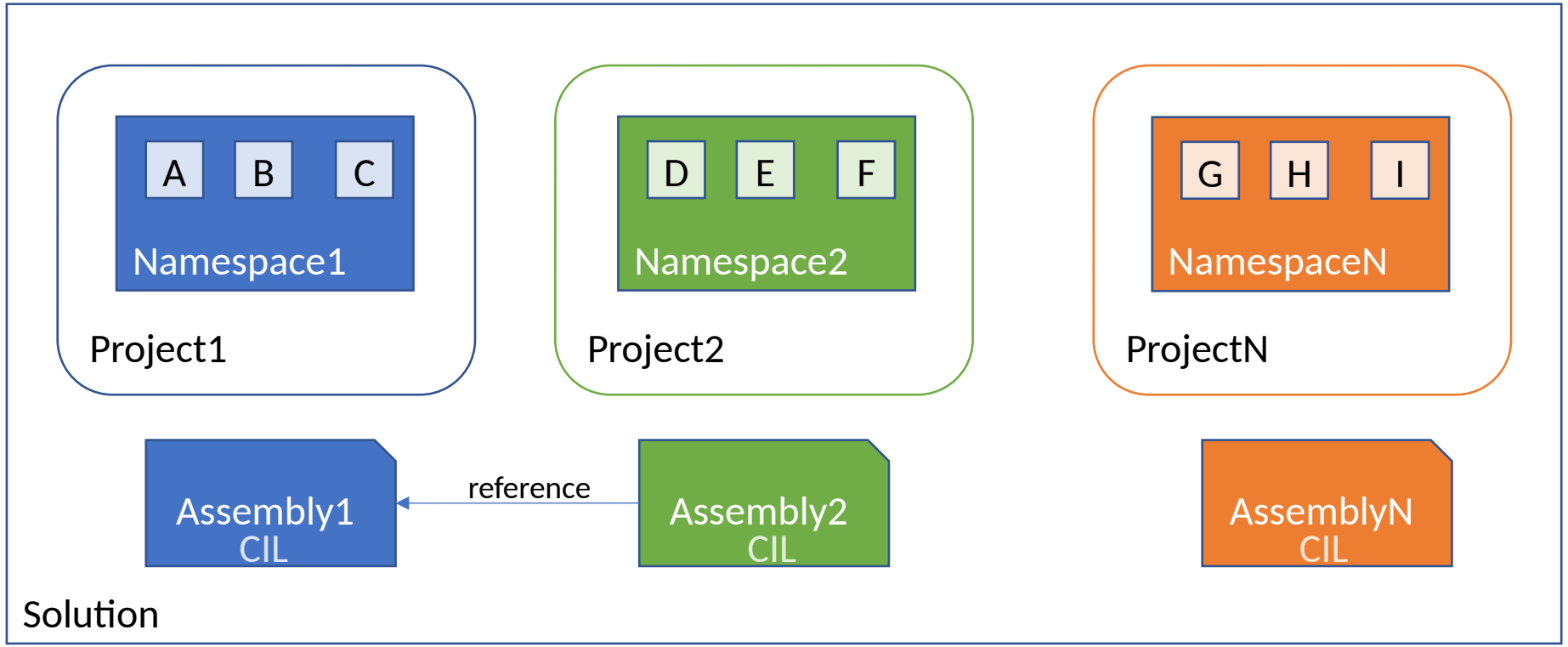
classes are by default visible **only inside the assembly** (project) where they have been defined
For example, classes A, B, C **could not be used** by D, E, F even though a **reference** exists

Assemblies: access modifiers




```
internal class A { ... } // visible only by classes B and C (Assembly1)
public class A { ... } // visible by all the classes that reference Assembly1 – classes D, E and F
```

Assemblies: access modifiers



classes are implicitly assigned the *internal* access modifier
The *public* access modifier can be used when a class is defined, to make it visible outside of its assembly

internal access modifier

- If no access modifier is specified when defining a class, it will be declared as internal
- internal classes are only **visible inside their assembly**

internal access modifier

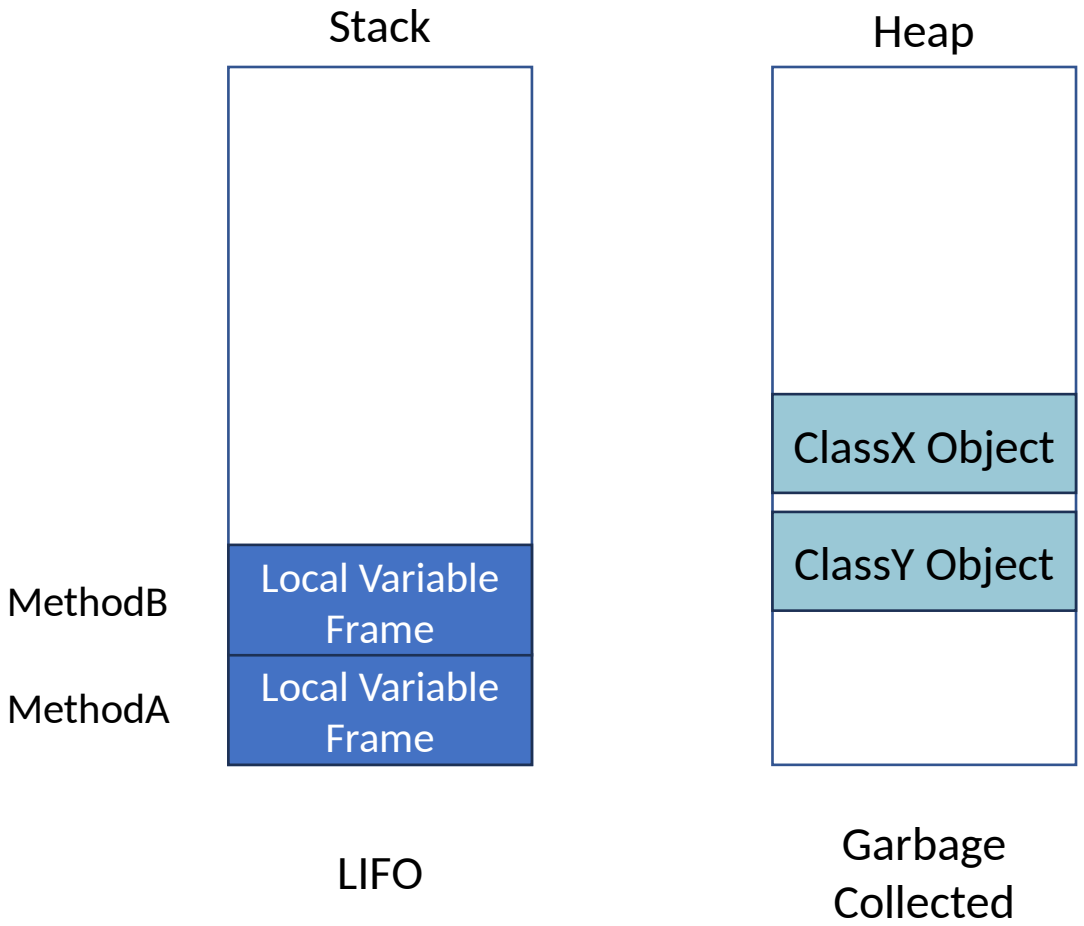
- internal can also be used as access modifier for *attributes* and *methods* (instead of private and public)
- Those members of the class would be **visible to other classes of the same assembly**

- *What is the **default** accessibility of attributes and methods when **no access modifier** is specified?*

Outline

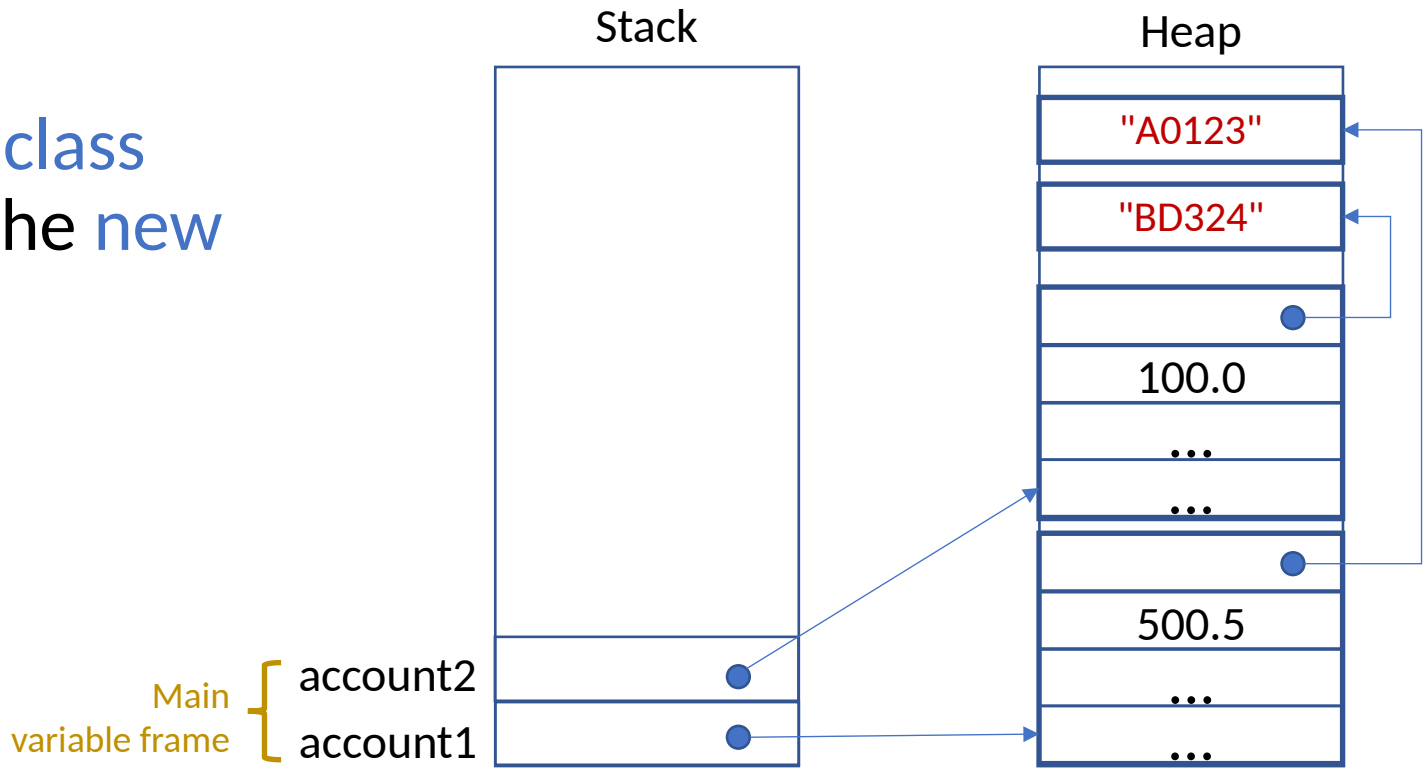
- Review of Encapsulation, C# Properties
- Namespaces and Assemblies
- Static
  - Attributes
  - Methods

Memory Management



Object instances

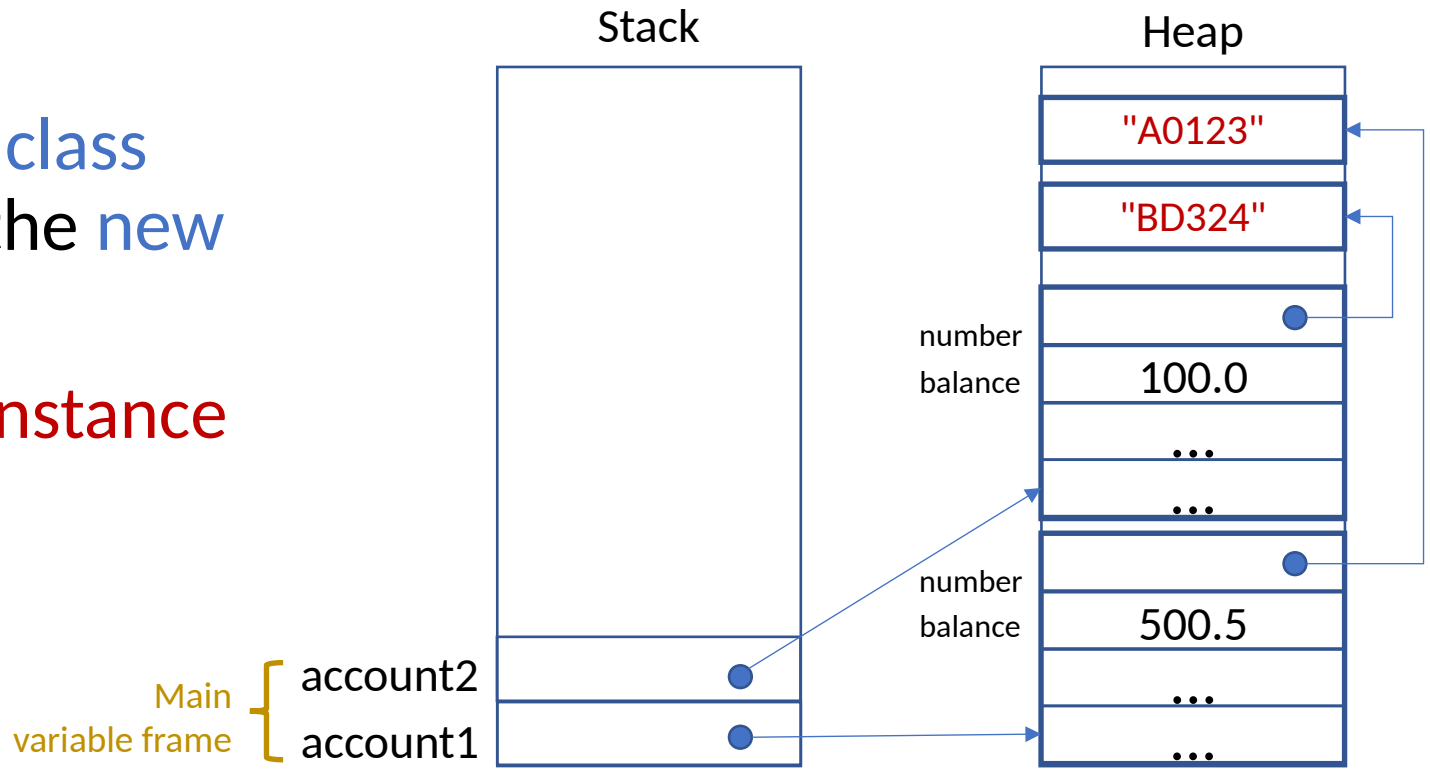
- Object *instances* of a given class are created at *runtime* via the new operator




```
class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        BankAccount  account2 = new BankAccount("BD324", 100.0);

        ...
}
```

Object instances

- Object *instances* of a given class are created at *runtime* via the new operator
- Every object has **separate** instance variables—the attributes




```
class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        BankAccount  account2 = new BankAccount("BD324", 100.0);

        ...
}
```

static keyword

- It applies to *attributes* and *methods*
- Indicates that the *attribute* or *method*
  - Is associated with a class
  - Is not tied to any specific object created from that class

Outline

- Review of Encapsulation, C# Properties
- Namespaces and Assemblies
- Static
  - Attributes
  - Methods

static attributes

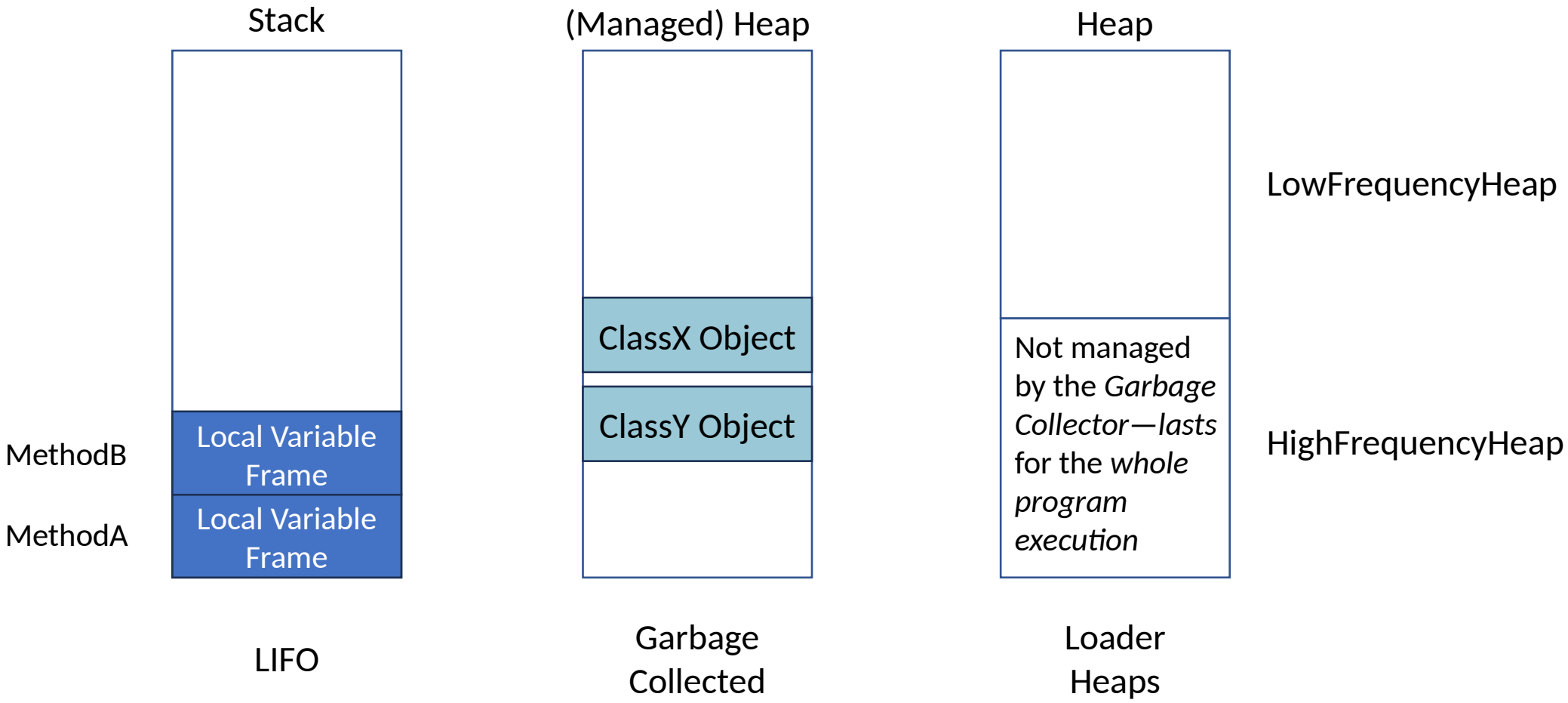
- A static *attribute* of a class is shared by *all the instances* of that class
- It *exists* regardless of objects of that class being instantiated

- Where are static attributes stored?

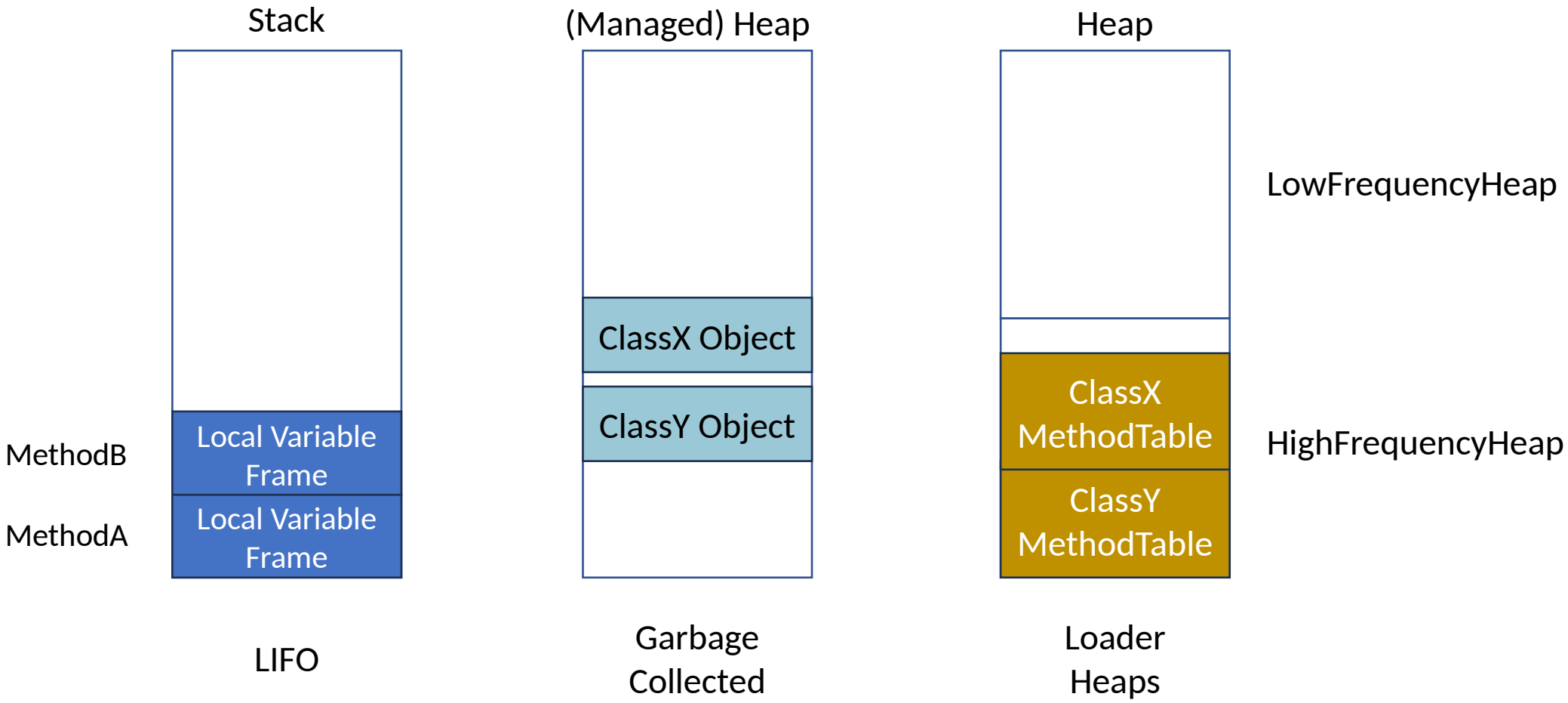
Where are the methods stored?

- We know how *value types*, *reference types* and *objects* are stored inside the *memory*
- The *CIL code* of a program is stored in *assembly files*
- How is the *CIL code*—defining the methods—executed?

Where are the methods stored?



Where are the methods stored?



The MethodTable

- Contains information (metadata) about classes *common* among objects
- References the *CIL code* of a class loaded from the *assemblies*
- References the *native code*—the methods—of a class generated from the *CIL code*

- How is it related to static attributes?

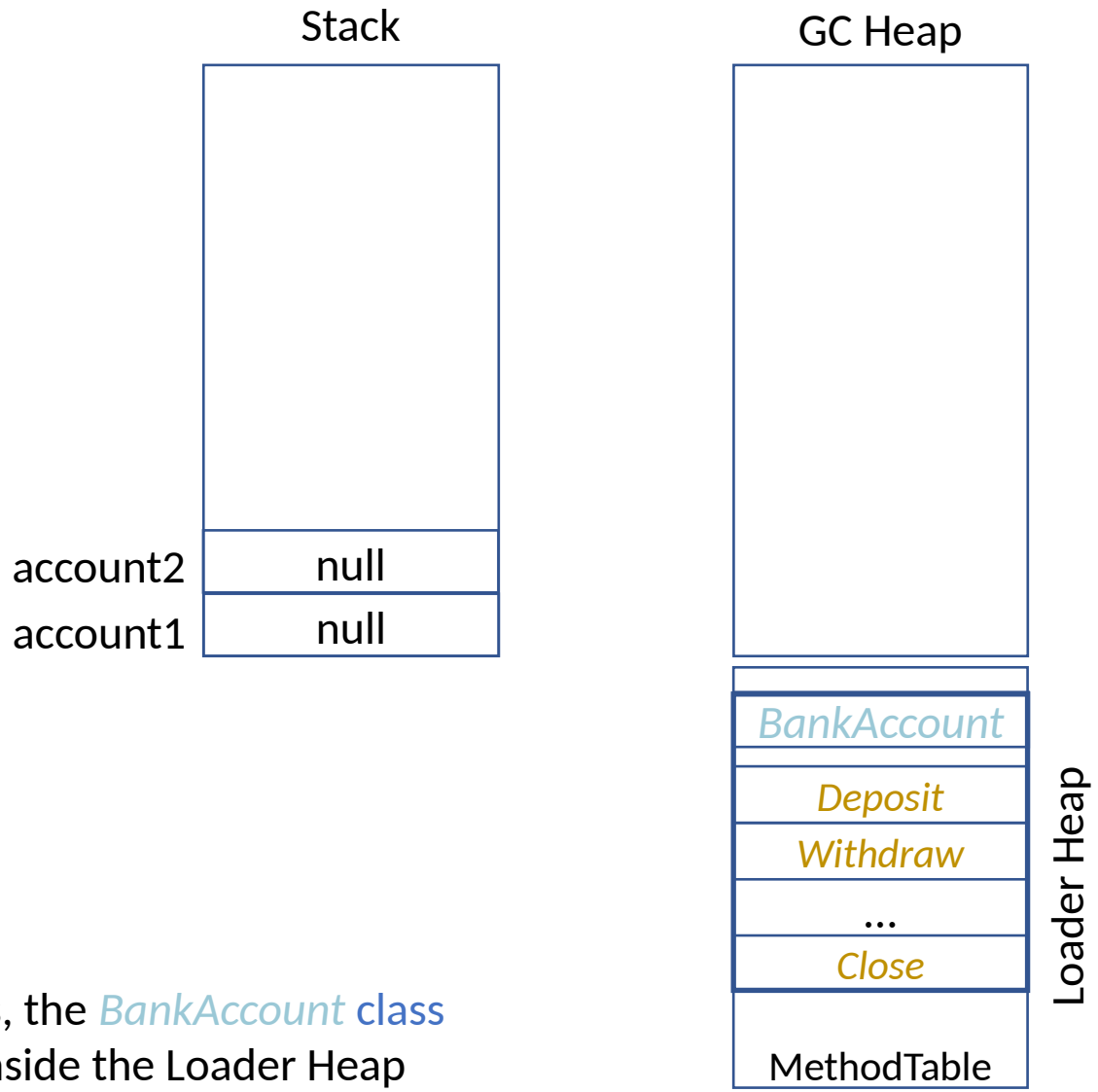
static attributes

- A static *attribute* of a class is shared by *all the instances* of that class
- It *exists* regardless of objects of that class being instantiated

- **Static attributes** are located on the *Loader Heap* inside the *MethodTable* of that class

static attributes

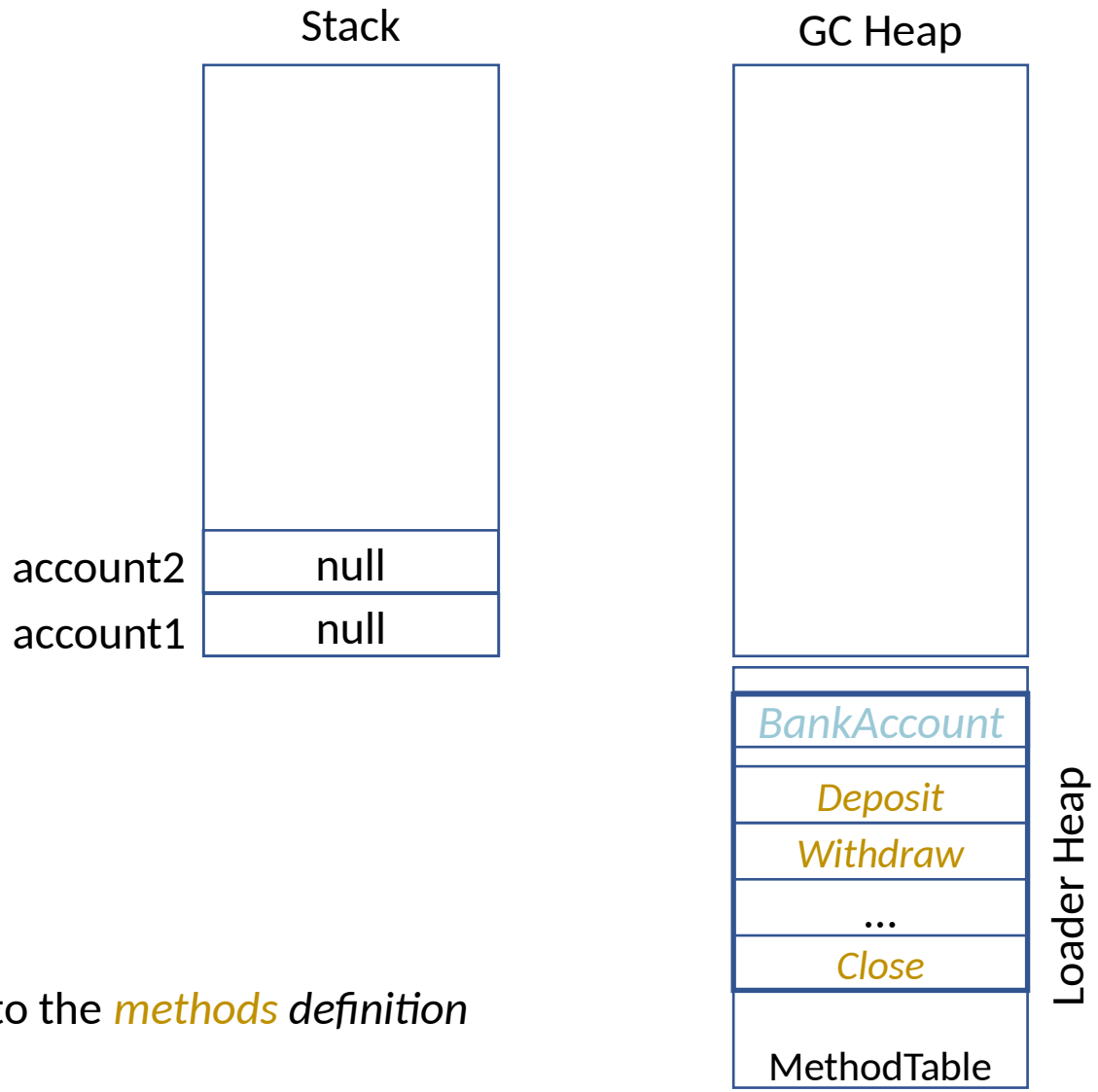
```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal)
    {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    ...
}

class Program
{
    public static void Main()
    {
        BankAccount  account1, account2;


    }
}
```



When the program starts, the *BankAccount* class *MethodTable* is **loaded** inside the Loader Heap

static attributes

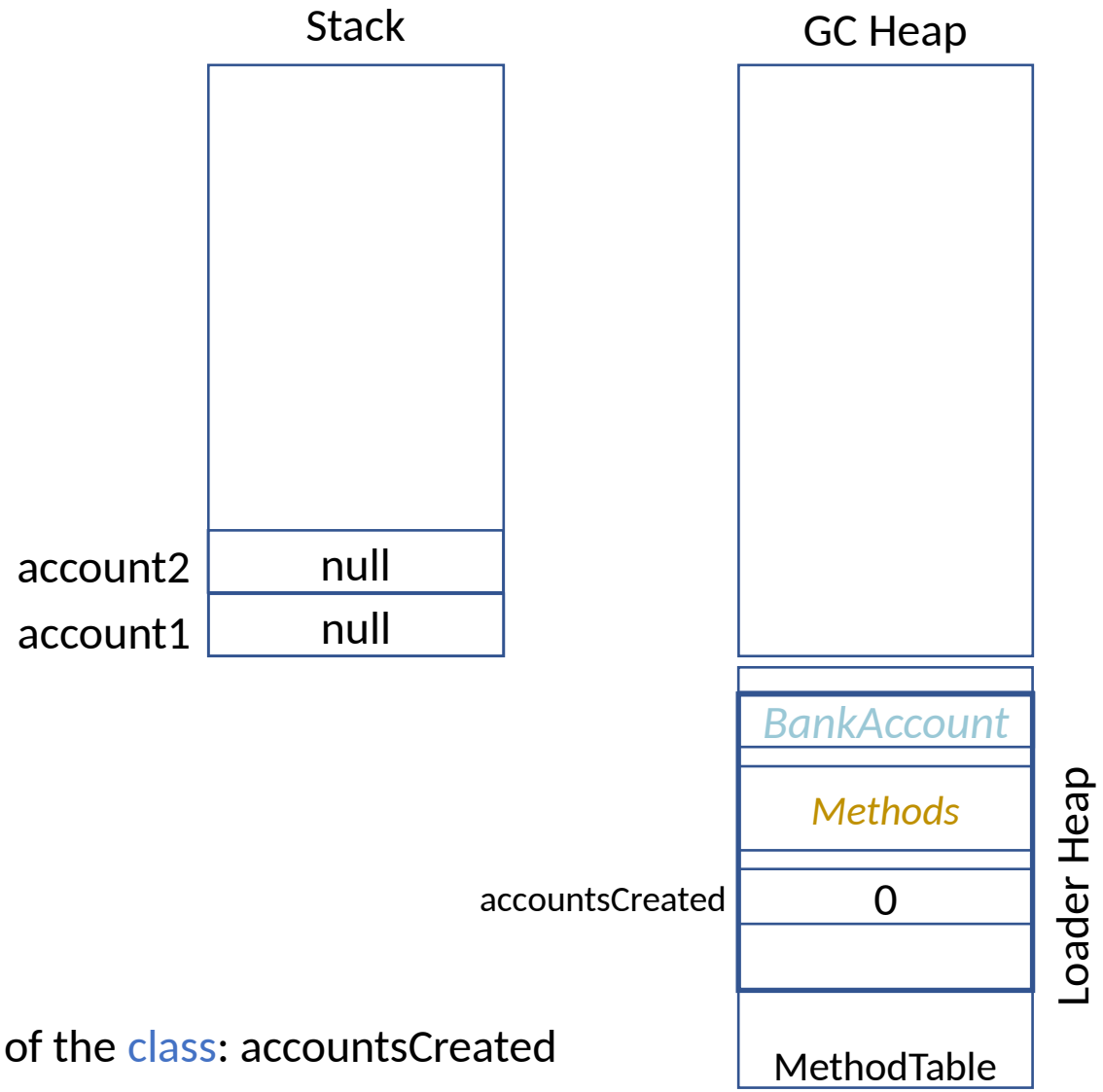
```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal)
    {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    ...
}

class Program
{
    public static void Main()
    {
        BankAccount  account1, account2;


    }
}
```



This includes references to the *methods definition* (as compiled CIL code)

static attributes

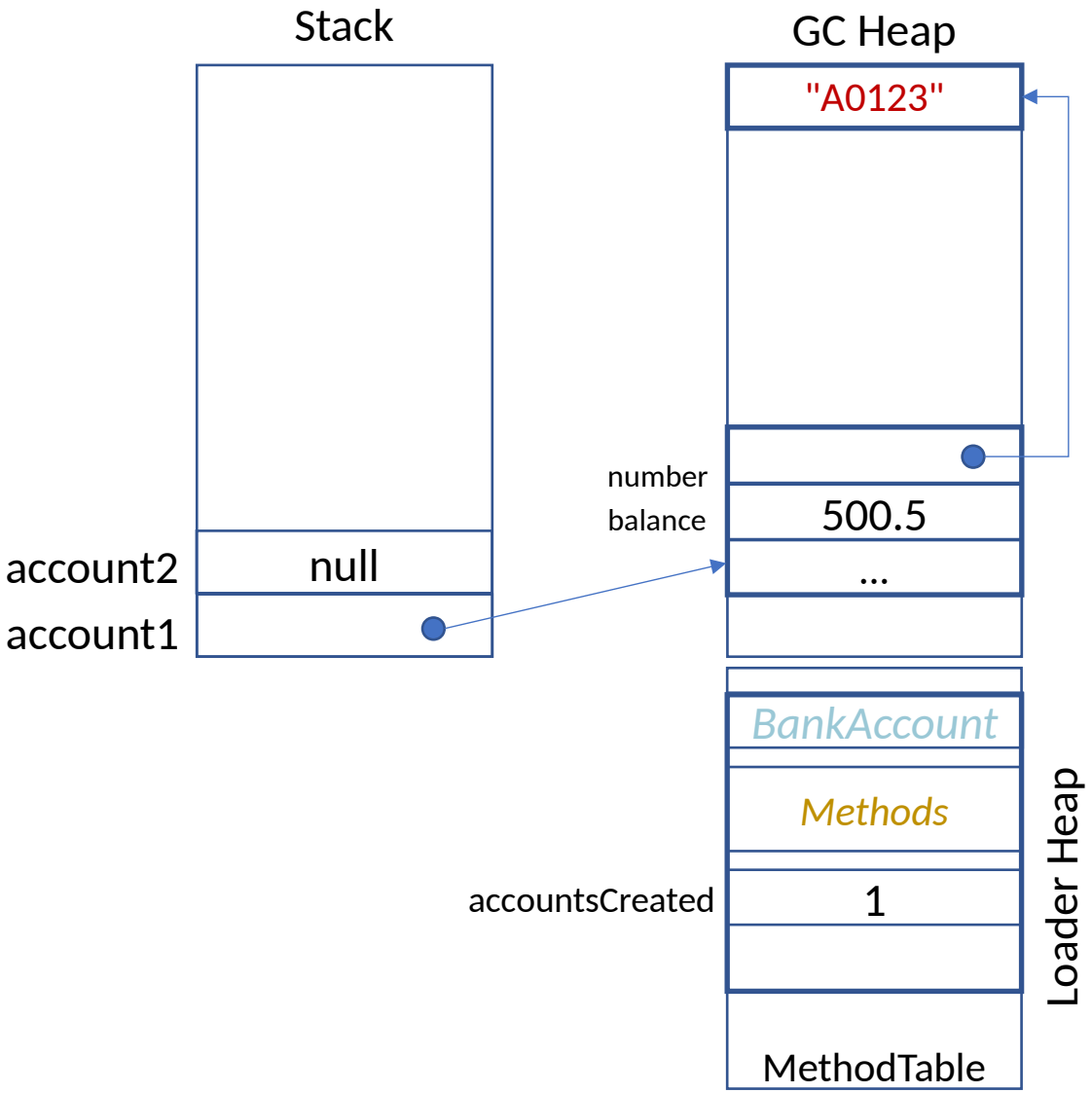
```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal)
    {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    ...
}

class Program
{
    public static void Main()
    {
        BankAccount  account1, account2;


    }
}
```



And the static *attributes* of the class: accountsCreated

static attributes

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal)
    {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    ...
}

class Program
{
    public static void Main()
    {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);

    }
}
```

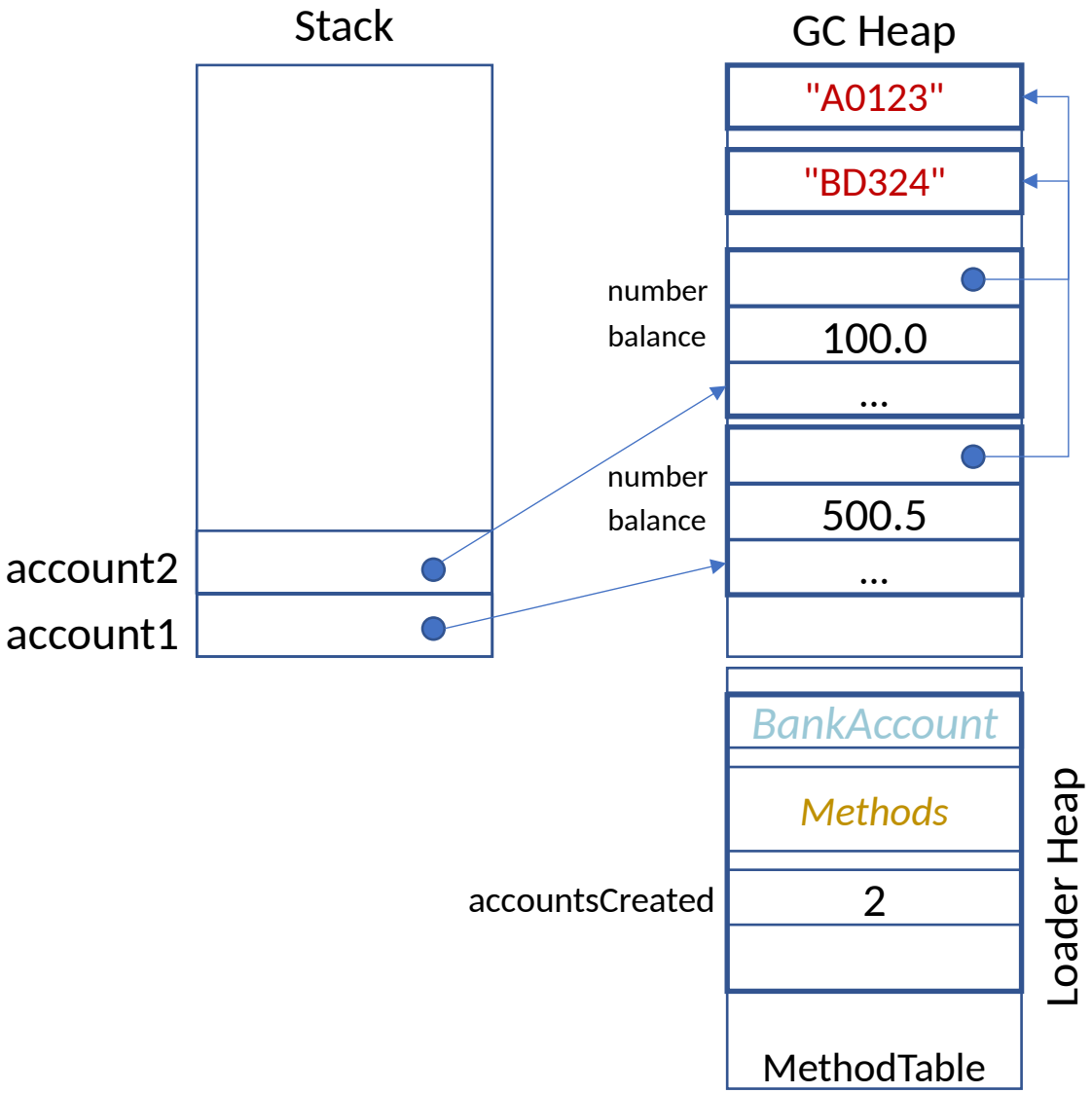


static attributes

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal)
    {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    ...
}
```

instance variables

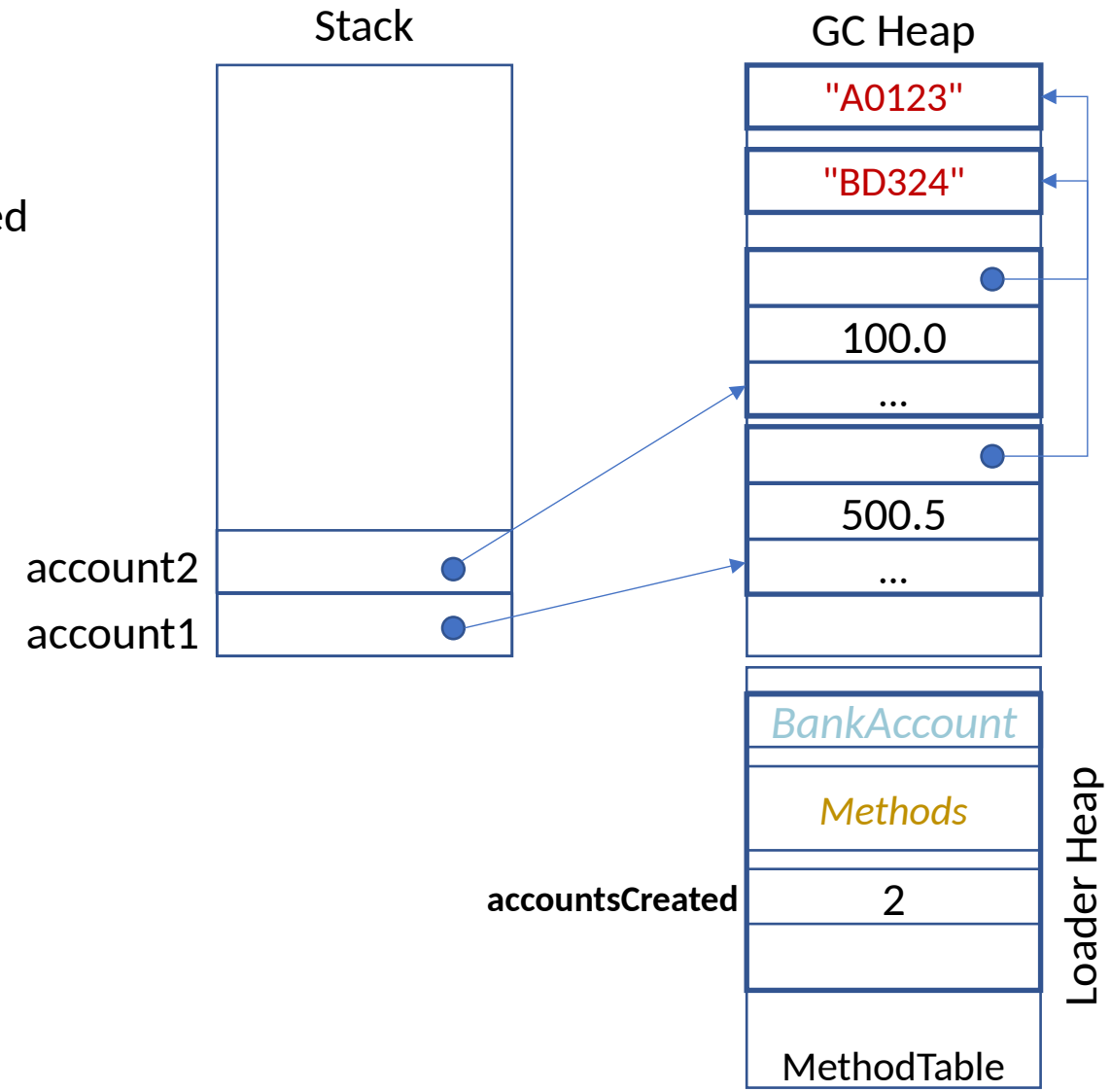
```
class Program
{
    public static void Main()
    {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        account2 = new BankAccount("BD324", 100.0);
    }
}
```



static attributes

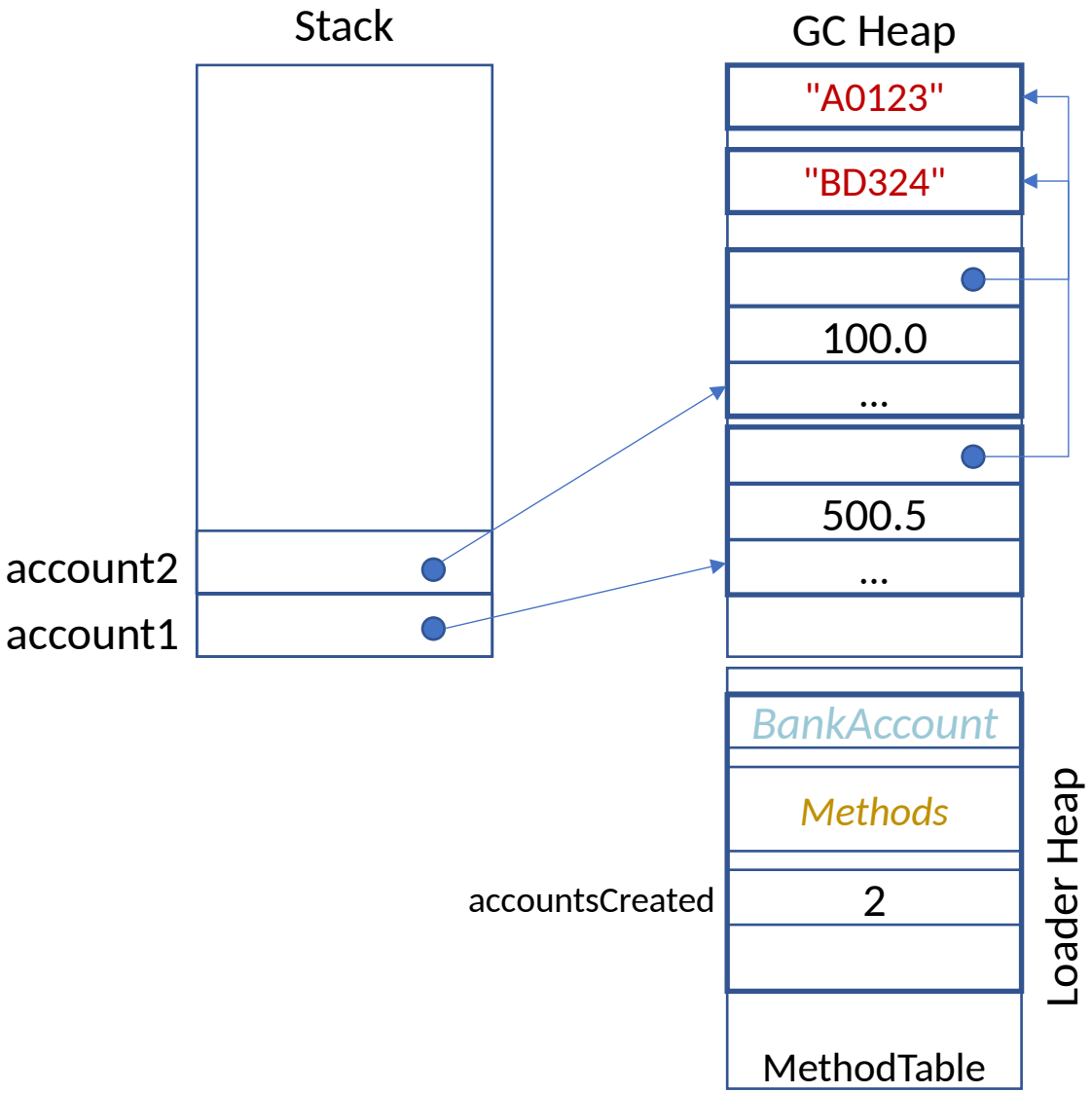
```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal)
    {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    ...
}

class Program
{
    public static void Main()
    {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        account2 = new BankAccount("BD324", 100.0);
    }
}
```

shared



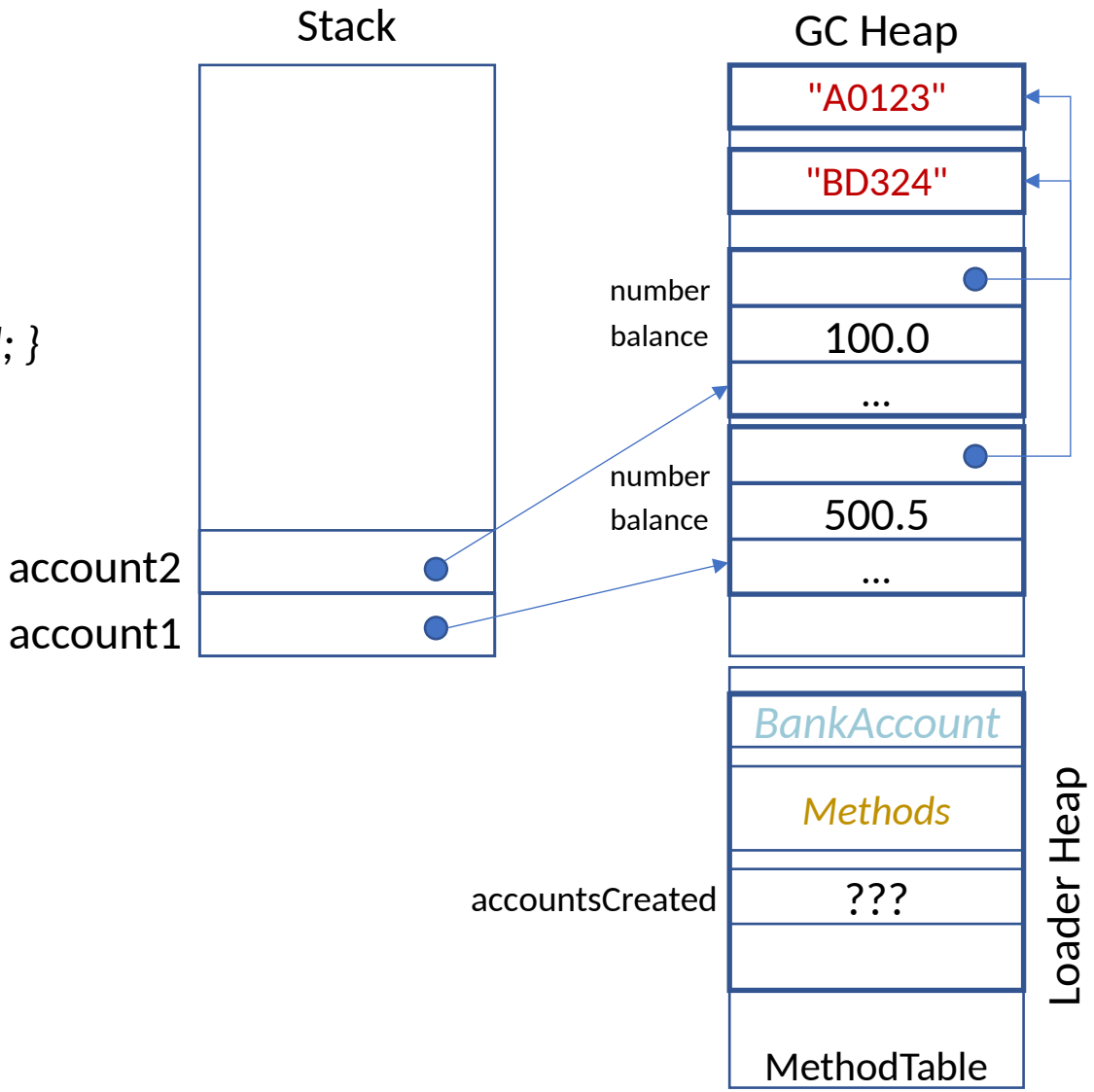
static attributes

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public int GetAccountsCreated ()
    {
        return accountsCreated;
    }
}


class Program
{
    public static void Main()
    {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        account2 = new BankAccount("BD324", 100.0);
    }
}
```



static attributes

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }
    public int GetAccountsCreated () { return accountsCreated; }
    ...
}

class Program
{
    public static void Main()
    {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        Console.WriteLine(account1.GetAccountsCreated());

        account2 = new BankAccount("BD324", 100.0);
        Console.WriteLine(account2.GetAccountsCreated());

        Console.WriteLine(account1.GetAccountsCreated());
    }
}
```



static attributes

- A static *attribute* of a class is shared by *all the instances* of that class
- It *exists* regardless of objects of that class being instantiated

- It is *located* on the *Loader Heap* inside the *MethodTable* of that class
- Its content is *not specific to an object*

Outline

- Review of Encapsulation, C# Properties
- Namespaces and Assemblies
- Static
  - Attributes
  - Methods

static methods

- static methods are attached to a class—can be called **without** referring to an object *instance* of that class

- **Do not** operate on *instance variables* (attributes) of an object
- **Use** other static attributes and methods of the same class or exposed by other classes

static methods

- We have already used static methods
  - *Math*.*Sqrt*(…)
  - *Console*.*WriteLine*(…)
  - *Main*(string[] args)

- *Console* is the class name (static – cannot be instantiated)
- *WriteLine* is a static void method of the static *Console* class

static methods

- static methods are attached to a class—can be called **without** referring to an object *instance* of that class

- **Do not** operate on *instance variables* (attributes) of an object
- **Use** other static attributes and methods of the same class or exposed by other classes

static methods

```
public class CalcManager
{
    public bool IsEven(int n)
    {
      if (n % 2 == 0)
         return true;
      else
         return false;
    }

    public int Cube(int n)
    {
      return n * n * n;
    }

    public double Add(double[] x)
    {
      double sum = 0.0;
      foreach (double e in x)
         sum = sum + e;
      return sum;
    }
}
```

What would be the state of the objects of the *CalcManager* class?

static methods

```
public class CalcManager
{   // no attributes defined!
     public bool IsEven(int n)
     {
       if (n % 2 == 0)
          return true;
       else
          return false;
    }

    public int Cube(int n)
    {
       return n * n * n;
    }

    public double Add(double[] x)
    {
       double sum = 0.0;
       foreach (double e in x)
          sum = sum + e;
       return sum;
    }
}
```

CalcManager does not define any attributes – **no state**

static methods

```
public class CalcManager
{   // no attributes defined!
     public bool IsEven(int n)
     {
       if (n % 2 == 0)
          return true;
       else
          return false;
    }

    public int Cube(int n)
    {
       return n * n * n;
    }

    public double Add(double[] x)
    {
       double sum = 0.0;
       foreach (double e in x)
          sum = sum + e;
       return sum;
    }
}
```

The methods will *not depend on the attributes*

static methods

```
public class CalcManager
{
   public bool IsEven(int n)
    {
      if (n % 2 == 0)
         return true;
      else
         return false;
   }

   public int Cube(int n)
   {
      return n * n * n;
   }

   public double Add(double[] x)
   {
      double sum = 0.0;
      foreach (double e in x)
         sum = sum + e;
      return sum;
   }
}
```

They perform operations on their *parameters* and return a value: *utility methods*

static methods

```
public class CalcManager
{
    public static bool IsEven(int n)
     {
       if (n % 2 == 0)
          return true;
       else
          return false;
    }

    public static int Cube(int n)
    {
       return n * n * n;
    }

    public static double Add(double[] x)
    {
       double sum = 0.0;
       foreach (double e in x)
          sum = sum + e;
       return sum;
    }
}
```

All the methods can be declared as static—they do not need to access objects' attributes

static methods

```
public static class CalcManager
{
   public static bool IsEven(int n)
    {
      if (n % 2 == 0)
         return true;
      else
         return false;
   }

   public static int Cube(int n)
   {
      return n * n * n;
   }

   public static double Add(double[] x)
   {
      double sum = 0.0;
      foreach (double e in x)
         sum = sum + e;
      return sum;
   }
}
```

A class that contains only static methods can also be declared as static

No **object instances** of that class can ever be created

static methods

```
public static class CalcManager
{
    public static bool IsEven(int n)
     {
       if (n % 2 == 0)
          return true;
       else
          return false;
    }

    public static int Cube(int n)
    {
       return n * n * n;
    }

    public static double Add(double[] x)
    {
       double sum = 0.0;
       foreach (double e in x)
          sum = sum + e;
       return sum;
    }
}
```

```
class Program
{
    public static void Main()
    {
       int number = 3;
       double[] values = { 0.4, 3.5, 7.8, 0.5 };

       Console.WriteLine(CalcManager.IsEven(number));
       Console.WriteLine(CalcManager.Add(values));
    }
}
```

no need to instantiate a CalcManager object

the static methods of CalcManager can be invoked by using CalcManager.Method by methods of other classes

static methods

```
public static class CalcManager
{
   public static bool IsEven(int n)
    {
      if (n % 2 == 0)
         return true;
      else
         return false;
   }

   public static int Cube(int n)
   {
      return n * n * n;
   }

   public static double Add(double[] x)
   {
      double sum = 0.0;
      foreach (double e in x)
         sum = sum + e;
      return sum;
   }

   public static void Main() {
      Console.WriteLine(IsEven(10));
   }
}
```

A static method can be invoked directly (by name) from other methods (e.g., *Main*) defined inside the class *CalcManager*

static methods

- static methods are attached to a class—can be called **without** referring to an object *instance* of that class

- **Do not** operate on *instance variables* (attributes) of an object
- **Use** other static attributes and methods of the same class or exposed by other classes

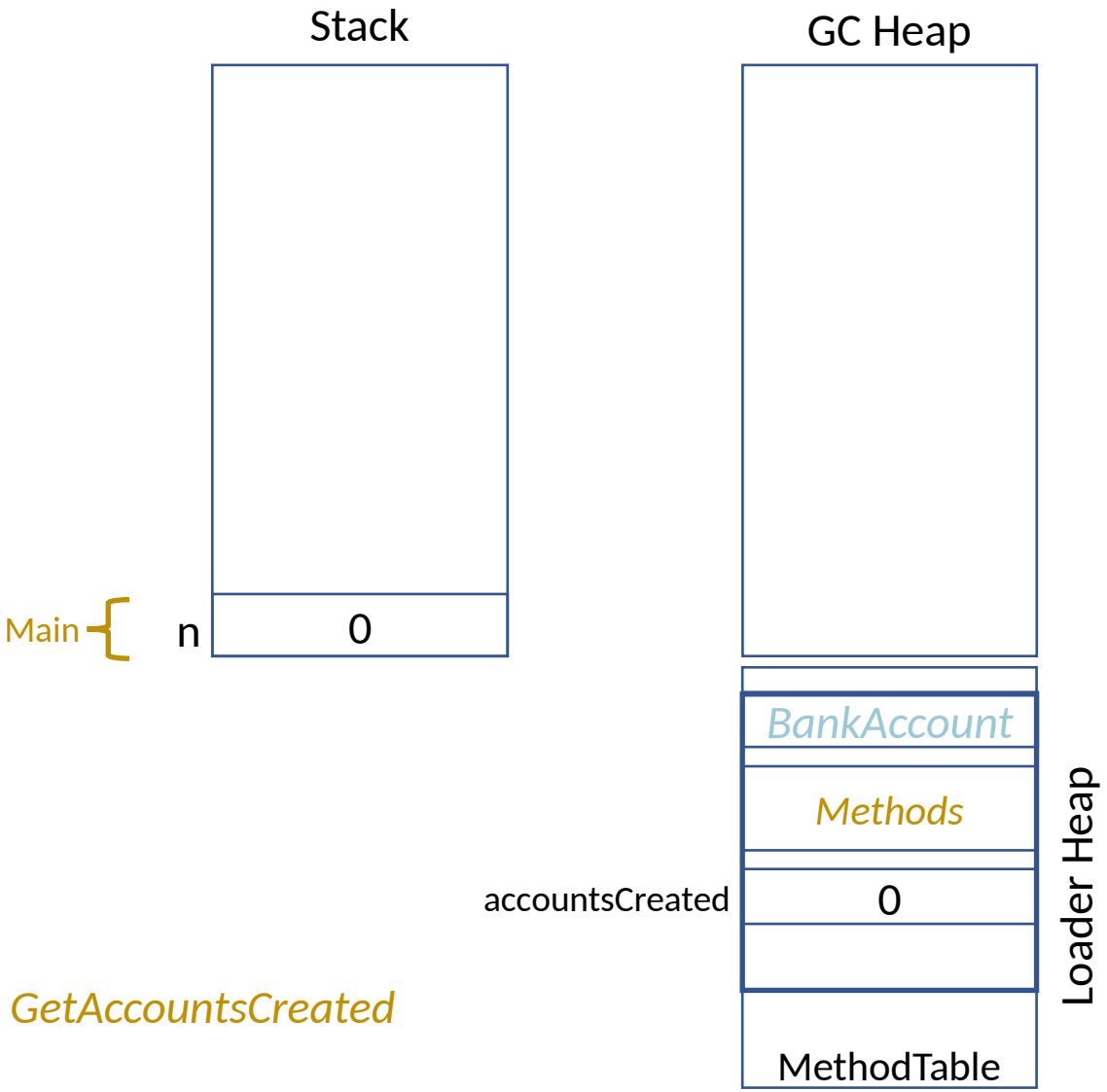
Question

- Should *GetAccountsCreated* be static?

static method invocation

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public static int GetAccountsCreated()
    {
        return accountsCreated;
    }

    // other methods of BankAccount
}


class Program
{
    public static void Main()
    {
        int n = BankAccount.GetAccountsCreated();

    }
}
```

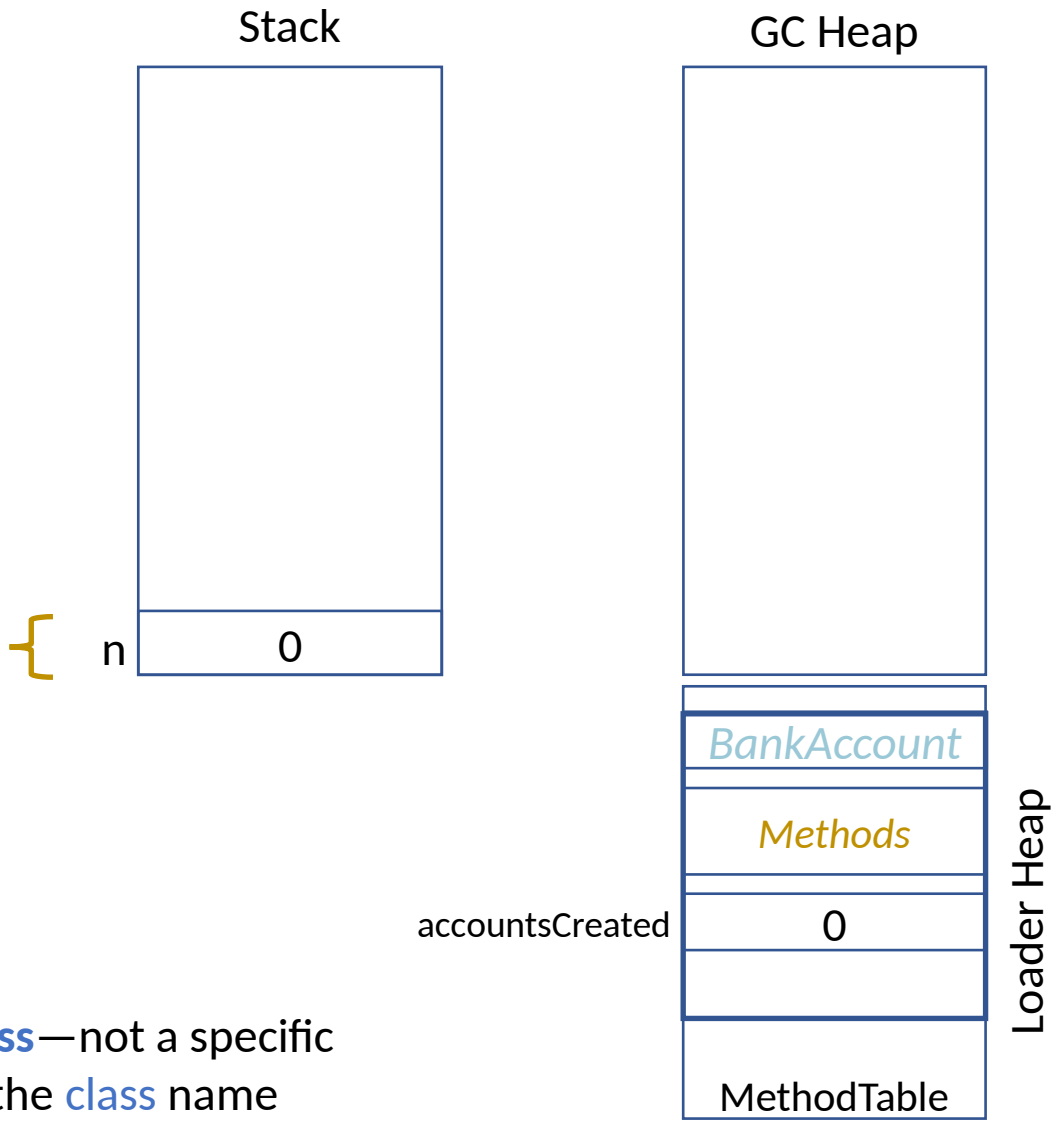
let's now define the method *GetAccountsCreated* as static



static method invocation

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public static int GetAccountsCreated()
    {
        return accountsCreated;
    }

    // other methods of BankAccount
}


class Program
{
    public static void Main()
    {
        int n = BankAccount.GetAccountsCreated();

    }
}
```

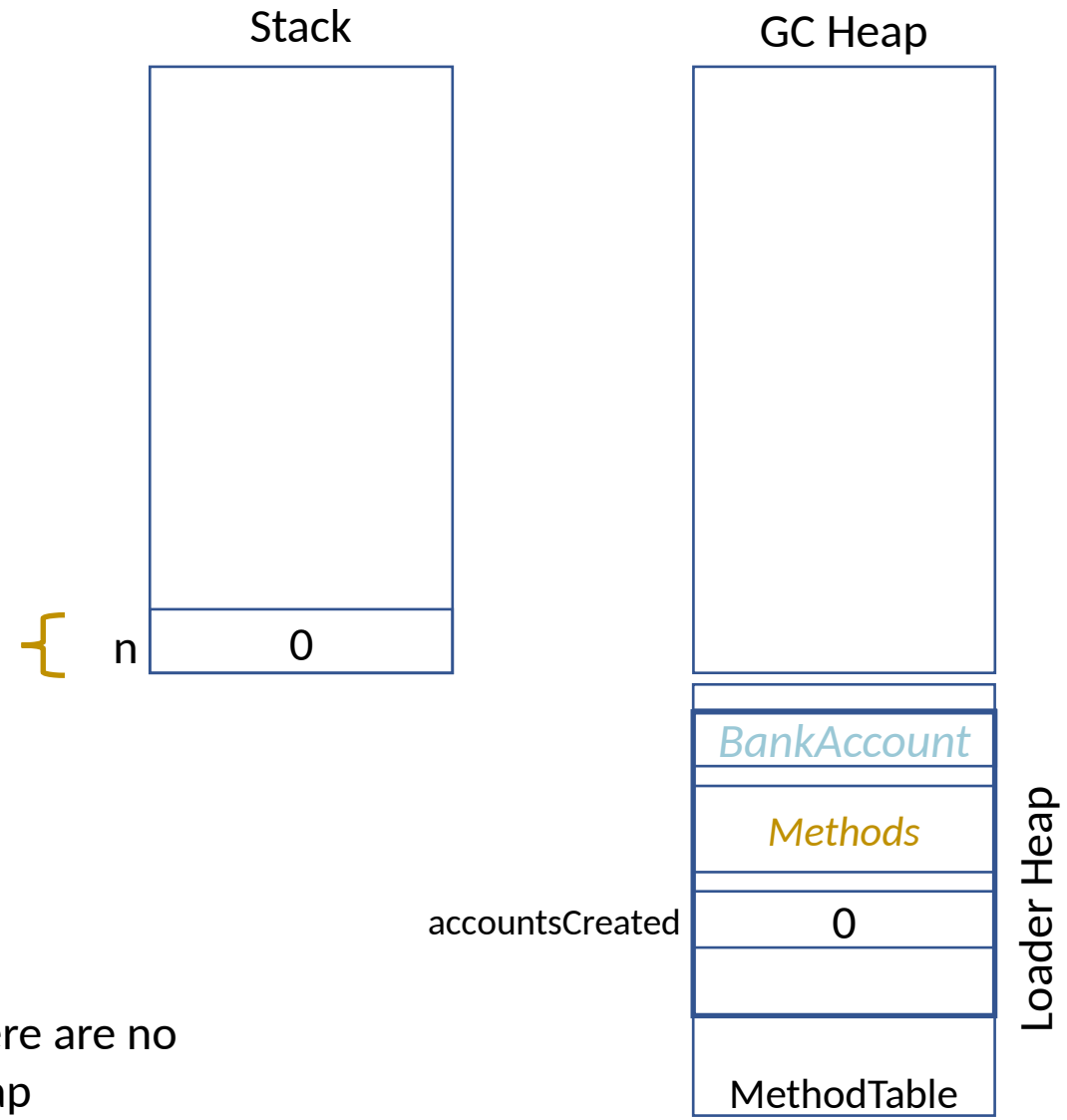
it is now **associated with the class**—not a specific **object**—and can be invoked via the class name



static method invocation

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public static int GetAccountsCreated()
    {
        return accountsCreated;
    }

    // other methods of BankAccount
}


class Program
{
    public static void Main()
    {
        int n = BankAccount.GetAccountsCreated();

    }
}
```

it can be called even though there are no *BankAccount* objects on the heap



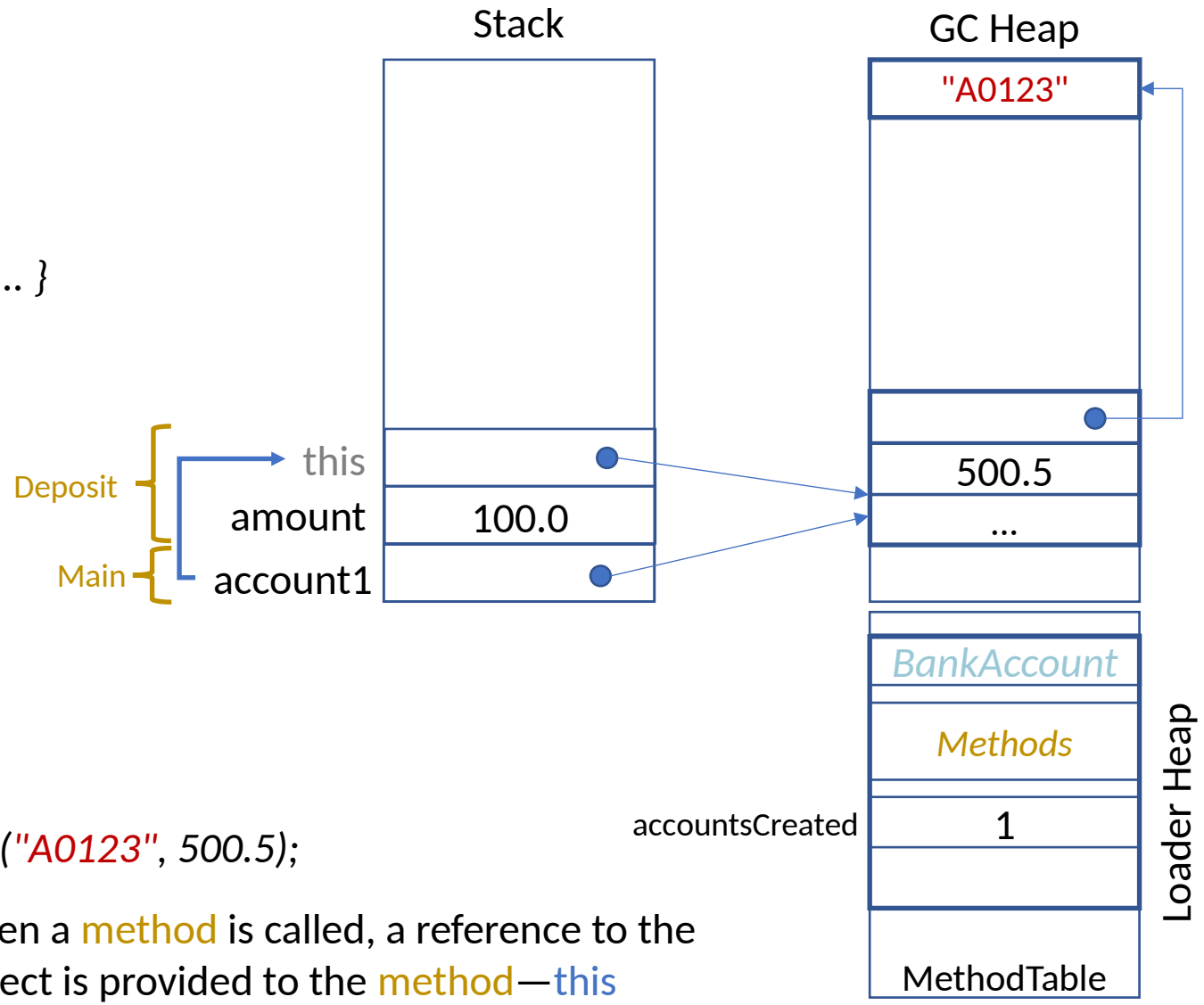
static method invocation: Question

```
class BankAccount
{
   private string number;
   private double balance;
   private static int accountsCreated = 0;

   public BankAccount(string num, double bal) { ... }

   public static int GetAccountsCreated()
   {
      Console.WriteLine(balance);
      return accountsCreated;
   }

   // other methods of BankAccount
}

class Program
{
   public static void Main()
   {
      int n = BankAccount.GetAccountsCreated();

   }
}
```

can a static method access the *instance* attribute balance?

Instance method invocation

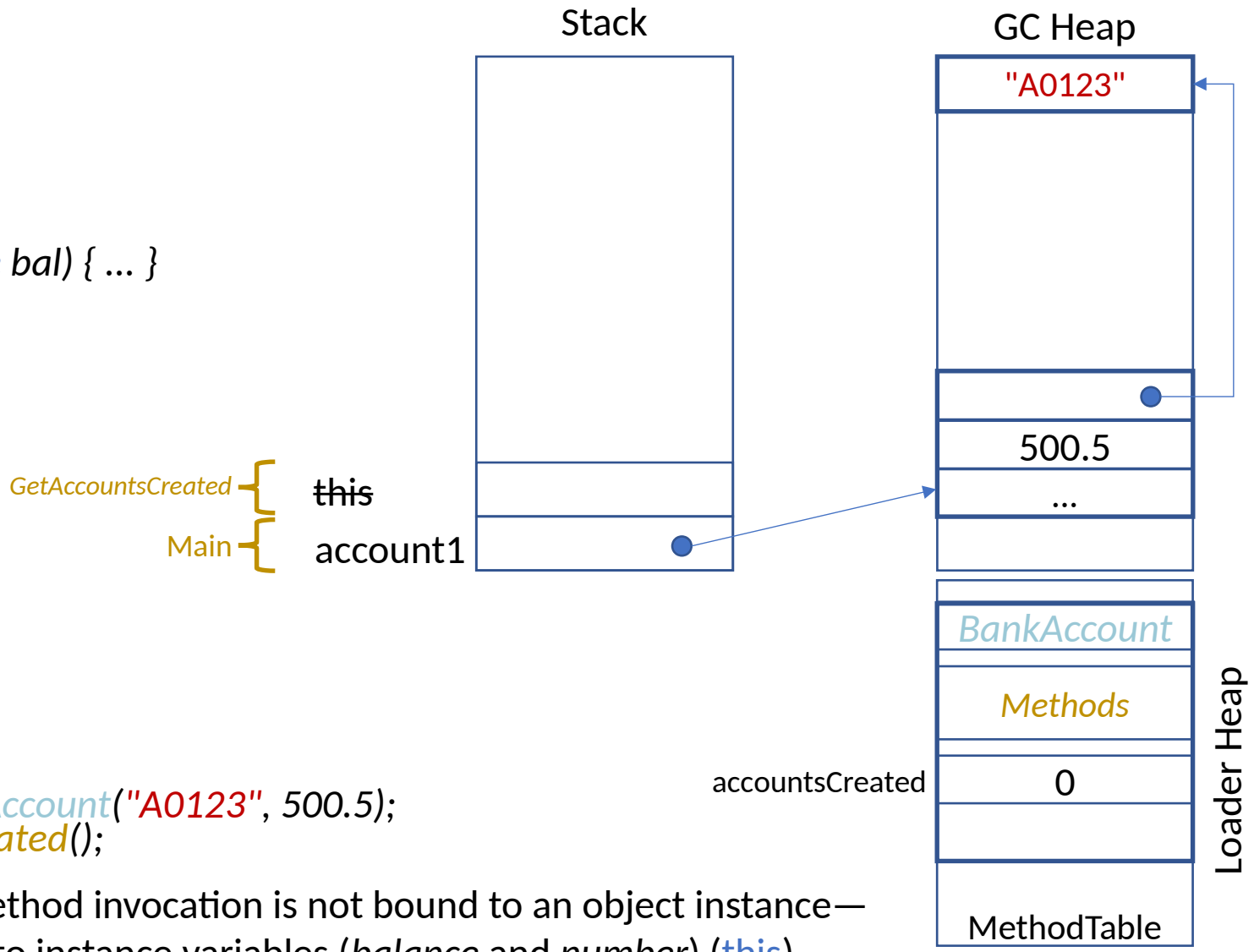
```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public static int GetAccountsCreated()
    {
        Console.WriteLine(balance);
        return accountsCreated;
    }

    // other methods of BankAccount
}

class Program
{
    public static void Main(string[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(100.0);
    }
}
```



when a method is called, a reference to the object is provided to the method—this

static method invocation

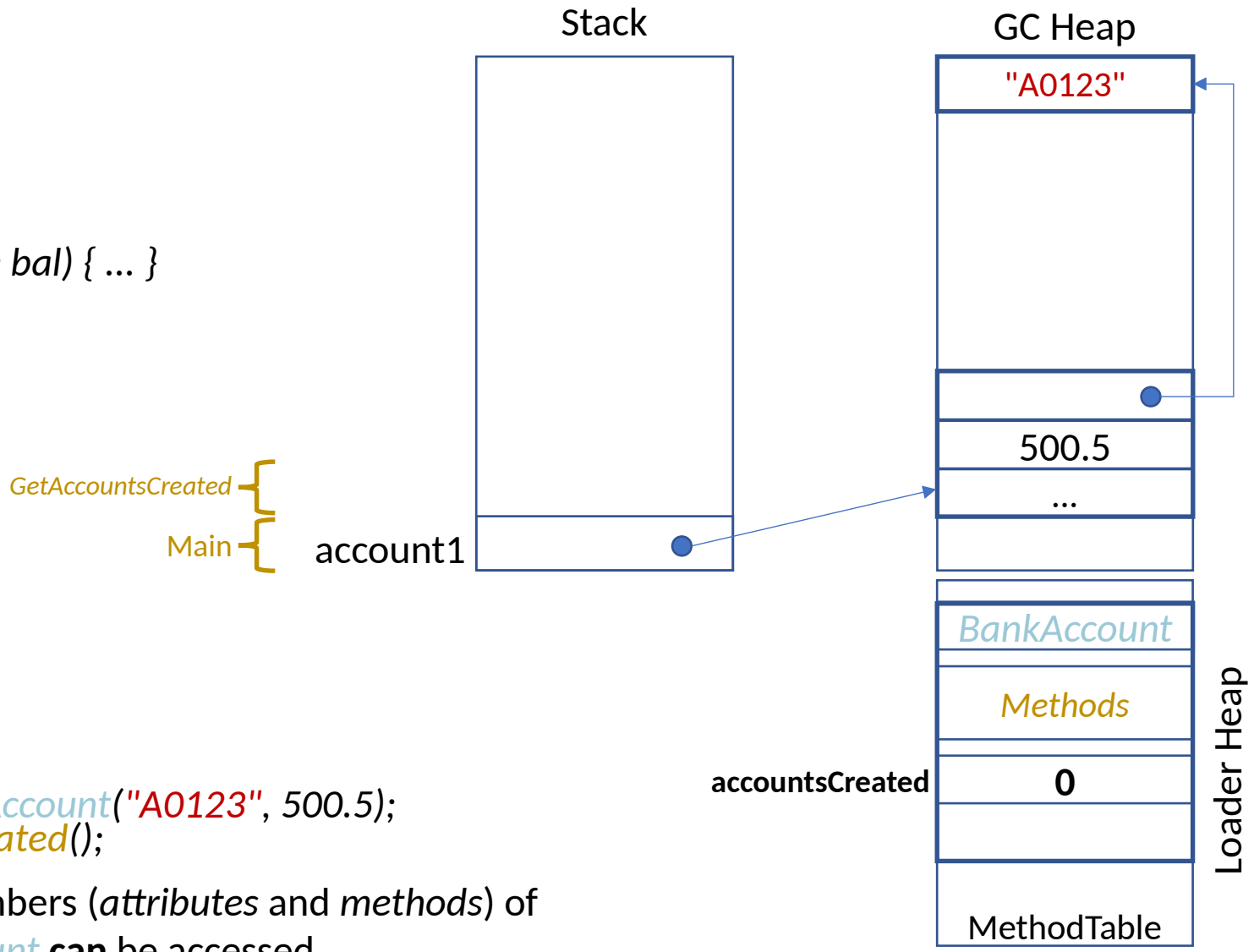
```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public static int GetAccountsCreated()
    {
        Console.WriteLine(balance);
        return accountsCreated;
    }

    // other methods of BankAccount
}

class Program
{
    public static void Main(string[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        int n = BankAccount.GetAccountsCreated();
    }
}
```



A static method invocation is not bound to an object instance—no access to instance variables (*balance* and *number*) (~~this~~)

static method invocation

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public static int GetAccountsCreated()
    {
        Console.WriteLine(balance);
        return accountsCreated;
    }

    // other methods of BankAccount
}

class Program
{
    public static void Main(string[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        int n = BankAccount.GetAccountsCreated();

    }
}
```



static members (*attributes* and *methods*) of *BankAccount* **can** be accessed

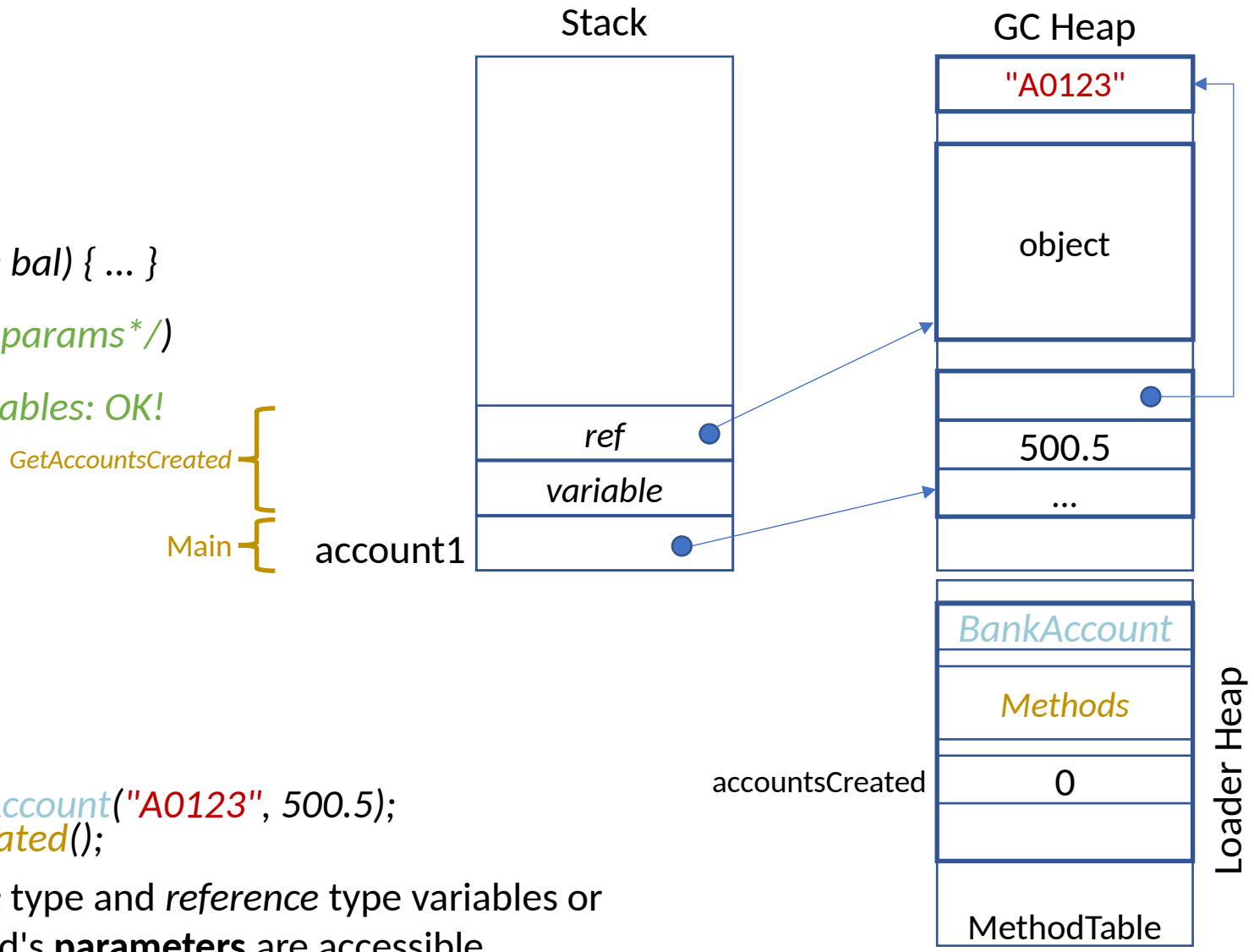
static method invocation: Answer

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public static int GetAccountsCreated()
    {
        Console.WriteLine(balance);
        return accountsCreated;
    }

    // other methods of BankAccount
}

class Program
{
    public static void Main(string[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        int n = BankAccount.GetAccountsCreated();

    }
}
```

no, it would generate a compiler error!

static method invocation

```
class BankAccount
{
    private string number;
    private double balance;
    private static int accountsCreated = 0;

    public BankAccount(string num, double bal) { ... }

    public static int GetAccountsCreated(/*params*/)
    {
        // local value and reference type variables: OK!
        return accountsCreated;
    }

    // other methods of BankAccount
}

class Program
{
    public static void Main(string[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        int n = BankAccount.GetAccountsCreated();

    }
}
```



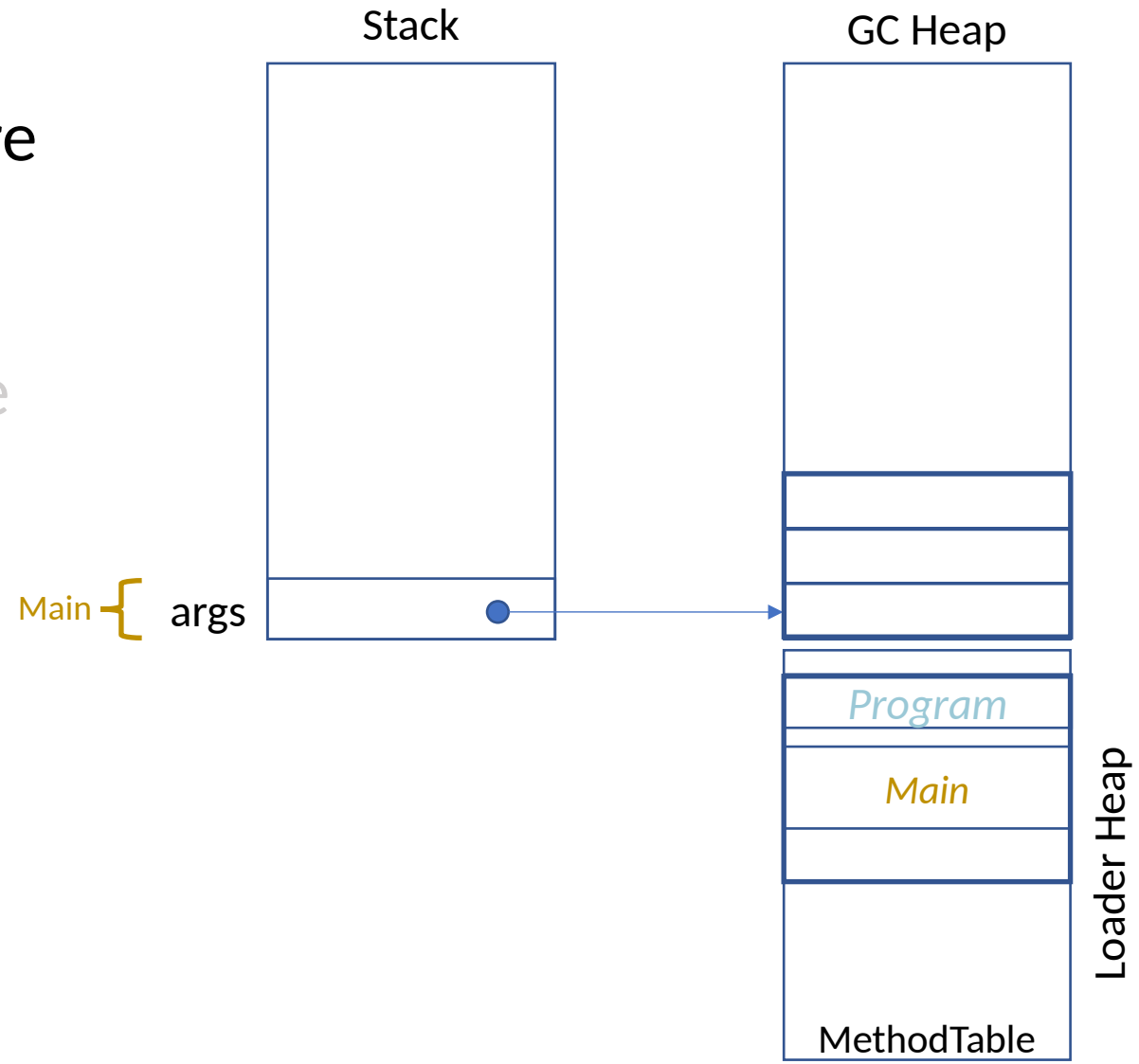
local *value* type and *reference* type variables or the method's **parameters** are accessible

static context: Main

- Every C# program must have a class that includes the static Main
- This is a convention to identify the **entry-point** of the program
- Why is the Main static?

static context: Main

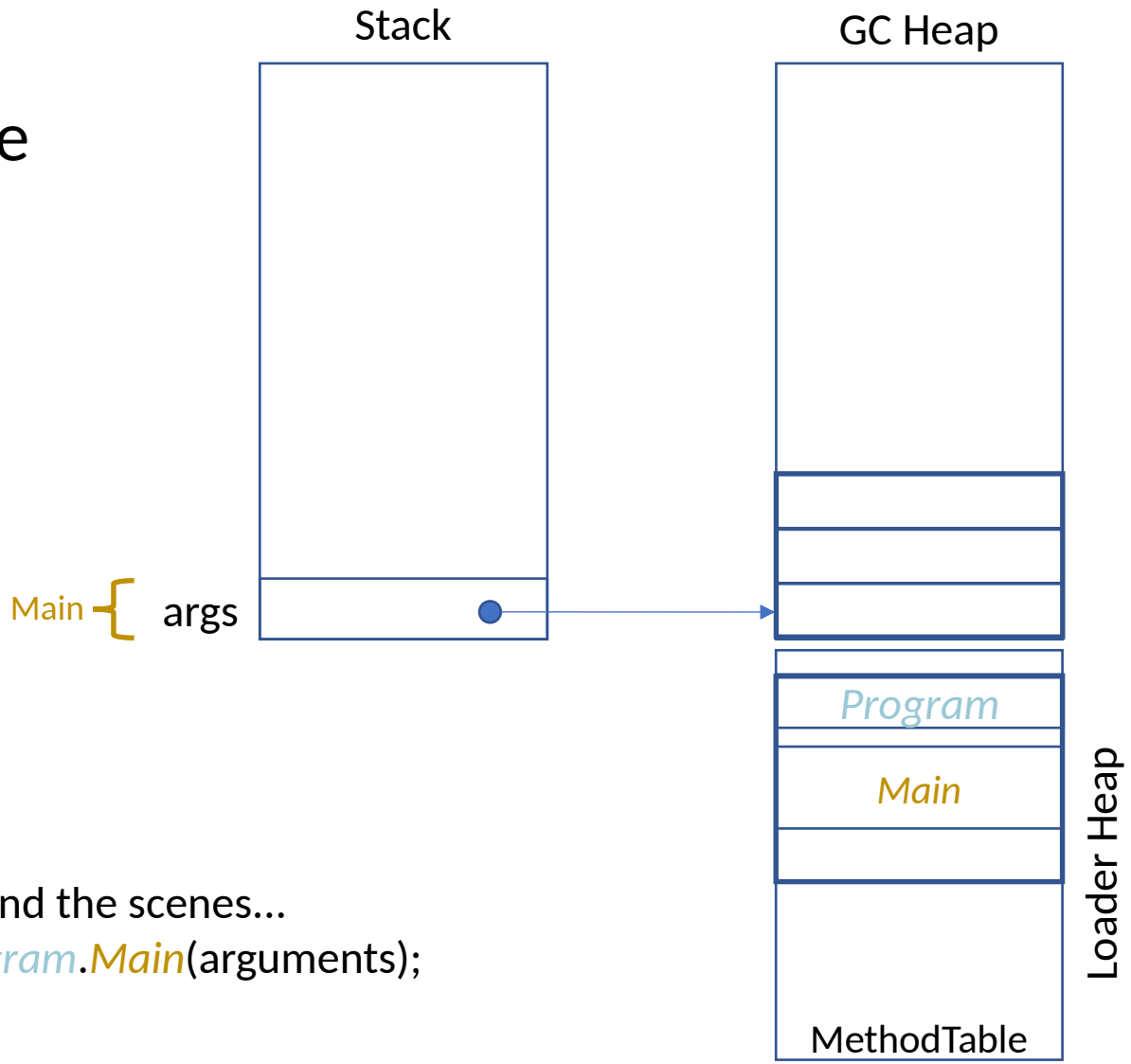
- When the program starts, there are no objects of the class where the Main is defined
- By design, the .NET CLR can invoke the Main without referring to an object instance of that class




```
class Program
{
    public static void Main(string[] args)
    {

    }
}
```

static context: Main

- When the program starts, there are no objects of the class where the Main is defined
- By design, the .NET CLR can invoke the Main without referring to an object instance of that class




```
class Program
{
    public static void Main(string[] args)
    {

    }
}
```

behind the scenes...
Program.*Main*(arguments);