# 7SENG011W Object Oriented Programming

*More on Value Types and Reference Types, 'this' keyword, more on Encapsulation*

**Dr Francesco Tusa**

# Readings

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

- Programming C# 10
  - Chapter 3: Types
- Object-Oriented Thought Process
  - Chapters 1-5

**Online**

- Passing Parameters
- Garbage Collection
- Passing Value Types by Value
- Passing Reference Types by Value
- this keyword
- Methods Overload
- C# Access Modifiers

# Outline

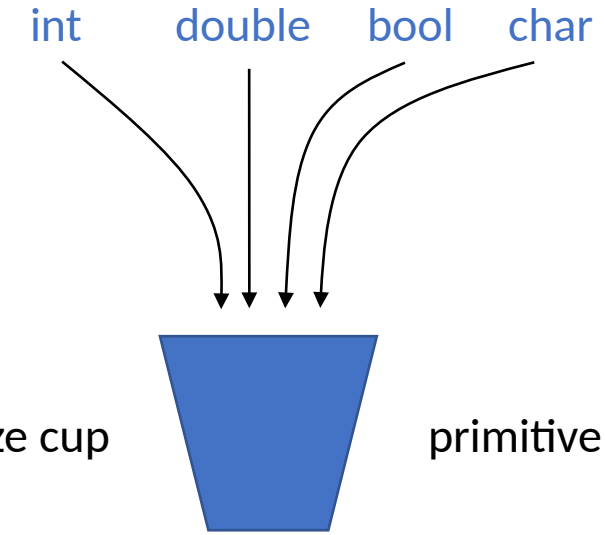
- More on *Value Types* and *Reference Types*
  - '=' and '==' operators
  - Method invocation and parameter passing
- The *this* keyword
- More on *Encapsulation*

# Outline

- More on *Value Types* and *Reference Types*
  - '=' and '==' operators
  - Method invocation and parameter passing
- The *this* keyword
- More on *Encapsulation*

# How primitive types are stored

- **Primitive** types are basic C# types:
  - int, double, bool, char, etc. – the variable **contains its data**
- Have a well-defined standard size (between 8-64 bits)

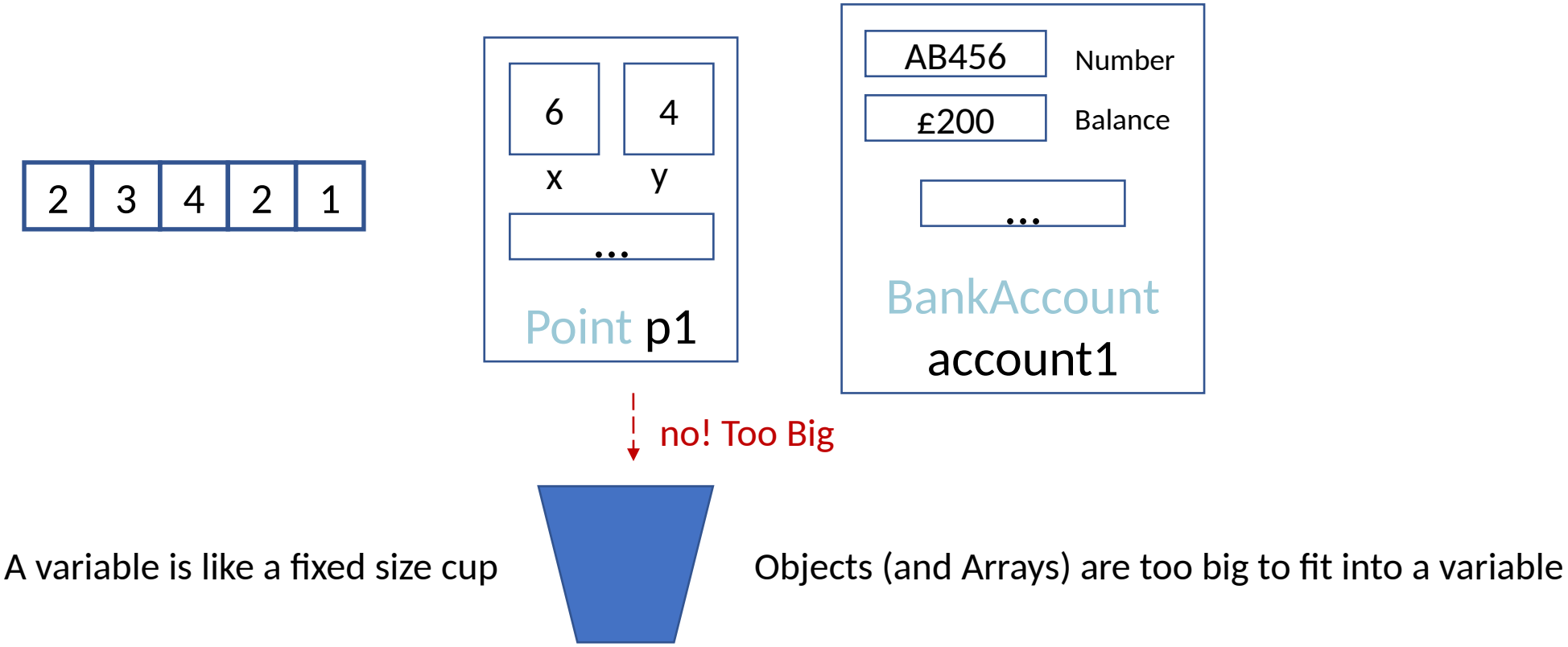



A variable is like a fixed size cup

primitive are small enough to fit into that cup

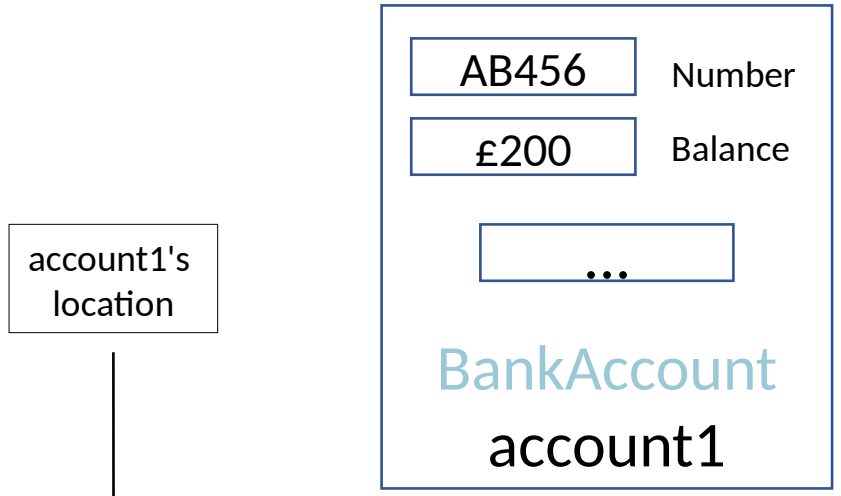
# How reference types are stored

- **Reference** types are *arrays* and *objects*:
  - string, int[], double[], *Point*, *Circle*, *BankAccount*, etc.



# How reference types are stored

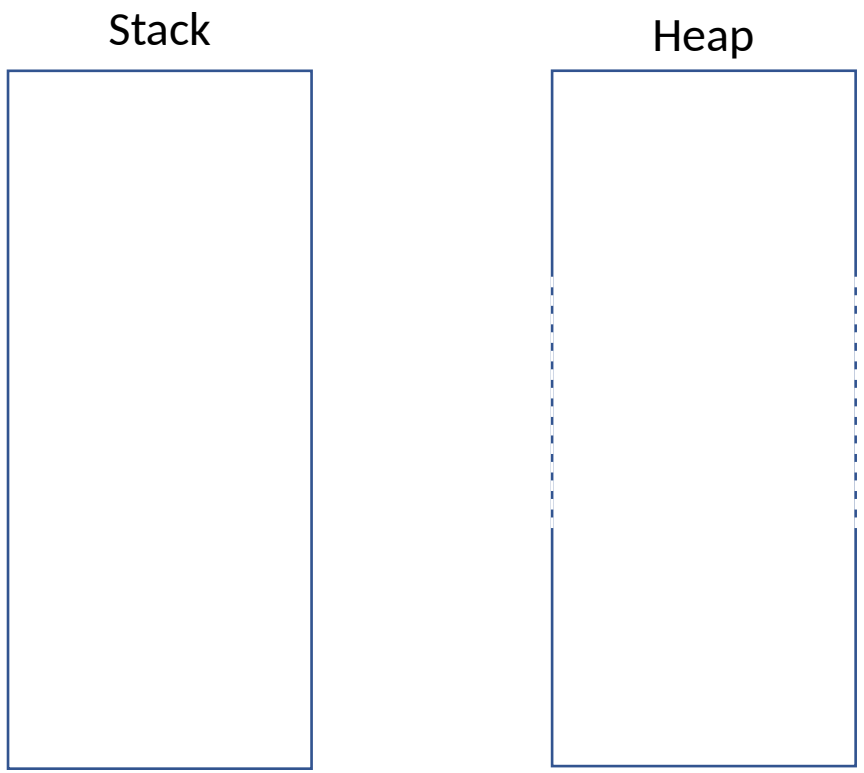
- The data (object) **is not** stored inside the **variable**
- The **variable** stores a number (address) that locates that *object*: **reference**




A variable is like a fixed size cup

The object's location (reference) can fit into a variable

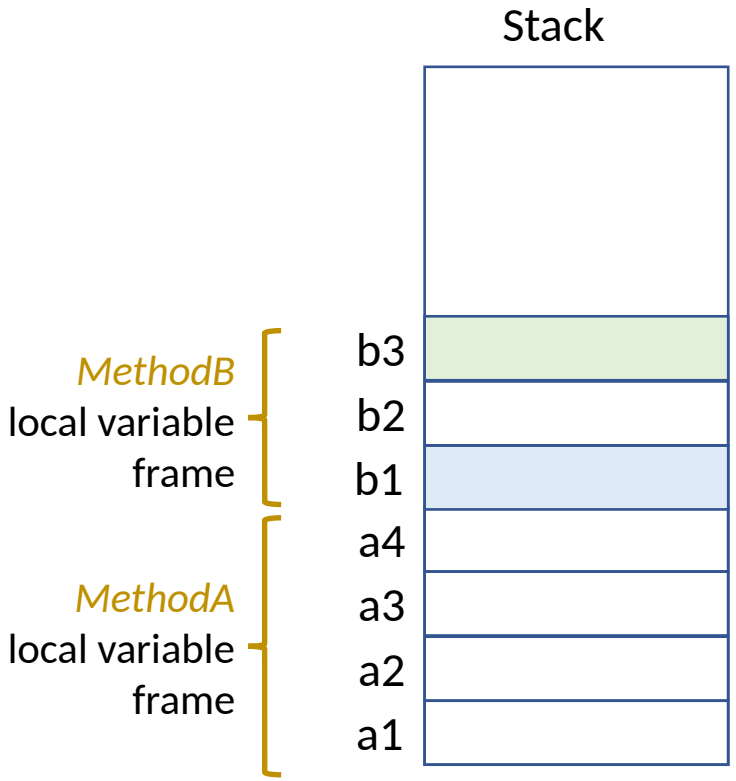
# Memory stack and heap

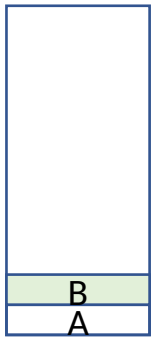



| | | |
|---|---|---|
| **Size** | Small (1-4 MB) | Large (hundreds of MB) |
| **Memory Allocation** | *Last In First Out (LIFO)* | *Pattern Free* |
| **Speed** | Fast | Slower than Stack |

# Method Invocation

```
void MethodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

void MethodA(double a1, int a2)
{
   int a3 = 10;
   double a4 = a1;
   MethodB(a4);
}
```



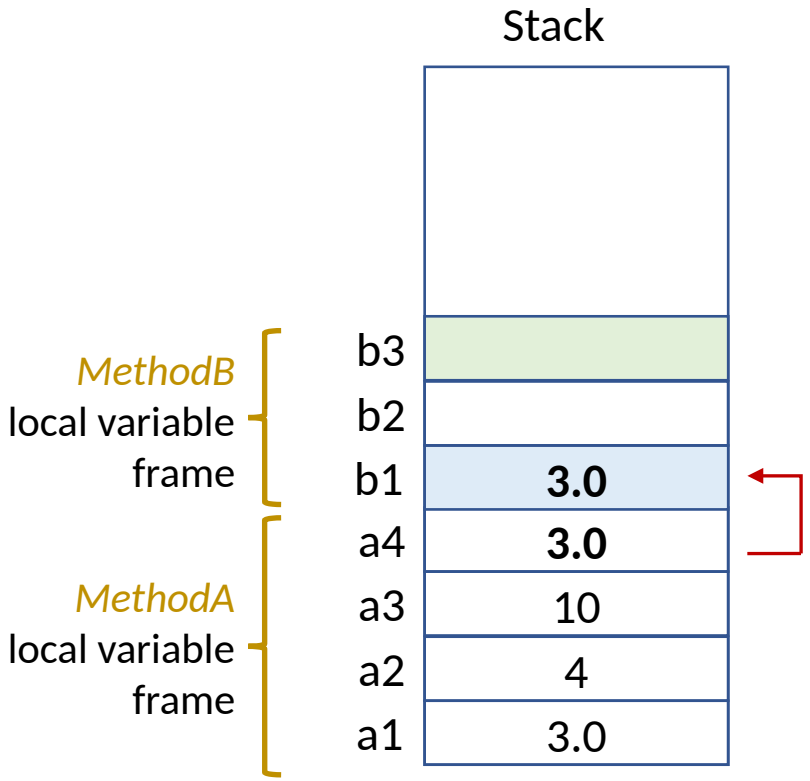



# Method Invocation

C#'s default way of **passing** parameters is **by value**—a copy of the arguments' content is stored in the parameters of the method being called—they are **different** areas of the memory

The content of *a4* is **copied** to *b1*

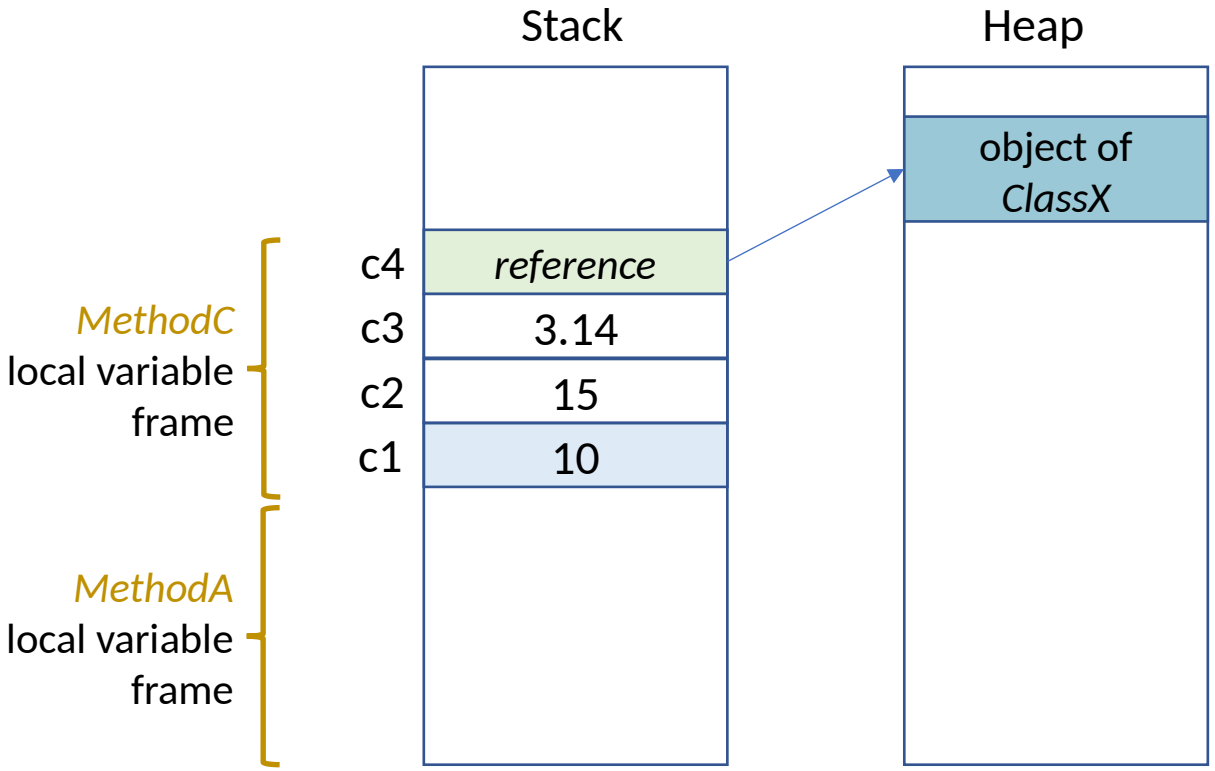
```
void MethodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

void MethodA(double a1, int a2)
{
  int a3 = 10;
  double a4 = a1;
  MethodB(a4);
}
```



# Objects allocation

The object of *ClassX* is created on the heap—*c4* (on the stack) contains a reference to it

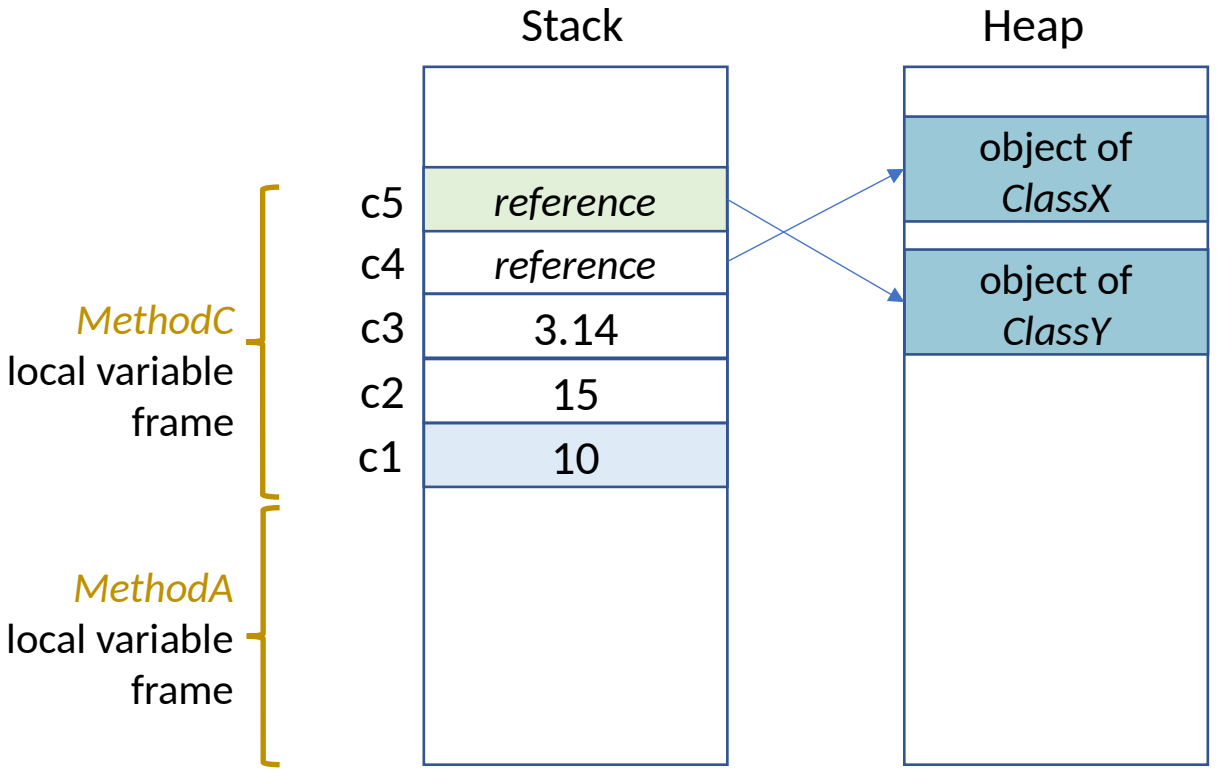
```
void MethodC(int c1)
{
  int c2 = 15;
  double c3 = 3.14;
  ClassX c4 = new ClassX();
}

void MethodA(double a1, int a2)
{
  int a3 = 10;
  double a4 = a1;
  MethodB(a4);
  MethodC(a3);
}
```



# Objects allocation

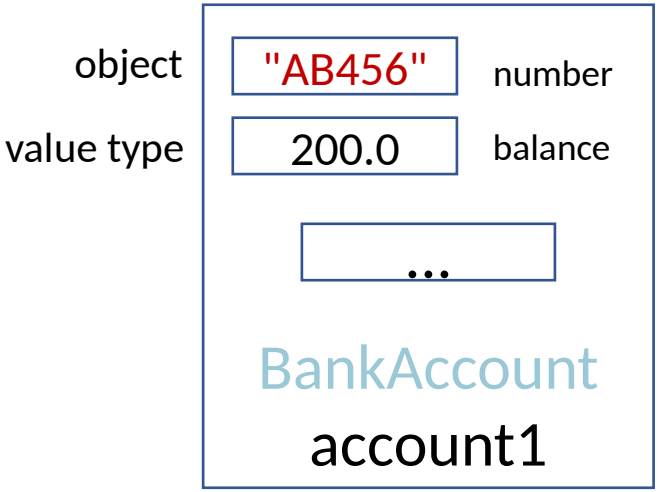
Adding another object of the *ClassY* class— it is created on the heap, and it is referenced by *c5*

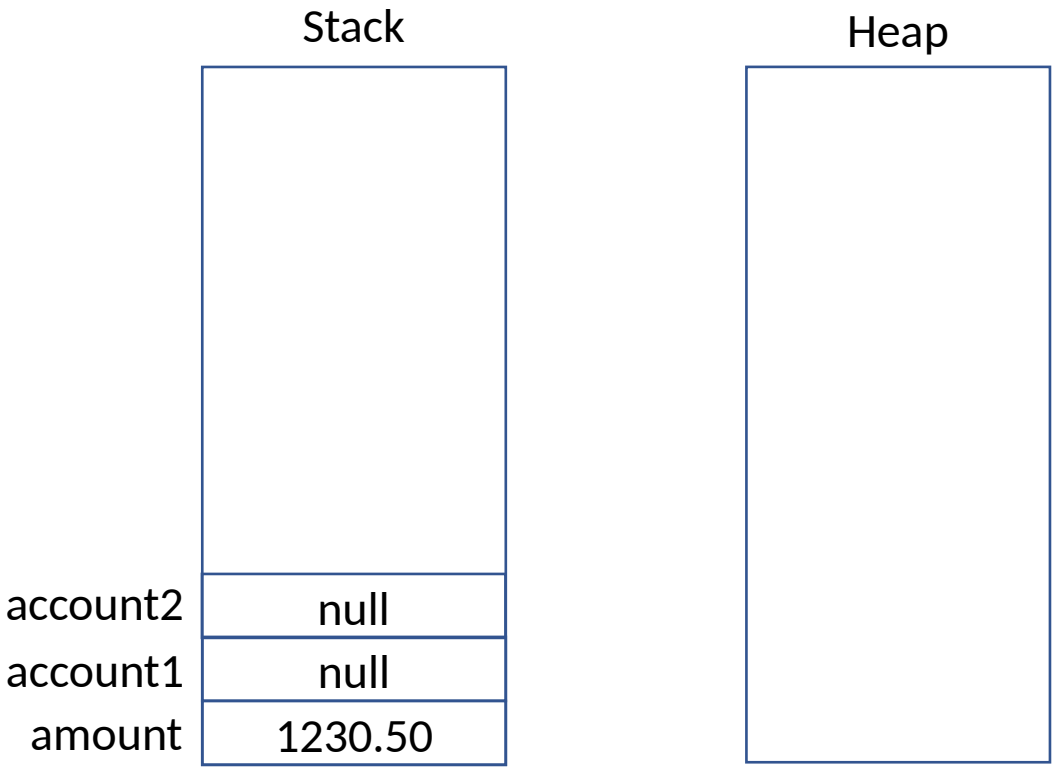
```
void MethodC(int c1)
{
  int c2 = 15;
  double c3 = 3.14;
  ClassX c4 = new ClassX();
  ClassY c5 = new ClassY();
}

void MethodA(double a1, int a2)
{
  int a3 = 10;
  double a4 = a1;
  MethodB(a4);
  MethodC(a3);
}
```



# Example: double and *BankAccount*

```
class Program
{
    public static void Main()
    {
        double amount = 1230.50;
        BankAccount  account1;
        BankAccount  account2;
    }
}
```

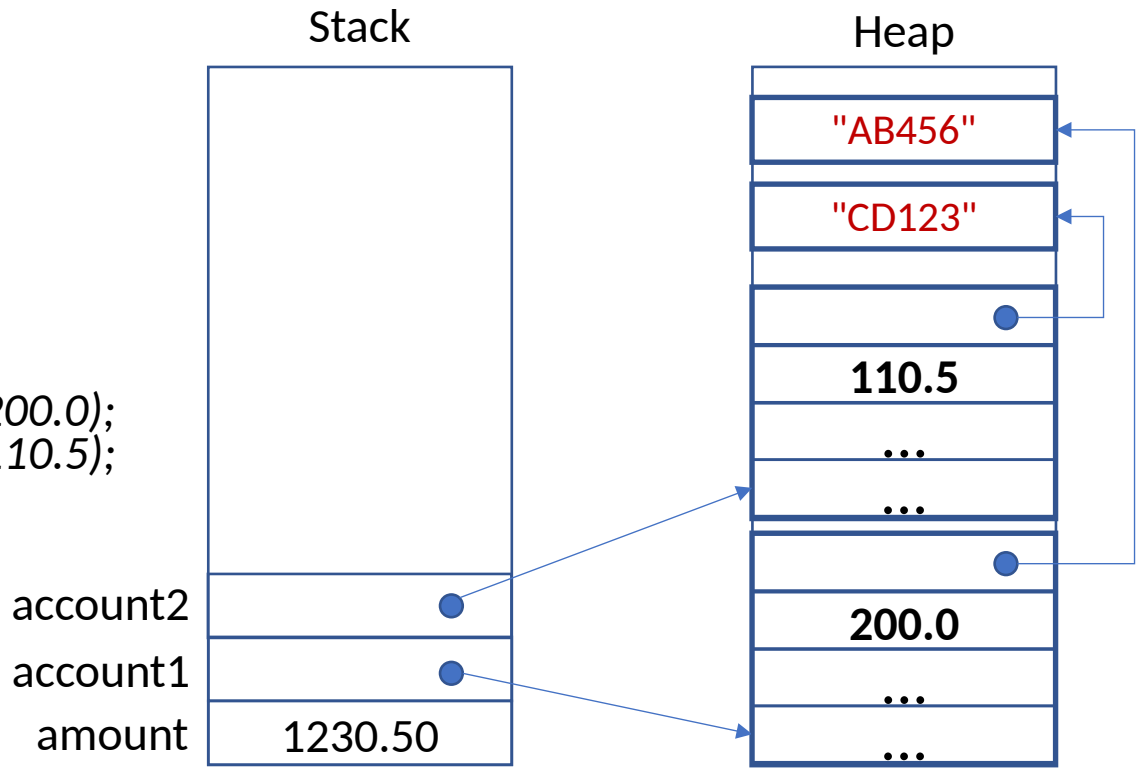



When the object is created, where will the *attributes* be stored?

# Example: double and *BankAccount*

```
class Program
{
   public static void Main()
   {
      double amount = 1230.50;
      BankAccount  account1 = new BankAccount("AB456", 200.0);
      BankAccount  account2 = new BankAccount("CD123", 110.5);
   }
}
```



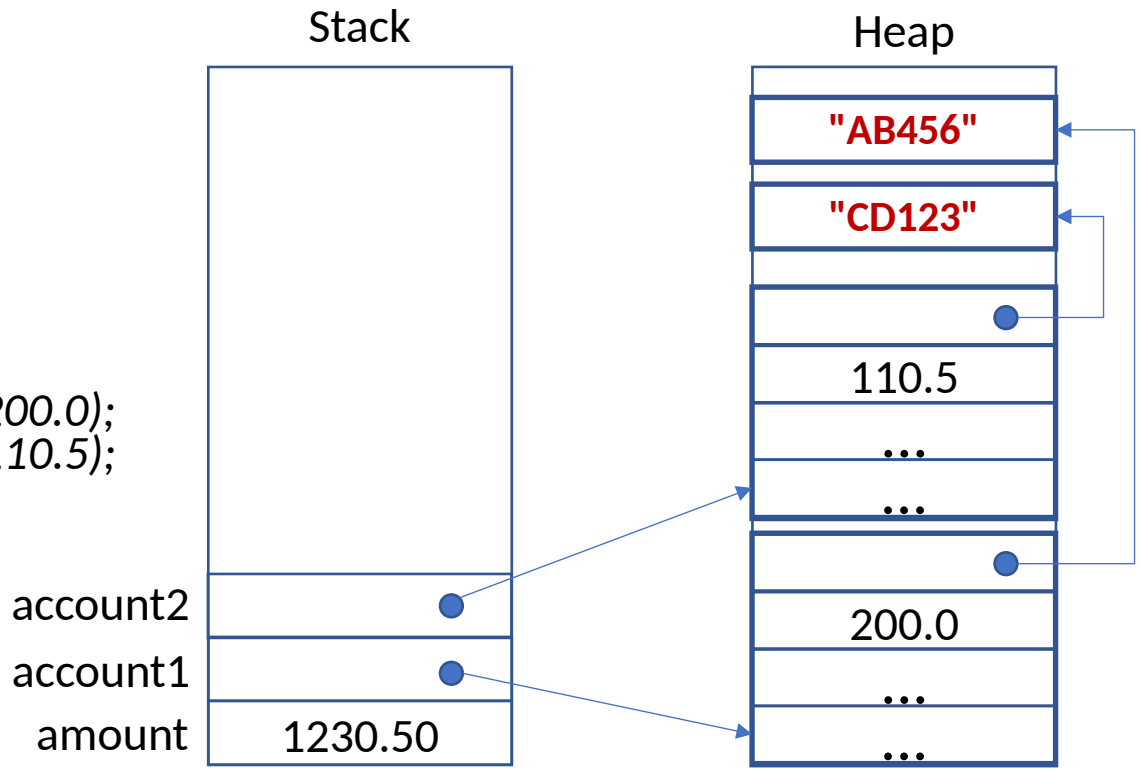

*account1* and *account2* are *reference* type variables

the attribute ***balance*** of *BankAccount* is a primitive (value) type (int)

its value is allocated **within** the object on the heap

# Example: double and *BankAccount*

```
class Program
{
   public static void Main()
   {
      double amount = 1230.50;
      BankAccount  account1 = new BankAccount("AB456", 200.0);
      BankAccount  account2 = new BankAccount("CD123", 110.5);
   }
}
```



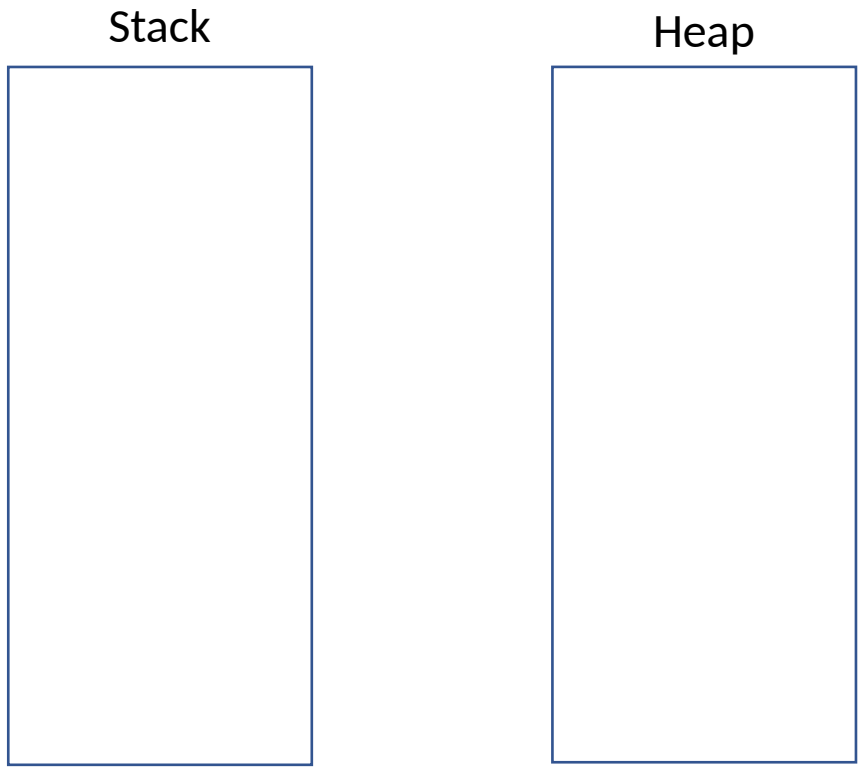

*account1* and *account2* are *reference* type variables

the attribute ***number*** of *BankAccount* is a reference type (string)

its content may be stored **somewhere else** on the heap and referenced by the *BankAccount* object

# Example: *int[]*

```
class Program
{
    public static void Main()
    {
        int[] scores = new int[4];


    }
}
```



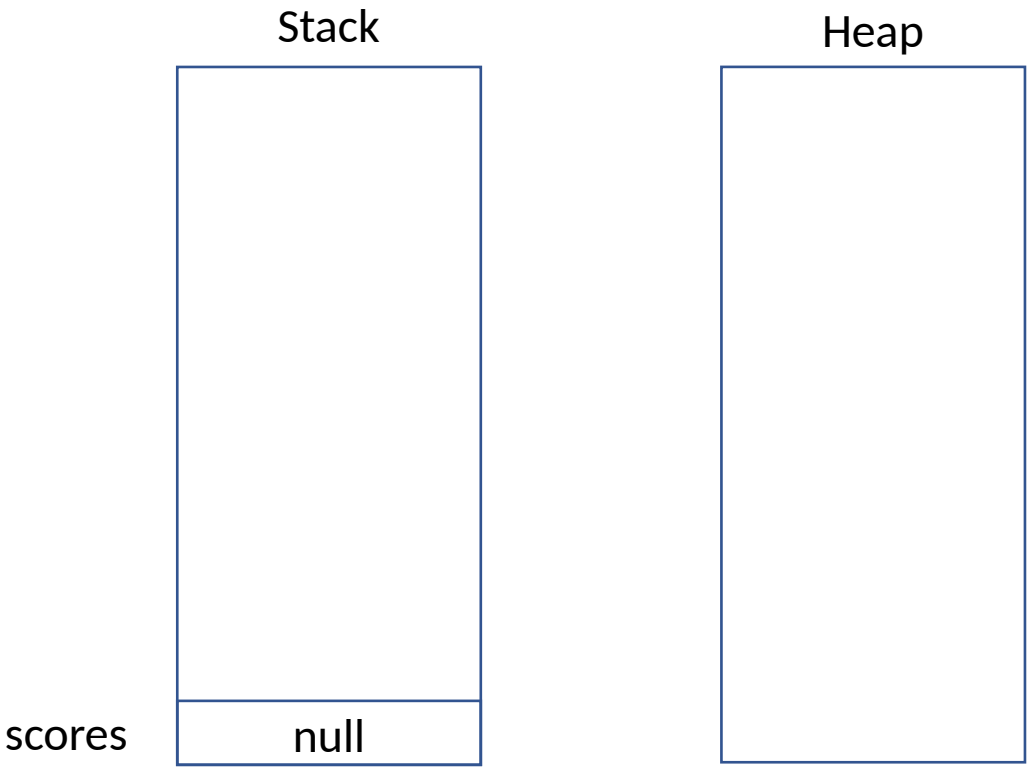

*scores* is a local array of *primitive* (value) types:
*stack* or *heap*?

# Example: *int[]*

```
class Program
{
    public static void Main()
    {
        int[] scores;

    }
}
```

*scores* is a local reference type variable allocated on the stack

the default value is *null* (not initialised)



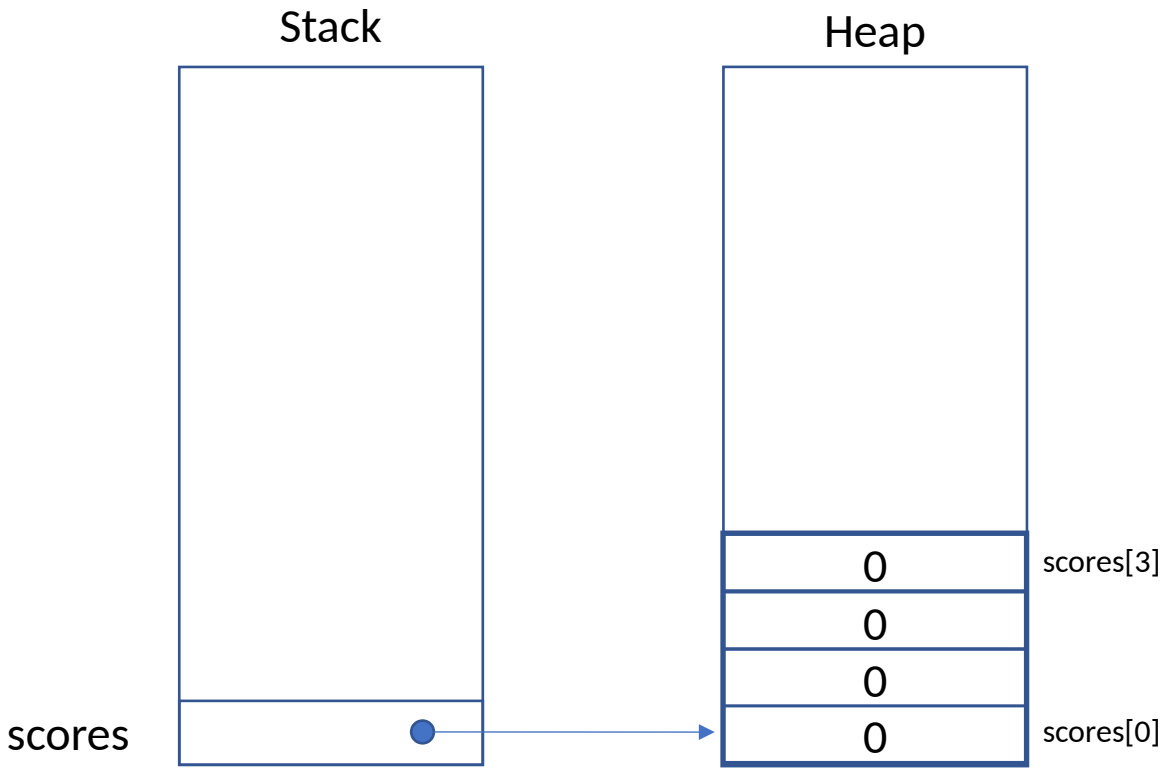
# Example: *int[]*

```
class Program
{
    public static void Main()
    {
        int[] scores = new int[4];

    }
}
```

when new is used, space for the 4 elements is allocated on the heap

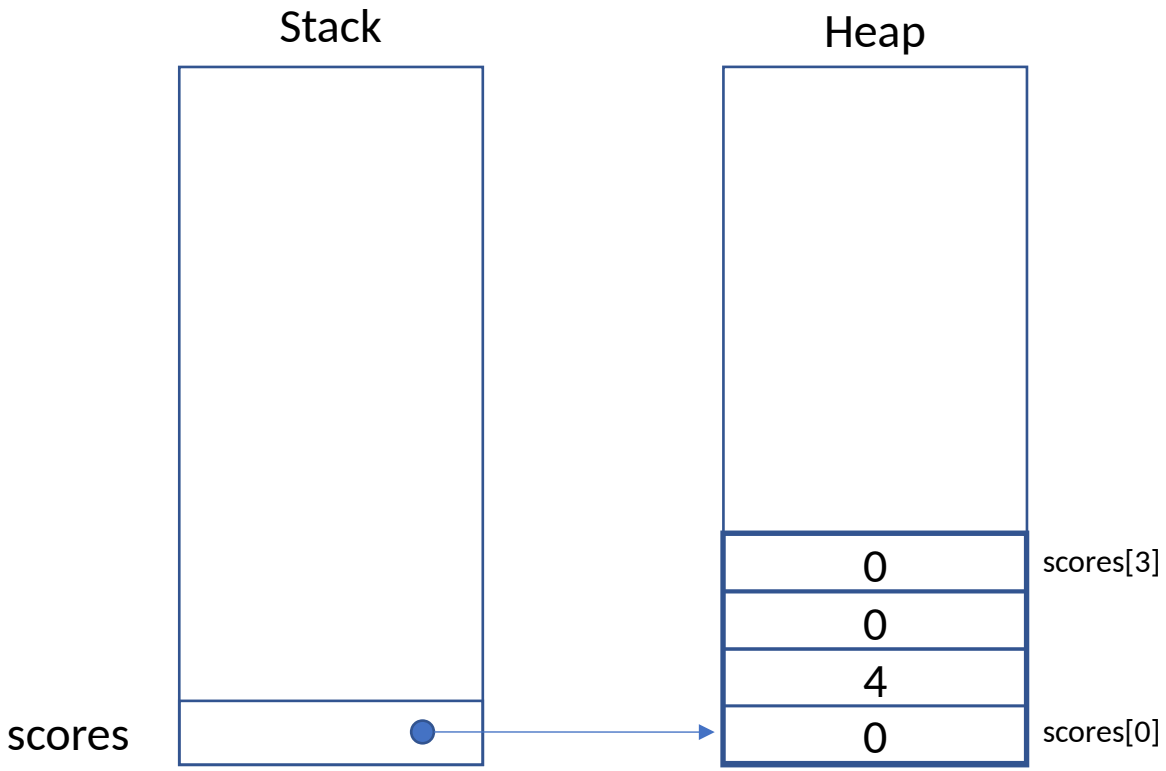
the reference is assigned to *scores*

the elements of the array are initialised to 0



# Example: *int[]*

```
class Program
{
    public static void Main()
    {
        int[] scores = new int[4];
        scores[1] = 4;

    }
}
```

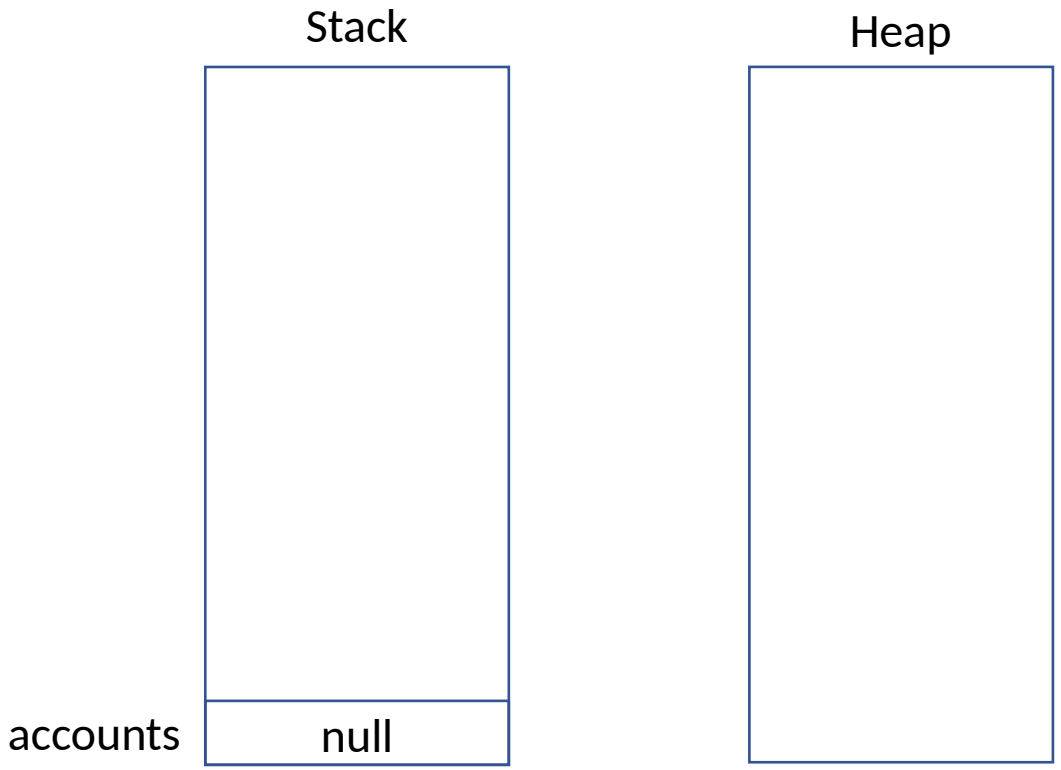


# Example: *BankAccount[]*

```
class Program
{
    public static void Main()
    {
        BankAccount[]  accounts;

    }
}
```

*accounts* is a reference type variable allocated on the stack

default value is *null* (not initialised)



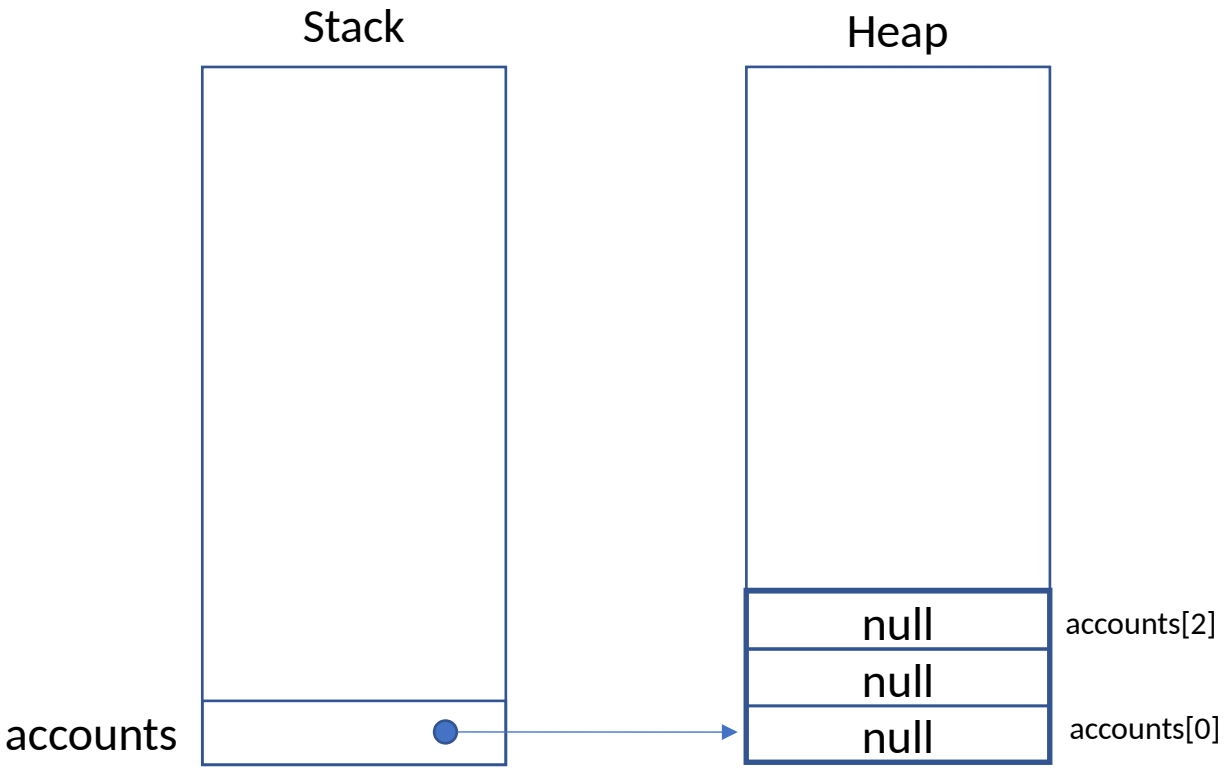
# Example: *BankAccount[]*

```
class Program
{
    public static void Main()
    {
        BankAccount[]  accounts = new BankAccount[3];


    }
}
```

when new is used, space for the 3 elements is allocated on the heap

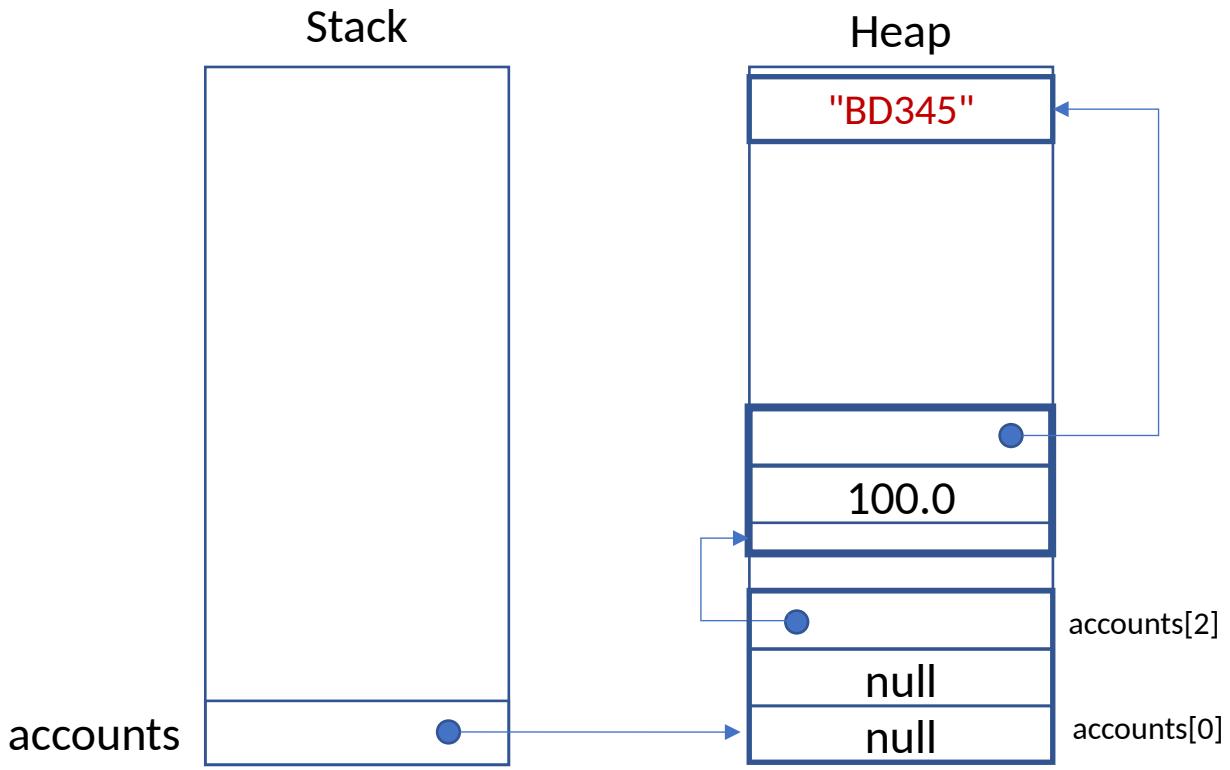
the reference is assigned to *accounts*

the elements of the array are initialised to *null*



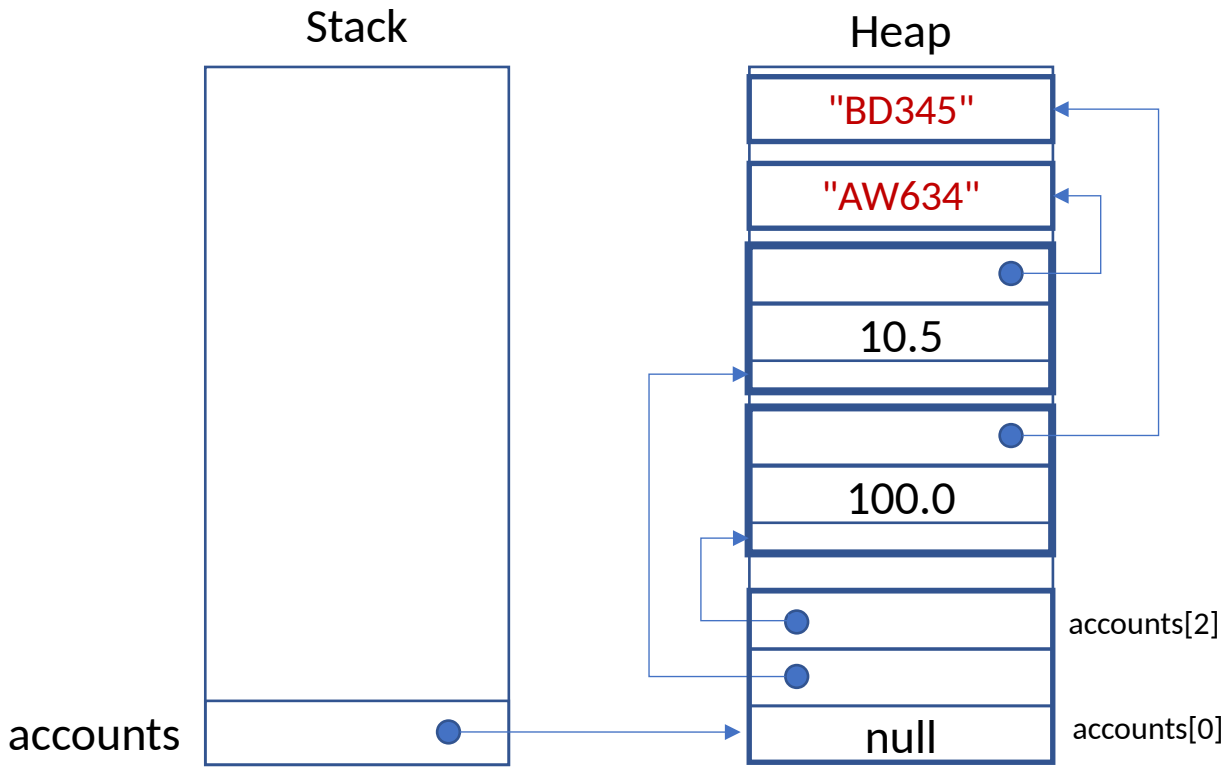
# Example: *BankAccount[]*

```
class Program
{
    public static void Main()
    {
        BankAccount[]  accounts = new BankAccount[3];
        accounts[2] = new BankAccount("BD345", 100.0);

    }
}
```



# Example: *BankAccount[]*

```
class Program
{
    public static void Main()
    {
        BankAccount[]  accounts = new BankAccount[3];
        accounts[2] = new BankAccount("BD345", 100.0);
        accounts[1] = new BankAccount("AW634", 10.5);
    }
}
```



# Outline

- More on *Value Types* and *Reference Types*
  - '=' and '==' operators
  - Method invocation and parameter passing
- The *this* keyword
- More on *Encapsulation*

# The == operator

```
class Program
{
    public static void Main(string[] args)
    {
        int a = 10;
        int b = 10;
        if (a == b)
            Console.WriteLine("a is equal to b");




    }
}
```

# The == operator

```
class Program
{
    public static void Main(string[] args)
    {
        int a = 10;
        int b = 10;
        if (a == b)
            Console.WriteLine("a is equal to b");

        BankAccount account1 = new BankAccount("AB456", 200.0);
        BankAccount account2 = new BankAccount("AB456", 200.0);
        if (account1 == account2)
            Console.WriteLine("account1 is equal to account2");
    }
}
```

# Question

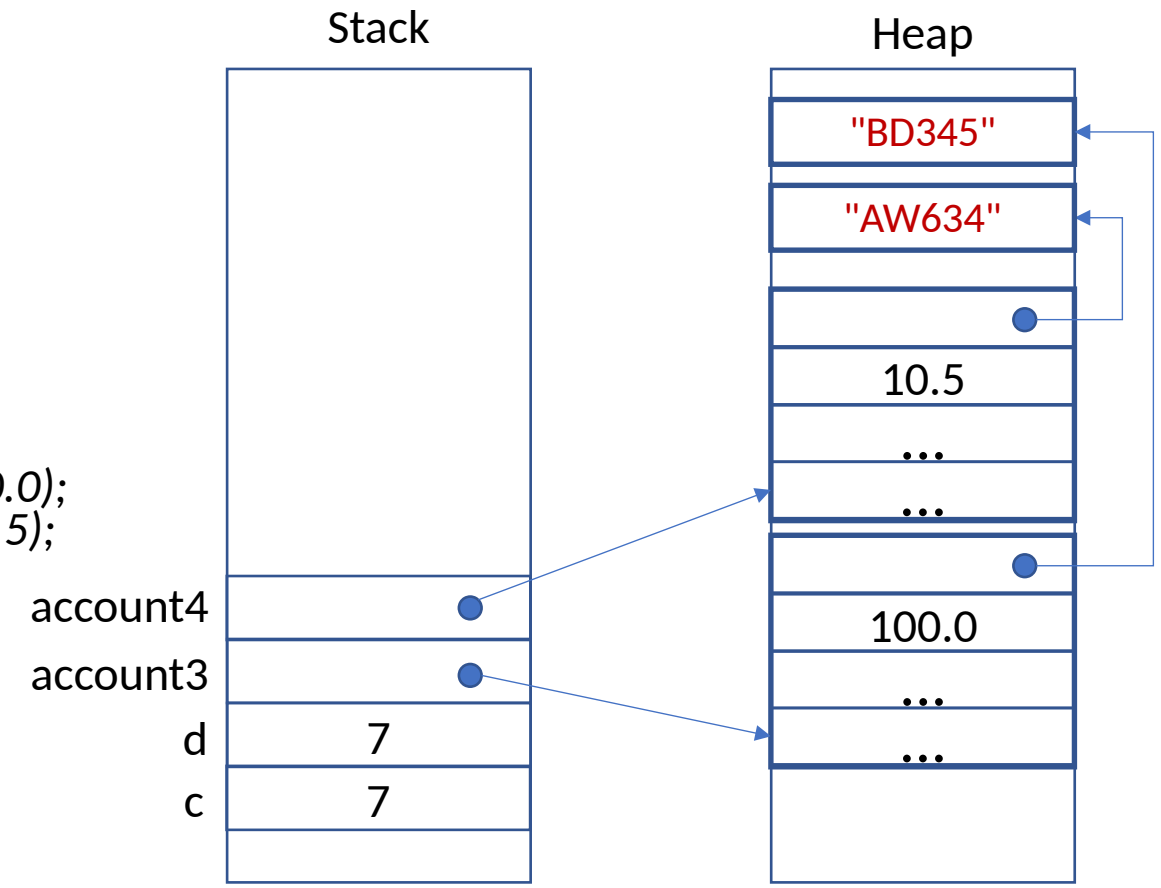
- What will the previous program print?

# The = operator

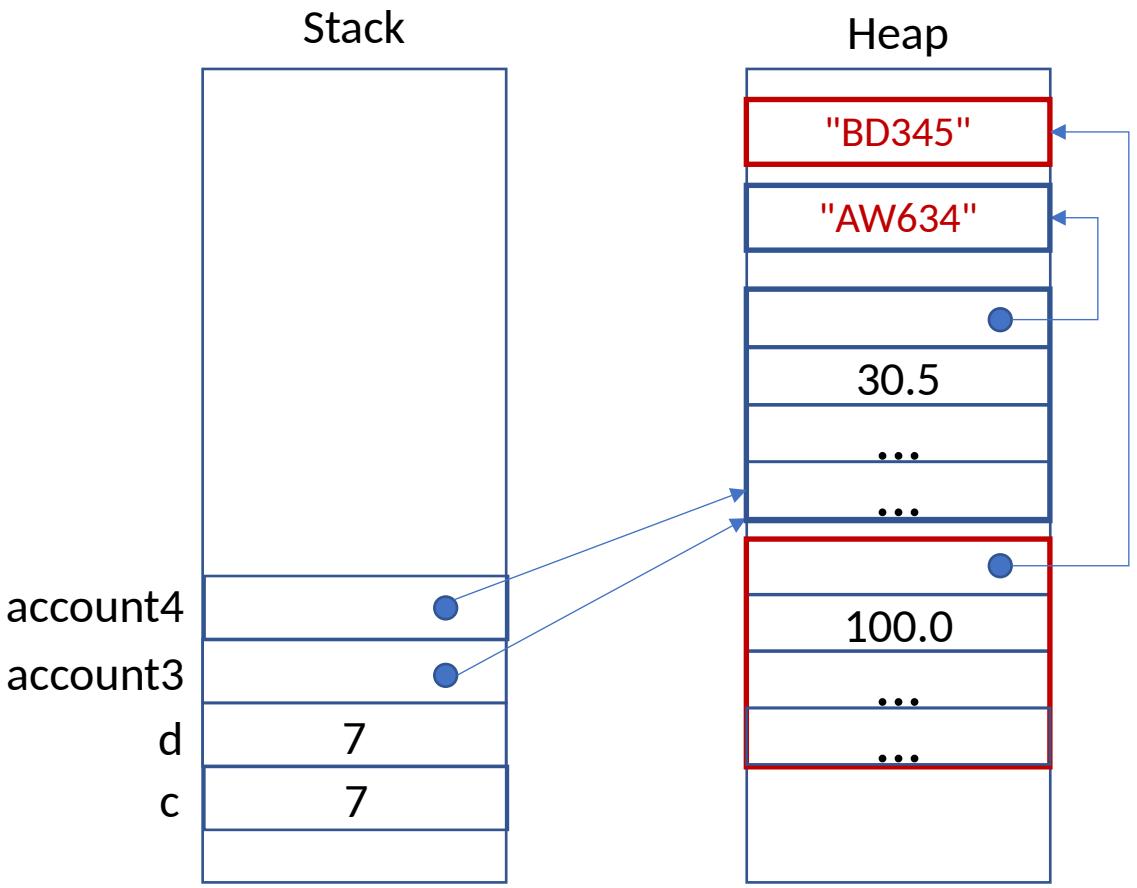
```
class Program
{
    public static void Main(string[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        Console.WriteLine(c);
        Console.WriteLine(d);

        BankAccount account3 = new BankAccount("BD345", 100.0);
        BankAccount account4 = new BankAccount("AW634", 10.5);
        account3 = account4;
        account3.Deposit(10.0);
        account4.Deposit(10.0);

        Console.WriteLine(account3.GetBalance());
        Console.WriteLine(account4.GetBalance());
    }
}
```

# Answer

```
class Program
{
    public static void Main(string[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        Console.WriteLine(c);
        Console.WriteLine(d);

        BankAccount account3 = new BankAccount("BD345", 100.0);
        BankAccount account4 = new BankAccount("AW634", 10.5);
        account3 = account4;
        account3.Deposit(10.0);
        account4.Deposit(10.0);

        Console.WriteLine(account3.GetBalance());
        Console.WriteLine(account4.GetBalance());
    }
}
```



# Garbage Collection




The red area of the heap memory is no longer referenced by any variables

The Garbage Collector (GC) of the Common Language Runtime (CLR) will mark it as "free"

# Garbage Collection

- Is a feature provided by the *Language Runtime*
- The programmer does not need to **deallocate** objects explicitly
- Can prevent issues due to **memory leaks**

# Outline

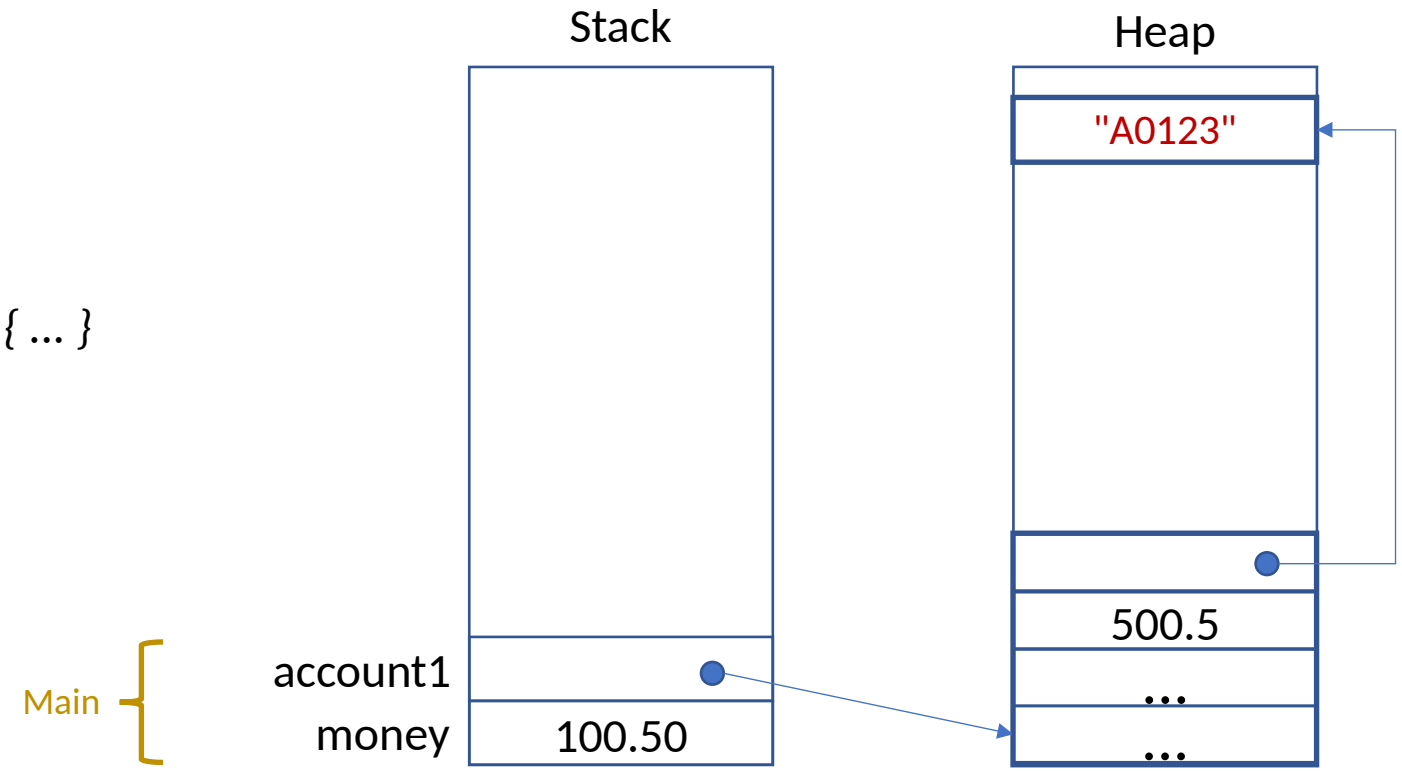
- More on *Value Types* and *Reference Types*
  - '=' and '==' operators
  - Method invocation and parameter passing
- The *this* keyword
- More on *Encapsulation*

# Reminder: Method invocation

- When a new method is invoked, a memory area—the *local variable frame*—is **reserved** for it at the **top** of the **stack**
- The method **arguments** and the **variables** declared inside the method will be allocated on the frame
- **Pass-by-value**: a **copy** of the arguments is passed to the method
- The above stack area is **deallocated** when the method **terminates**

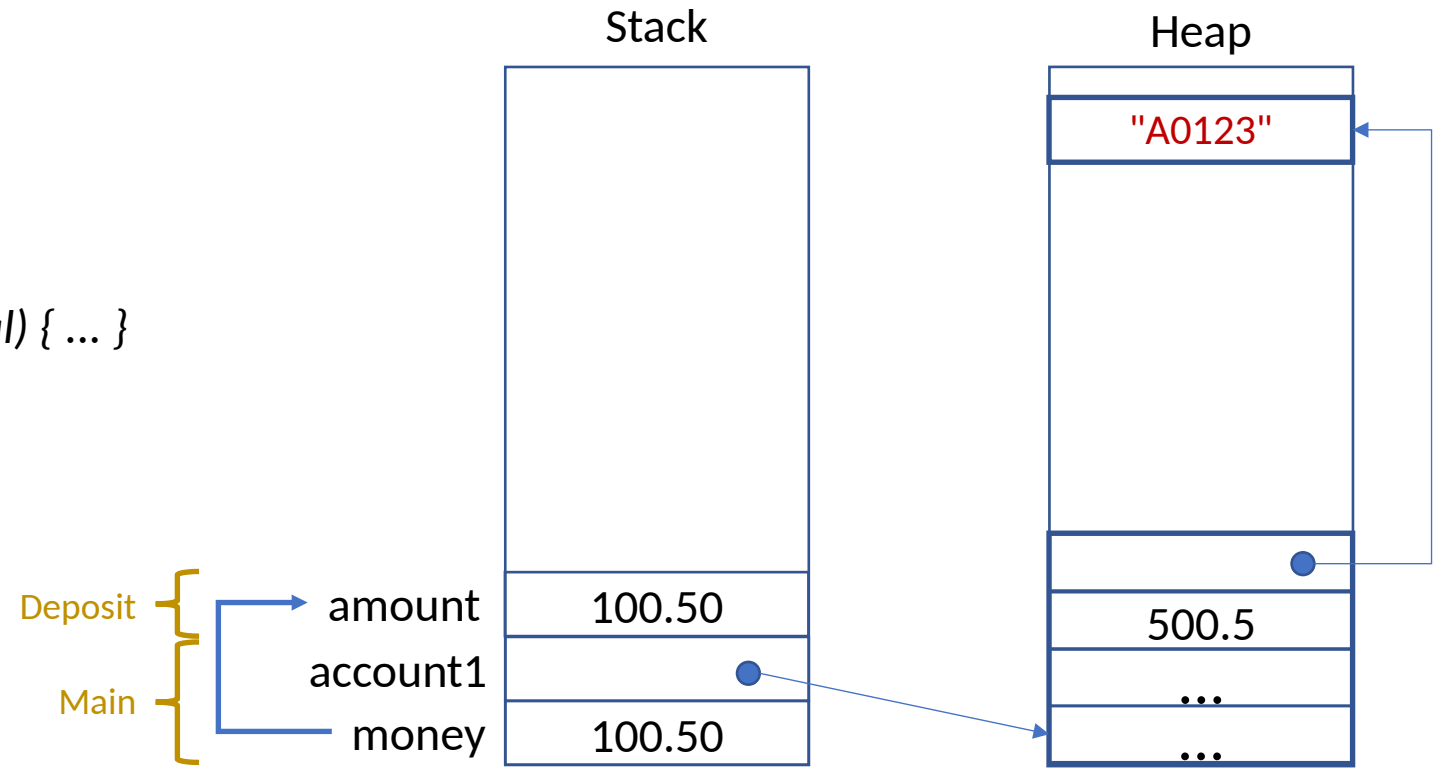
# Methods parameters: value types

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05; // e.g., an interest rate
        balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);


}
```



# Methods parameters: value types

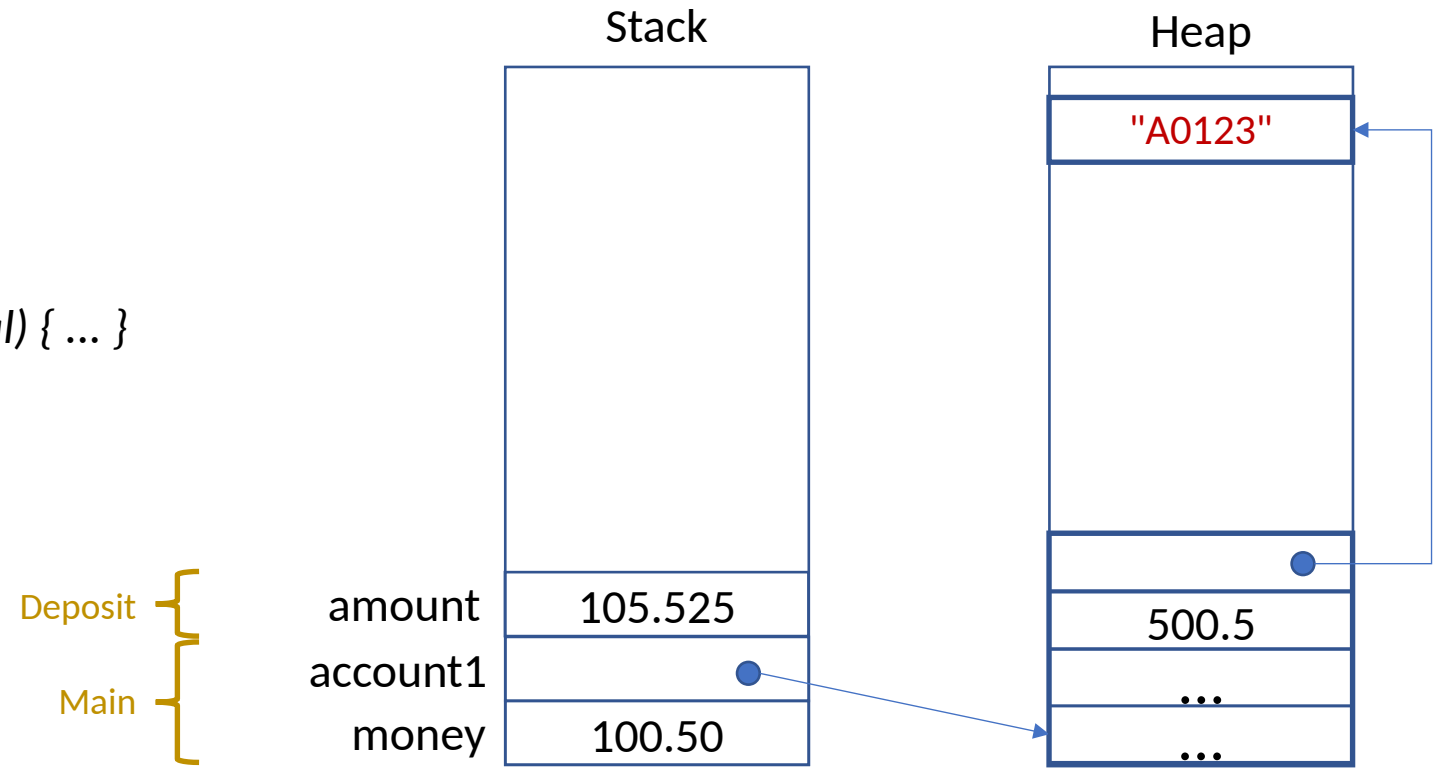
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05; // e.g., an interest rate
        balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(money);
        ...
}
```




when the Deposit method is invoked the value of *money* is copied into *amount*

# Methods parameters: value types

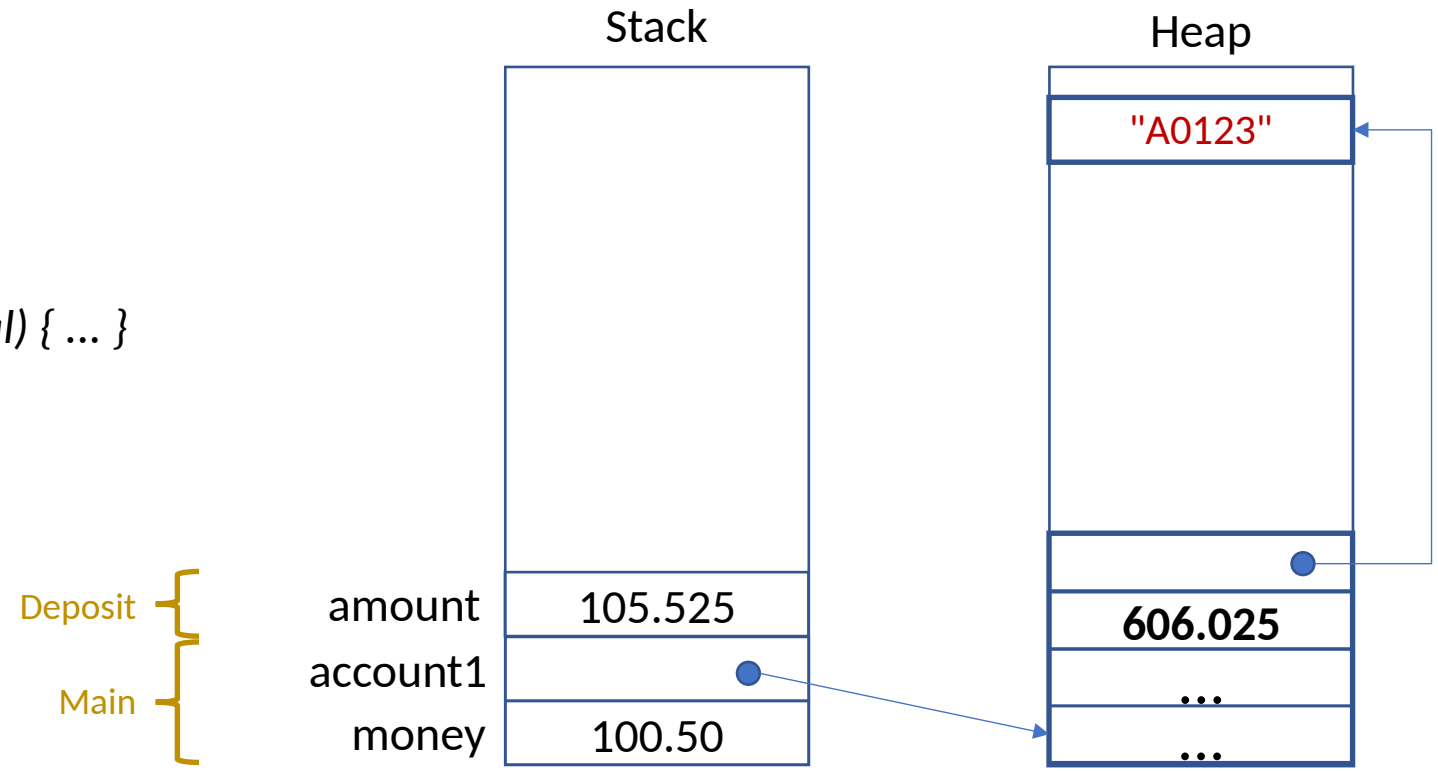
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05; // e.g., an interest rate
        balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(money);
        ...
}
```




the Deposit method changes *amount*

# Methods parameters: value types

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05; // e.g., an interest rate
        balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(money);
        ...
}
```



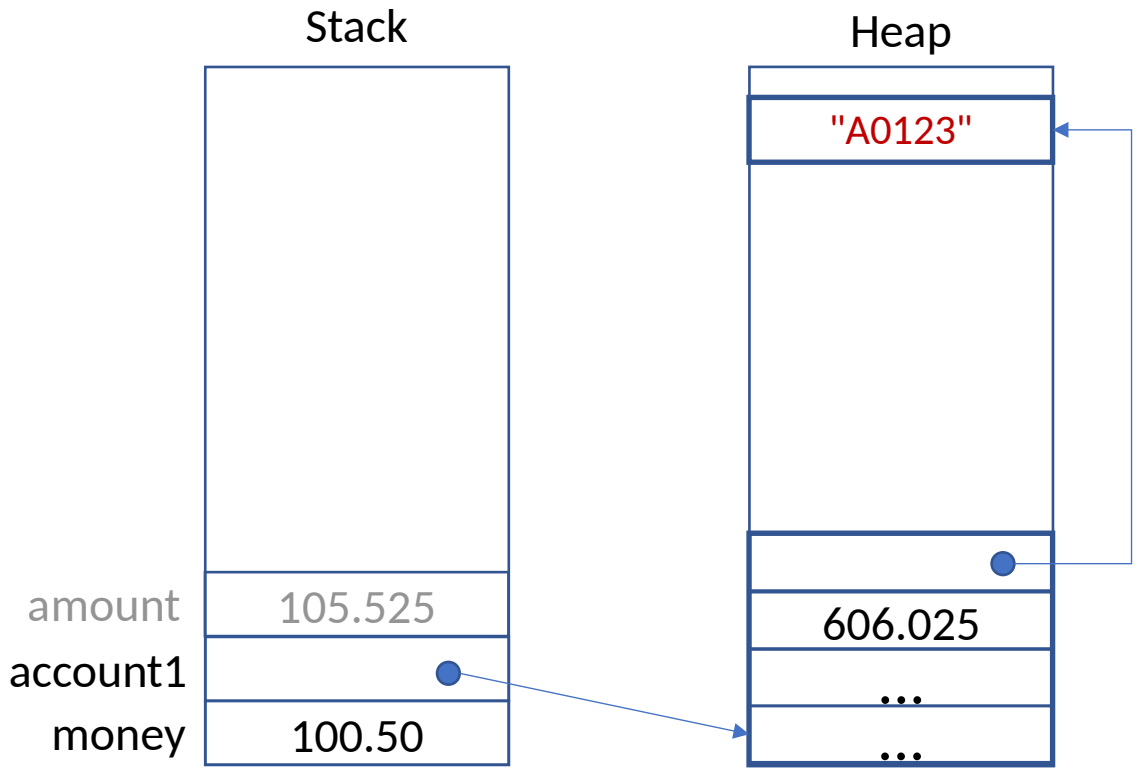

the Deposit method changes *balance* because it can access the attributes of the object *account1*

# Accessing attributes from a Method

- We know that a Method of a class can access the attributes defined in that class
- How is this implemented with *objects* created on the heap?

# Methods parameters: value types

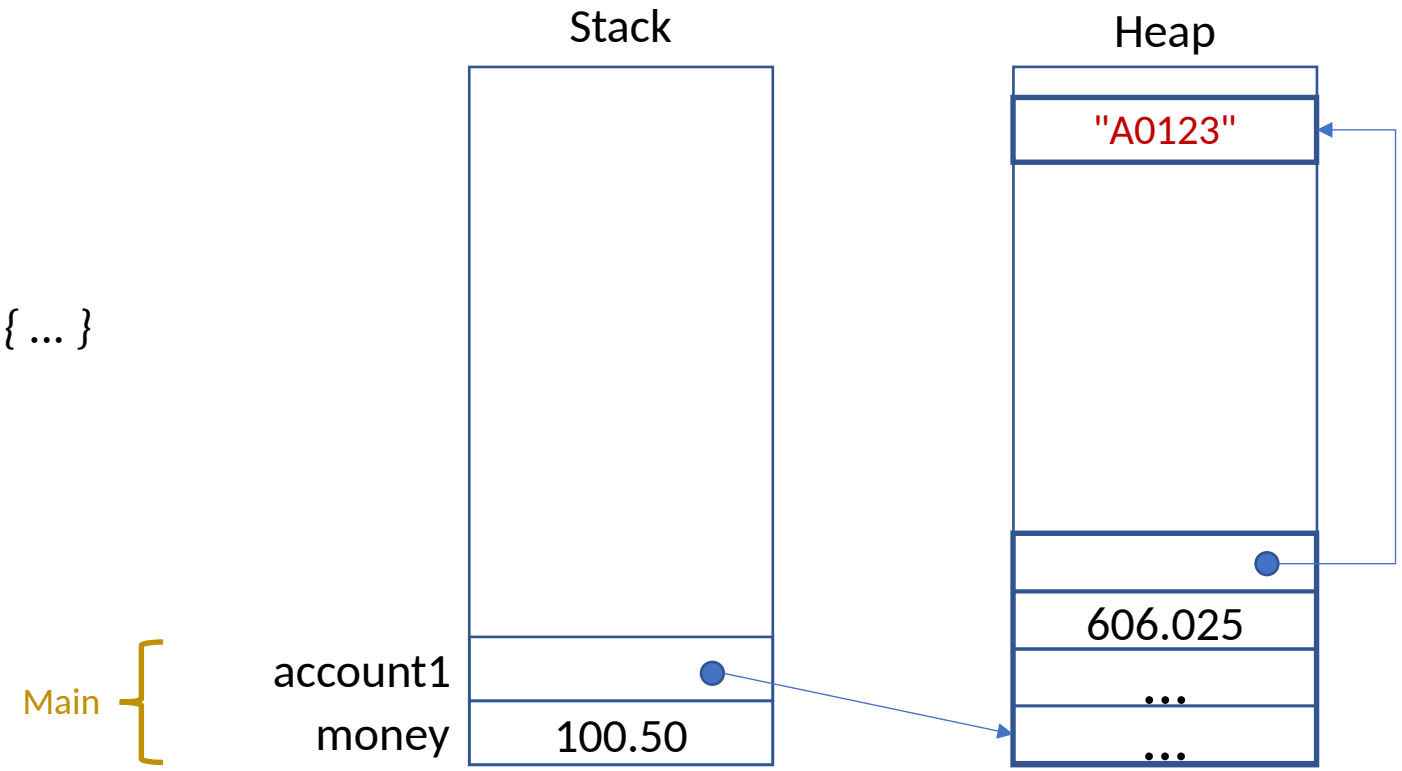
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05; // e.g., an interest rate
        balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(money);
        ...
}
```




when the Deposit method terminates the *amount* variable is removed from the stack

# Methods parameters: value types

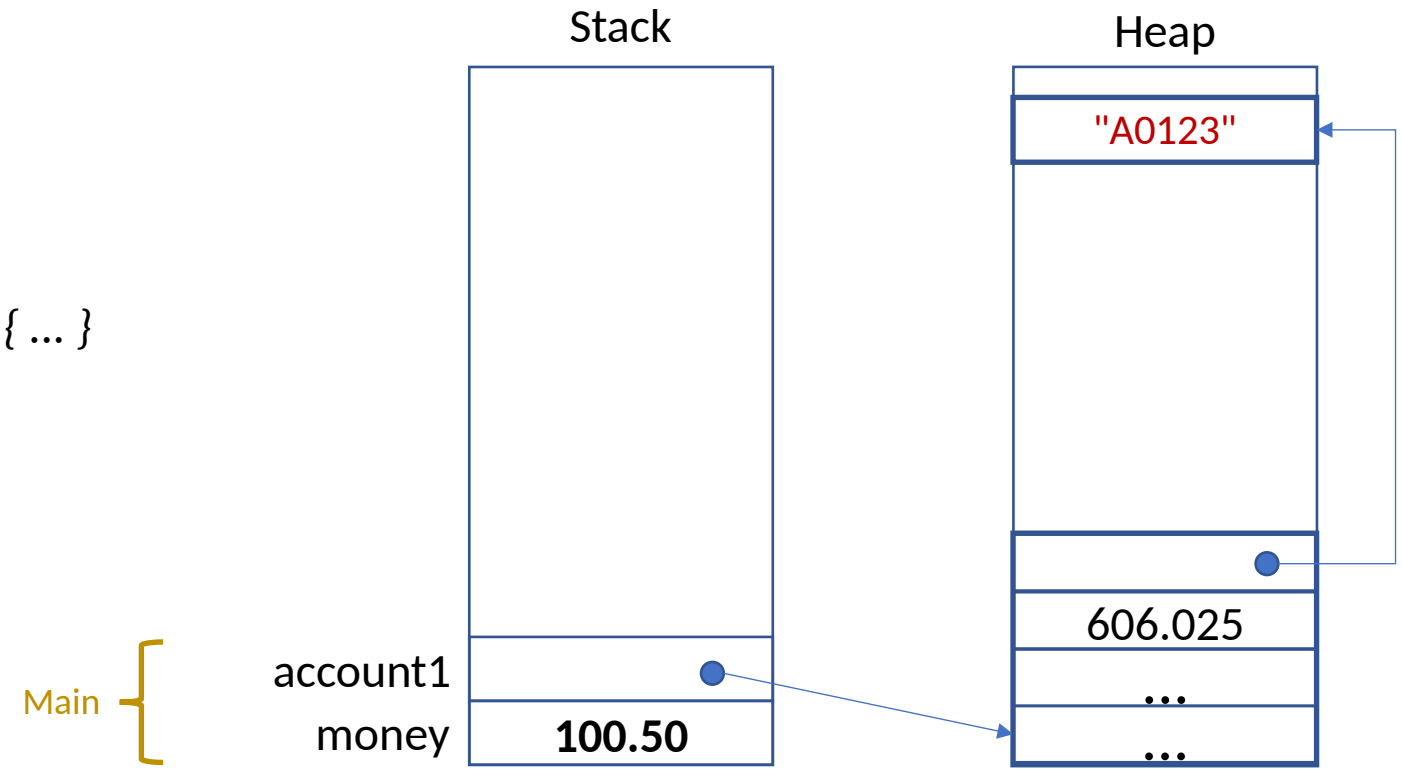
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05; // e.g., an interest rate
        balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(money);
        ...
}
```




Any changes to *amount* vanish when the *Deposit* method terminates

# Methods parameters: value types

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05; // e.g., an interest rate
        balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(money);
        ...
}
```




the content of the variable *money* is **not affected** by those changes

# Methods parameters: value types summary

- When a method is invoked, **new variables** are created on the stack according to the number of *parameters*
- A local **copy** of the values provided as argument is stored in them
- Any modifications would **only** affect the **local copy** of the arguments but **not** their **original values**
- Those variables (parameters) **only exist within** that method
- They are **removed** from the stack when the **method terminates**

**What happens with reference type parameters?**

# Methods parameters: reference types

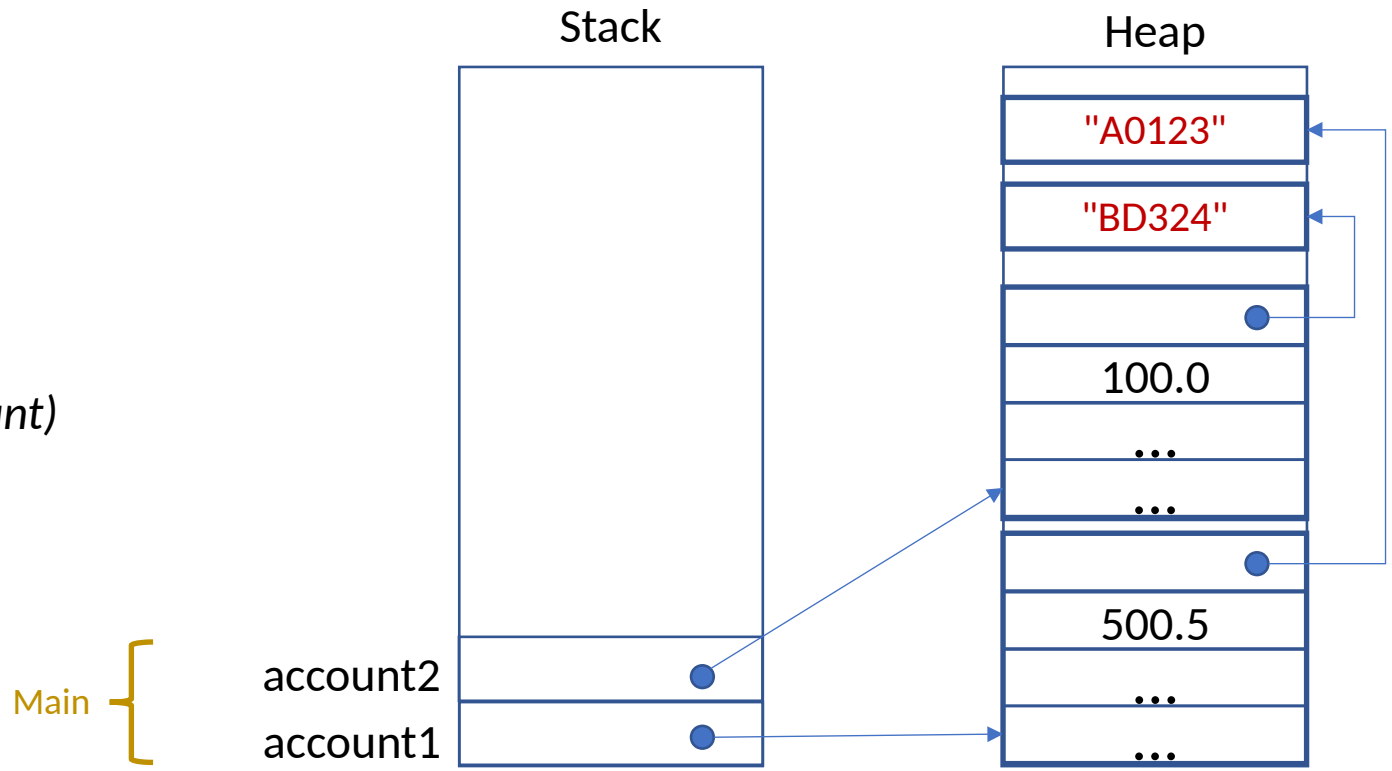
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void MoveAccount(SavingAccount dstAccount)
    {
        dstAccount.Save(balance);
        Close();
    }
}
```

```
class SavingAccount
{
    private string number;
    private double balance;
    private double interest;

    public SavingAccount(string num, double bal, double i) { ... }

    public void Save(double amount)
    {
        // deposit amount and calculate interest rate
    }
}
```

```
class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 3.8);

        ...
}
```

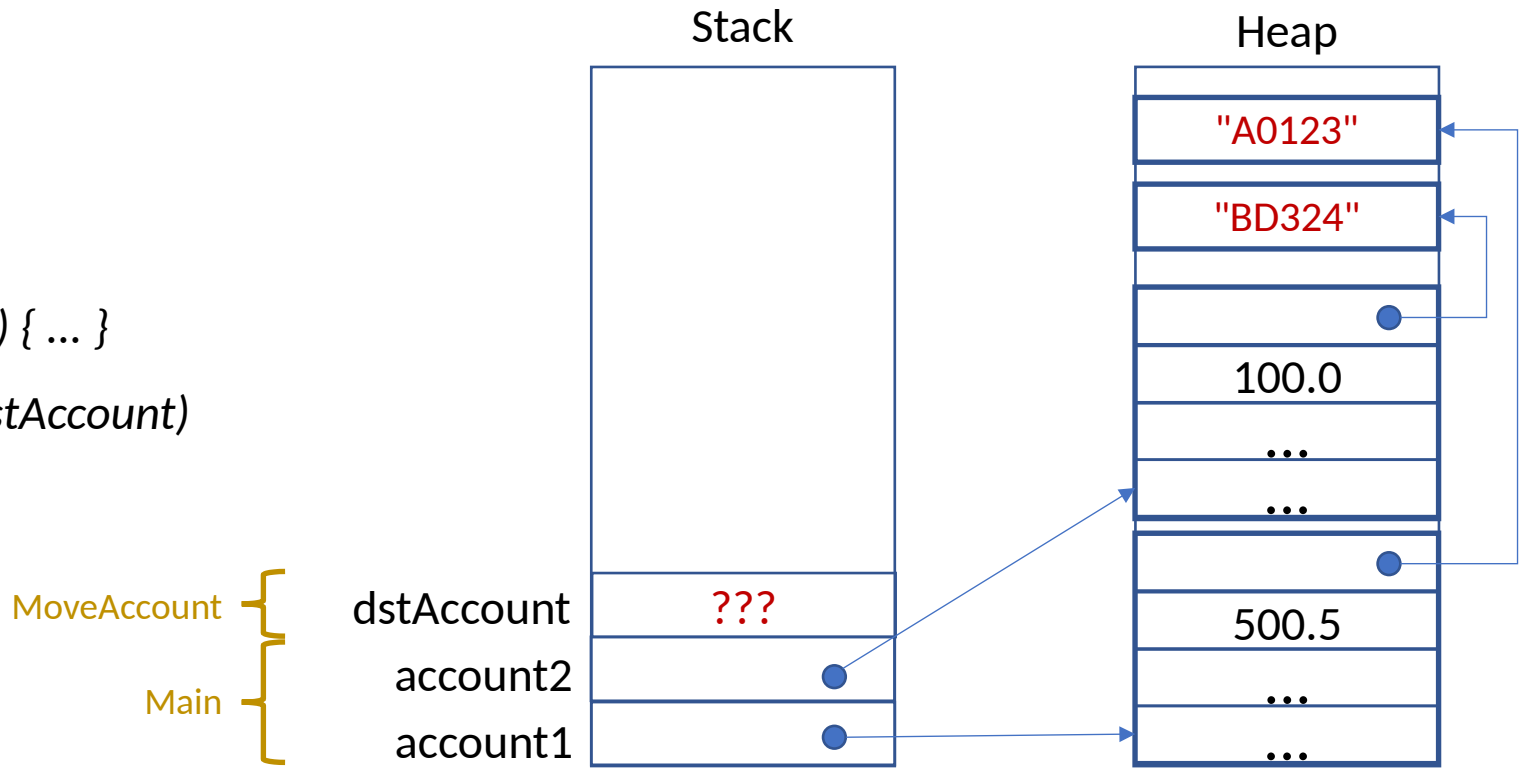
# Methods parameters: reference types

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void MoveAccount(SavingAccount dstAccount)
    {
        dstAccount.Save(balance);
        Close();
    }
}

class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 3.8);

        ...
}
```



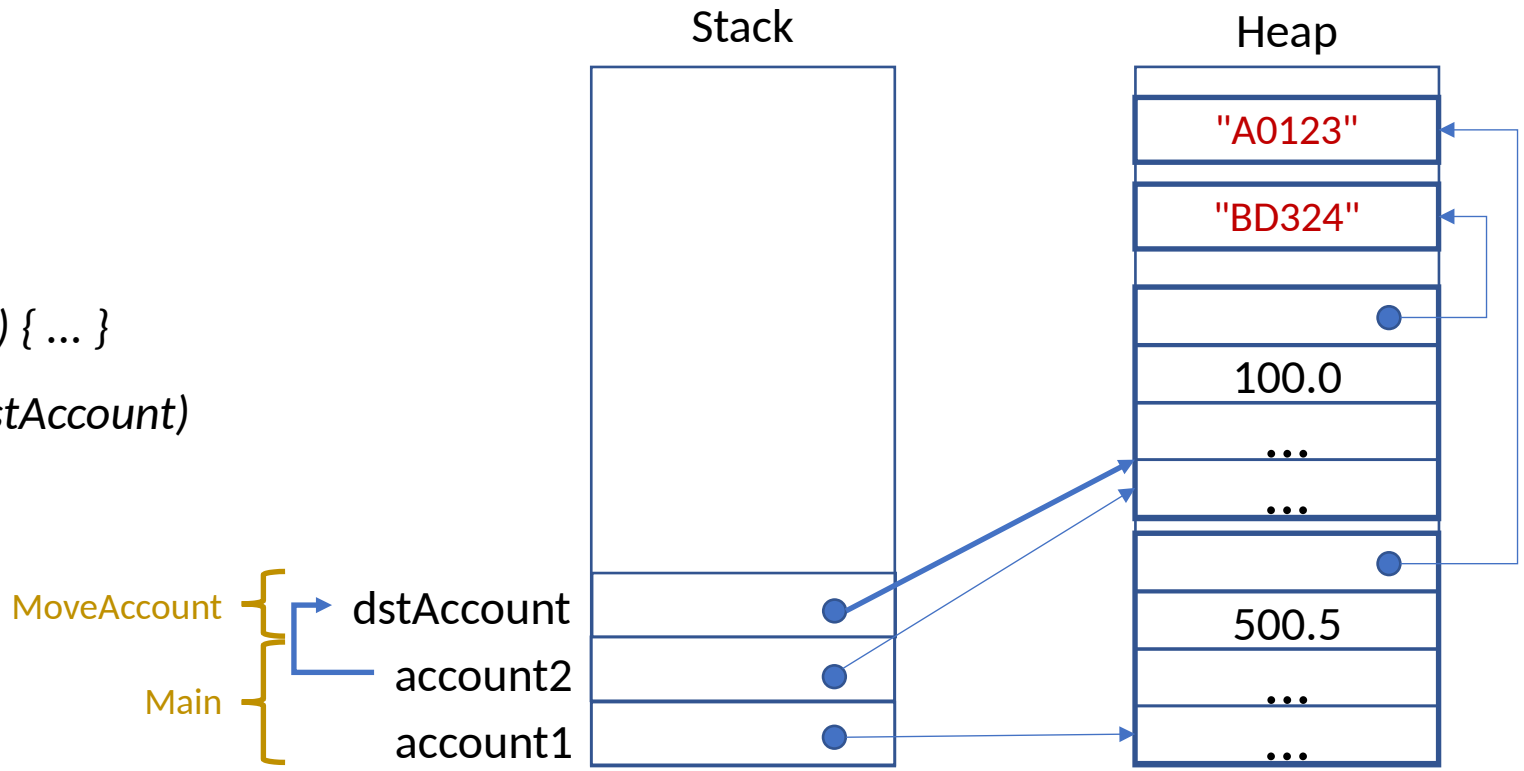
# Methods parameters: reference types

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void MoveAccount(SavingAccount dstAccount)
    {
        dstAccount.Save(balance);
        Close();
    }
}

class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 3.8);
        account1.MoveAccount(account2);
        ...
}
```



# Methods parameters: reference types

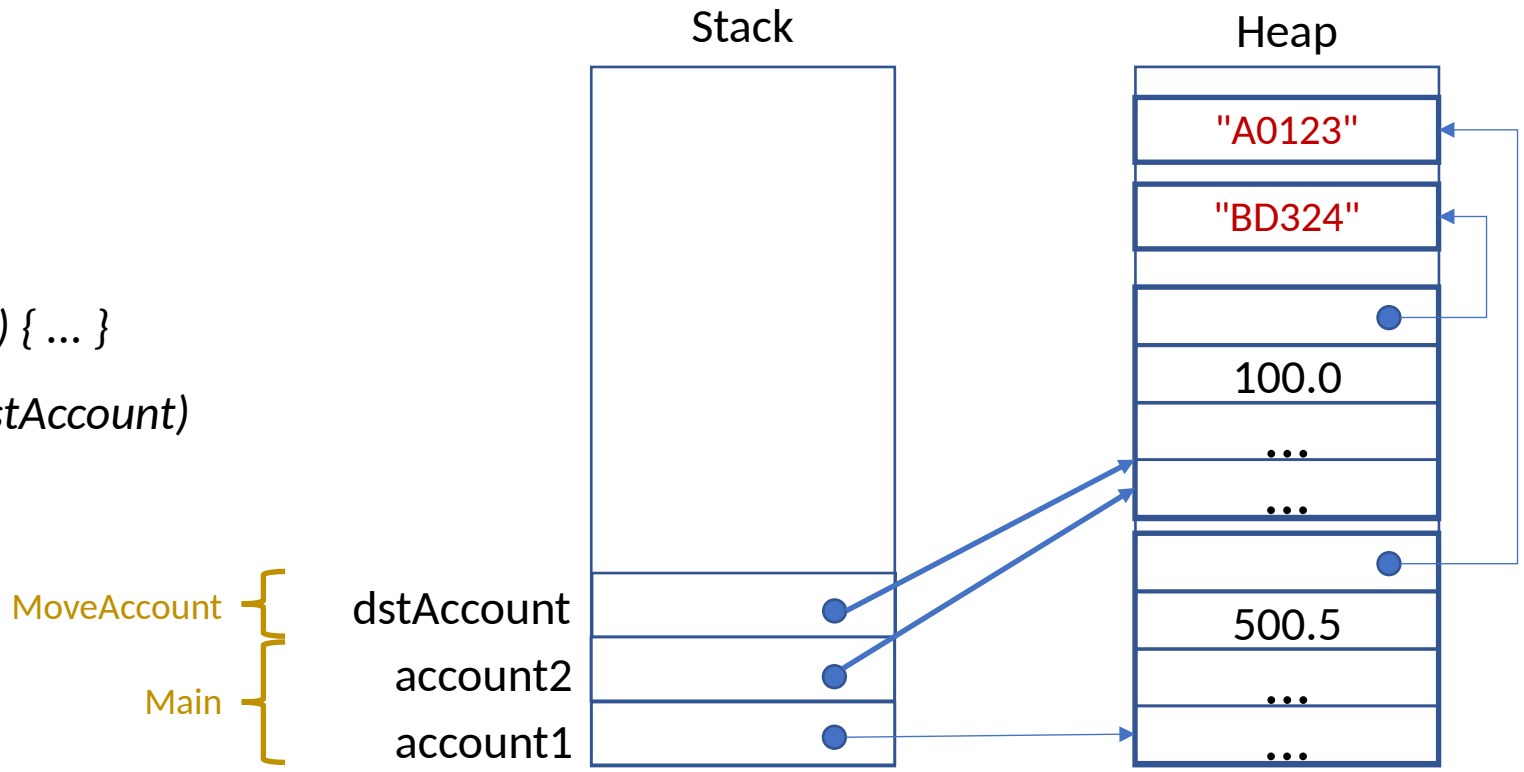
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void MoveAccount(SavingAccount dstAccount)
    {
        dstAccount.Save(balance);
        Close();
    }
}

class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 3.8);
        account1.MoveAccount(account2);
        ...
}
```




when MoveAccount is invoked the reference inside *account2* is copied into *dstAccount* – **not** the actual object

# Methods parameters: reference types

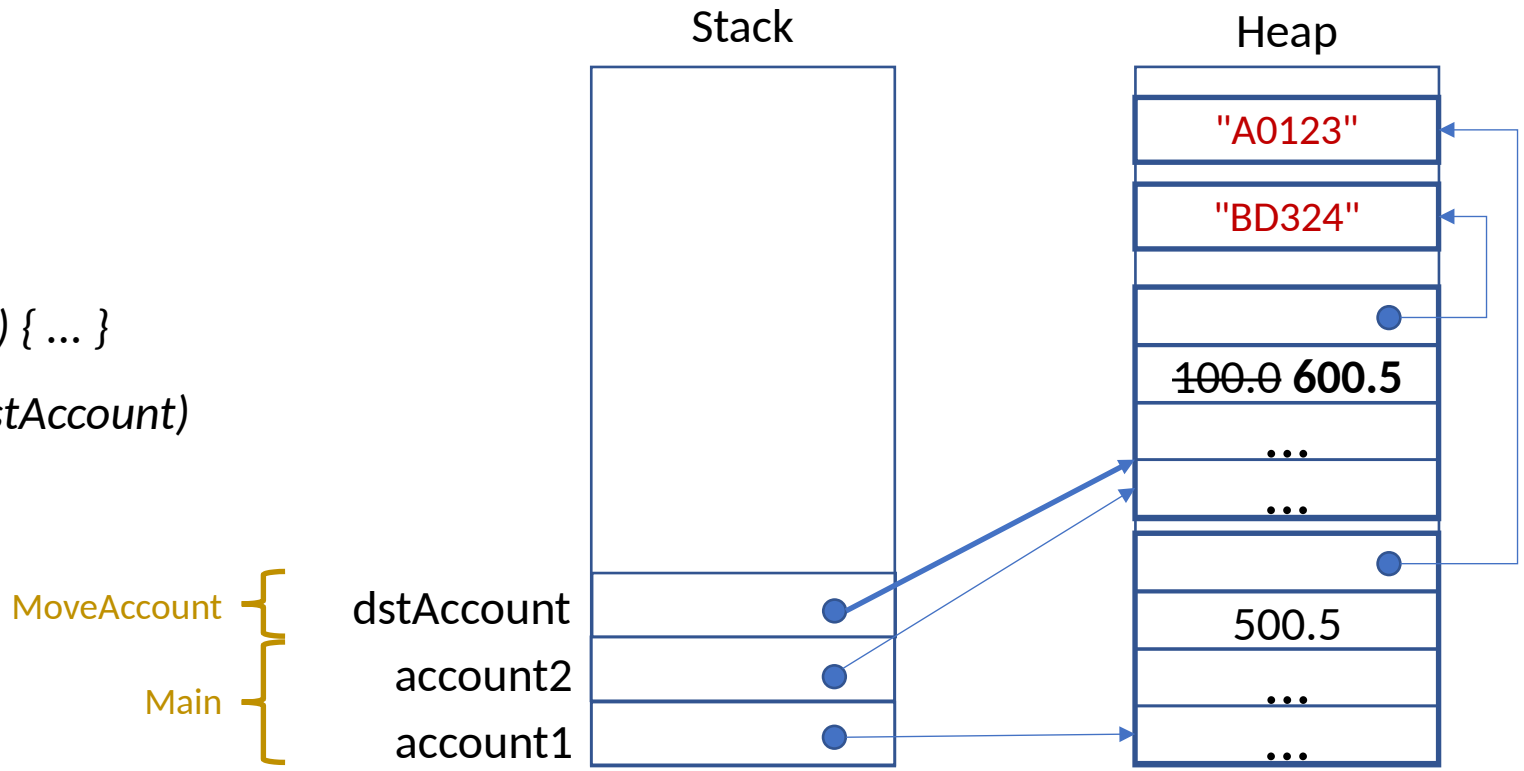
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void MoveAccount(SavingAccount dstAccount)
    {
        dstAccount.Save(balance);
        Close();
    }
}

class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 3.8);
→       account1.MoveAccount(account2);
        ...
}
```




*account2* and *dstAccount* refer to the same object until *MoveAccount* terminates

# Methods parameters: reference types

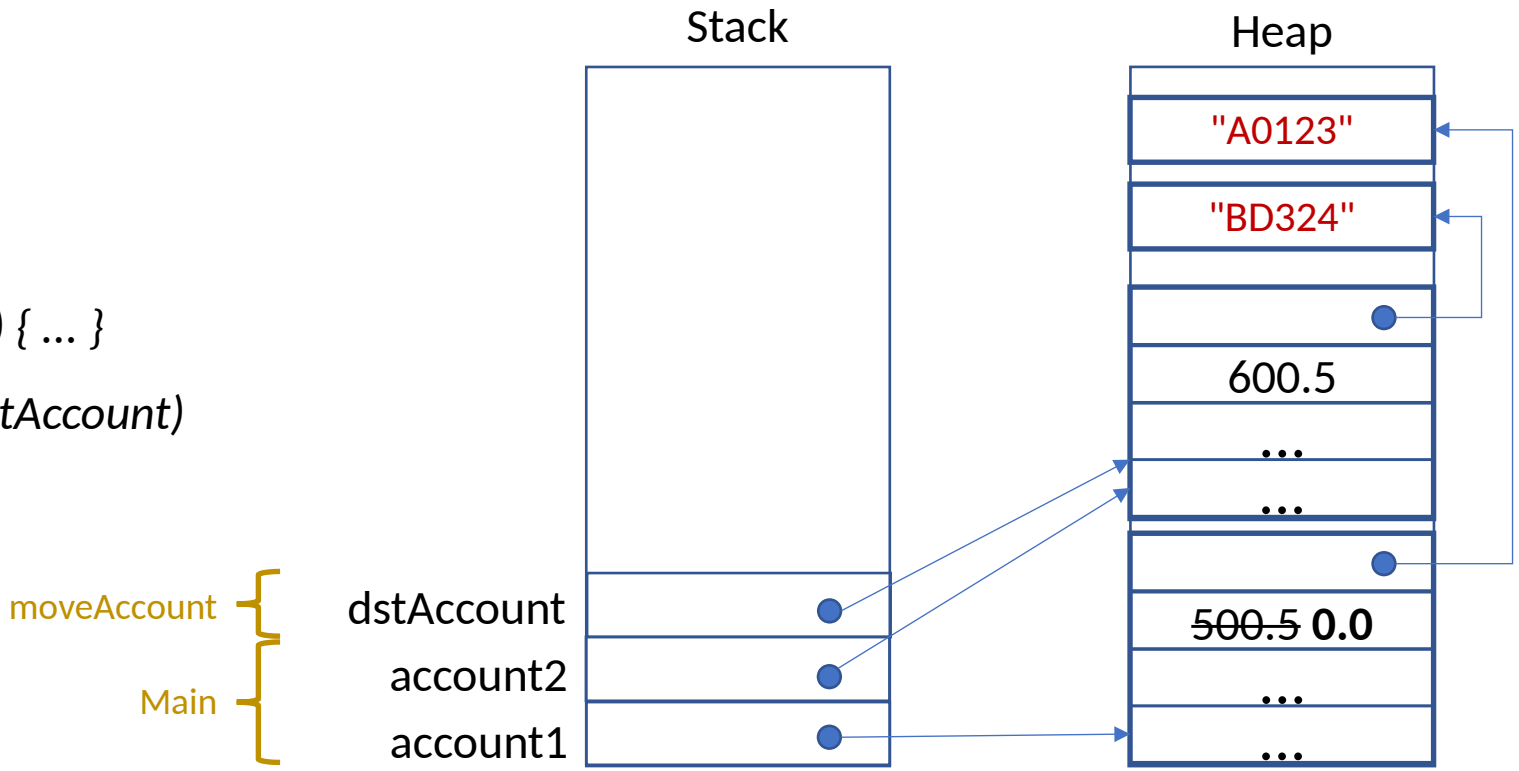
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void MoveAccount(SavingAccount dstAccount)
    {
        dstAccount.Save(balance);
        Close();
    }
}

class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 3.8);
        account1.MoveAccount(account2);
        ...
}
```




*dstAccount* can be used within *MoveAccount* to change the state of the referred object by invoking the *Deposit* method (not shown on the stack)

# Methods parameters: reference types

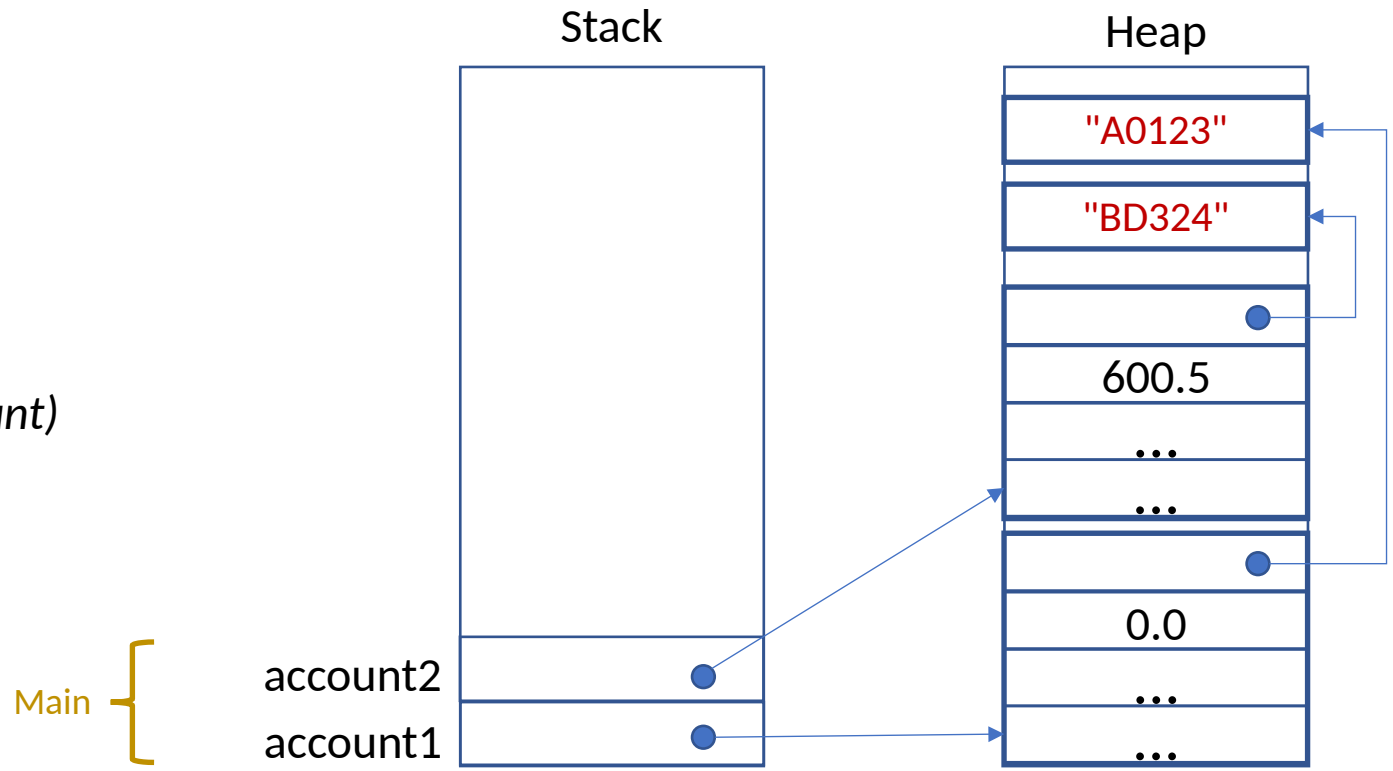
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void MoveAccount(SavingAccount dstAccount)
    {
        dstAccount.Save(balance);
        Close(); // will close account1
    }
}

class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 3.8);
        account1.MoveAccount(account2);
        ...
}
```




invoking the *Close* method of the object instance itself (not shown on the stack)

# Methods parameters: reference types

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void MoveAccount(SavingAccount dstAccount)
    {
        dstAccount.Save(balance);
        Close();
    }
}

class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 3.8);

        account1.MoveAccount(account2);
        ...
}
```




any operations performed via *dstAccount* on the referenced object will **persist** also **after** the termination of *MoveAccount*

# Methods parameters: reference types

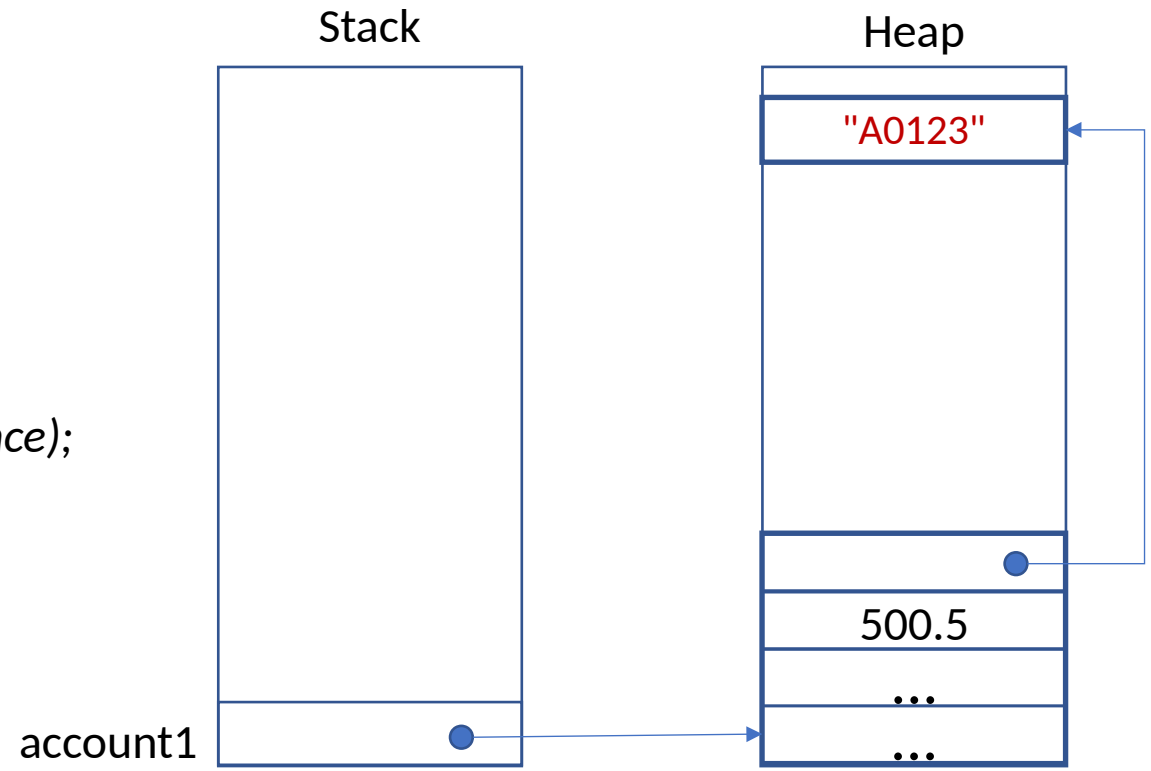
- When a method is invoked, **new variables** are created on the stack according to the number of *parameters*
- A local **copy** of the values provided as argument is stored into them
- **References** to objects are copied – **not the actual objects**
- That reference can be used **inside** the method to **send a message to** the referred object
- The effect of the invocation will then **persist** after the method terminates

# Variables and Objects Lifetime

```
class BankAccount
{
    private string number;
    private double balance;

    // ... all the methods defined in the tutorial

    public void CloneAccount()
    {
        BankAccount  clonedAcc = new BankAccount(number, balance);
        // do more things
    }
}
```

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

Main

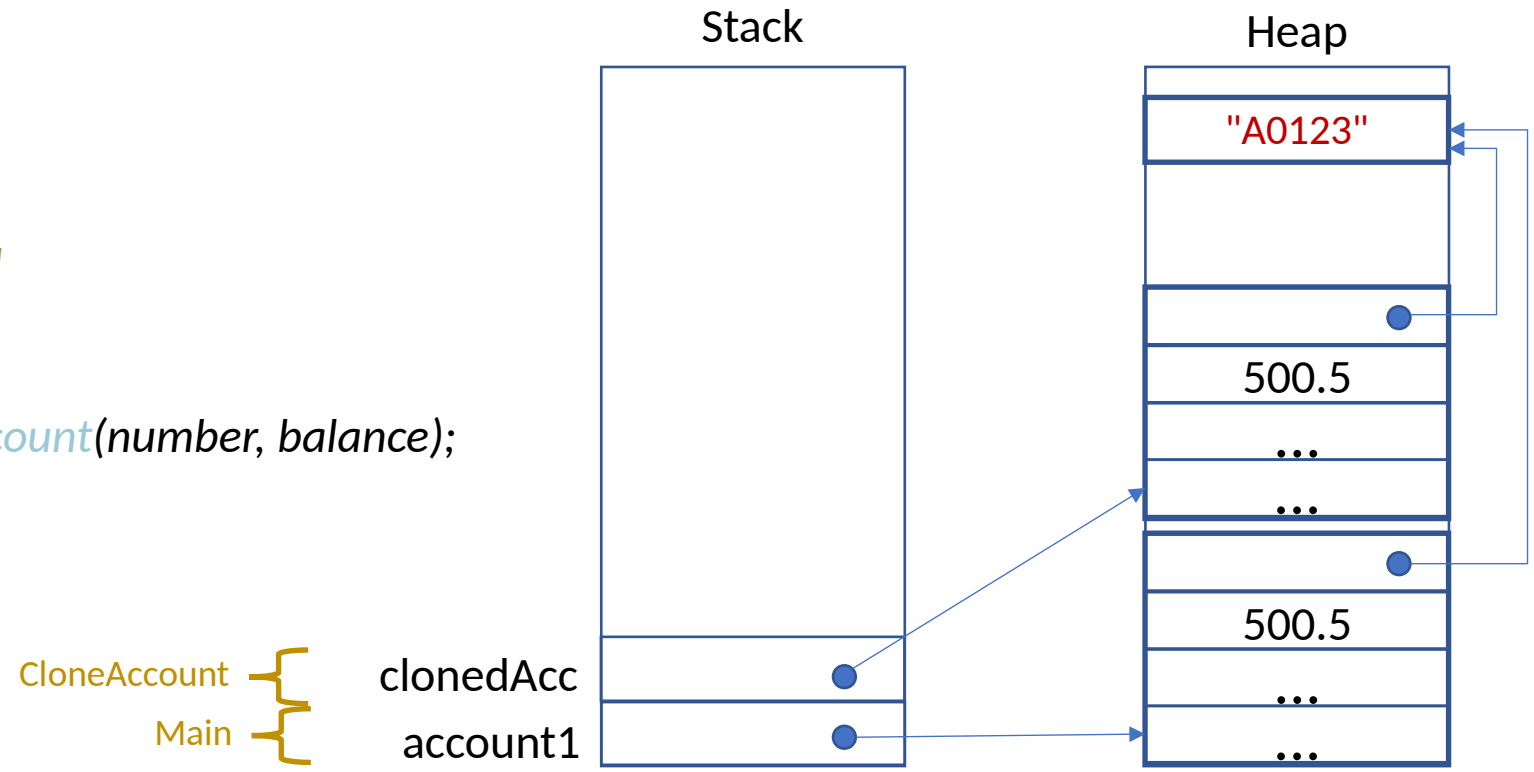



*CloneAccount* creates a (backup) copy of the account object on which is is invoked

# Variables and Objects Lifetime

```
class BankAccount
{
    private string number;
    private double balance;

    // ... all the methods defined in the tutorial

    public void CloneAccount()
    {
        BankAccount  clonedAcc = new BankAccount(number, balance);
        // do more things
    }
}
```

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



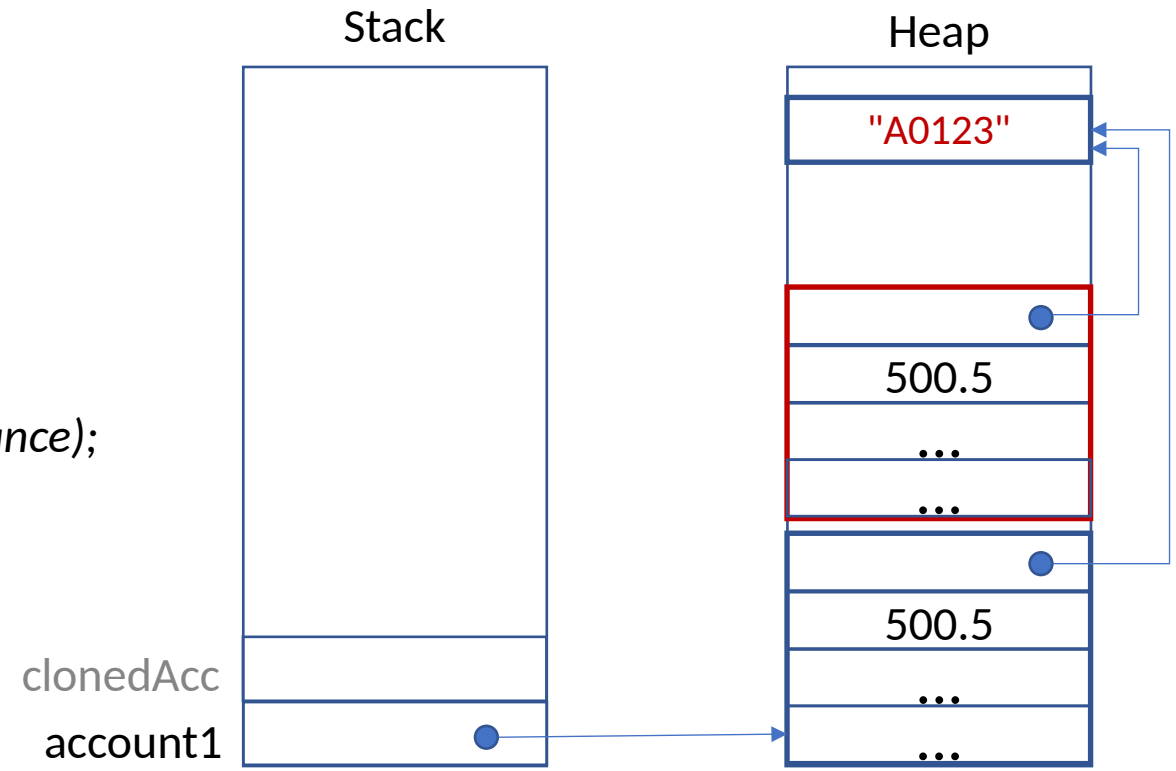

inside *CloneAccount*, a new object is allocated (heap) and a reference is assigned to the local reference type *clonedAcc* (stack)

*then // more things are performed*

# Variables and Objects Lifetime

```
class BankAccount
{
    private string number;
    private double balance;

    // ... all the methods defined in the tutorial

    public void CloneAccount()
    {
        BankAccount  clonedAcc = new BankAccount(number, balance);
        // do more things
    }
}
```

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



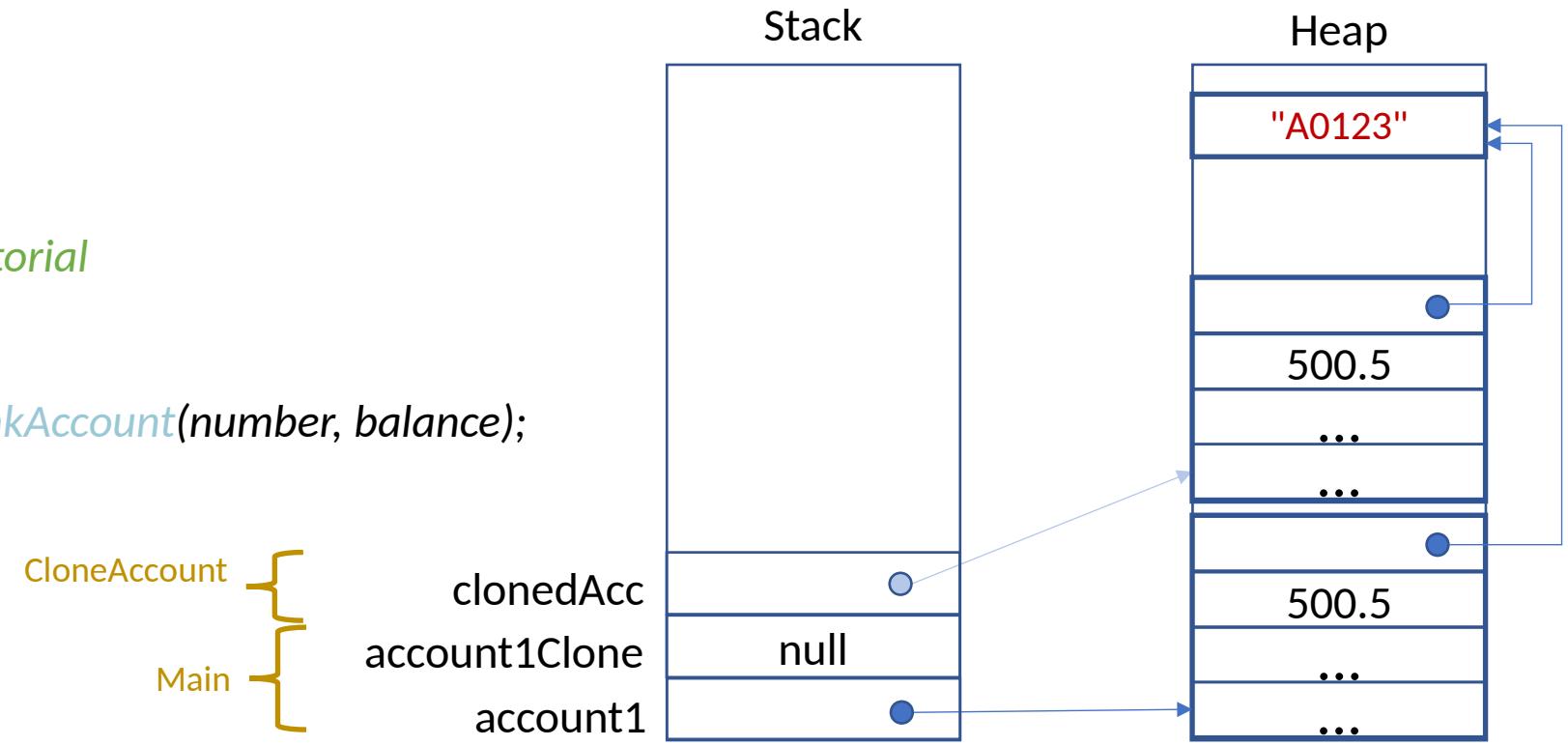

when *CloneAccount* terminates, its stack area is removed, so there will be no variables referring to the cloned object (could be garbage collected)

# Question

- How can the cloned object **persist** after the method termination?

# Variables and Objects Lifetime

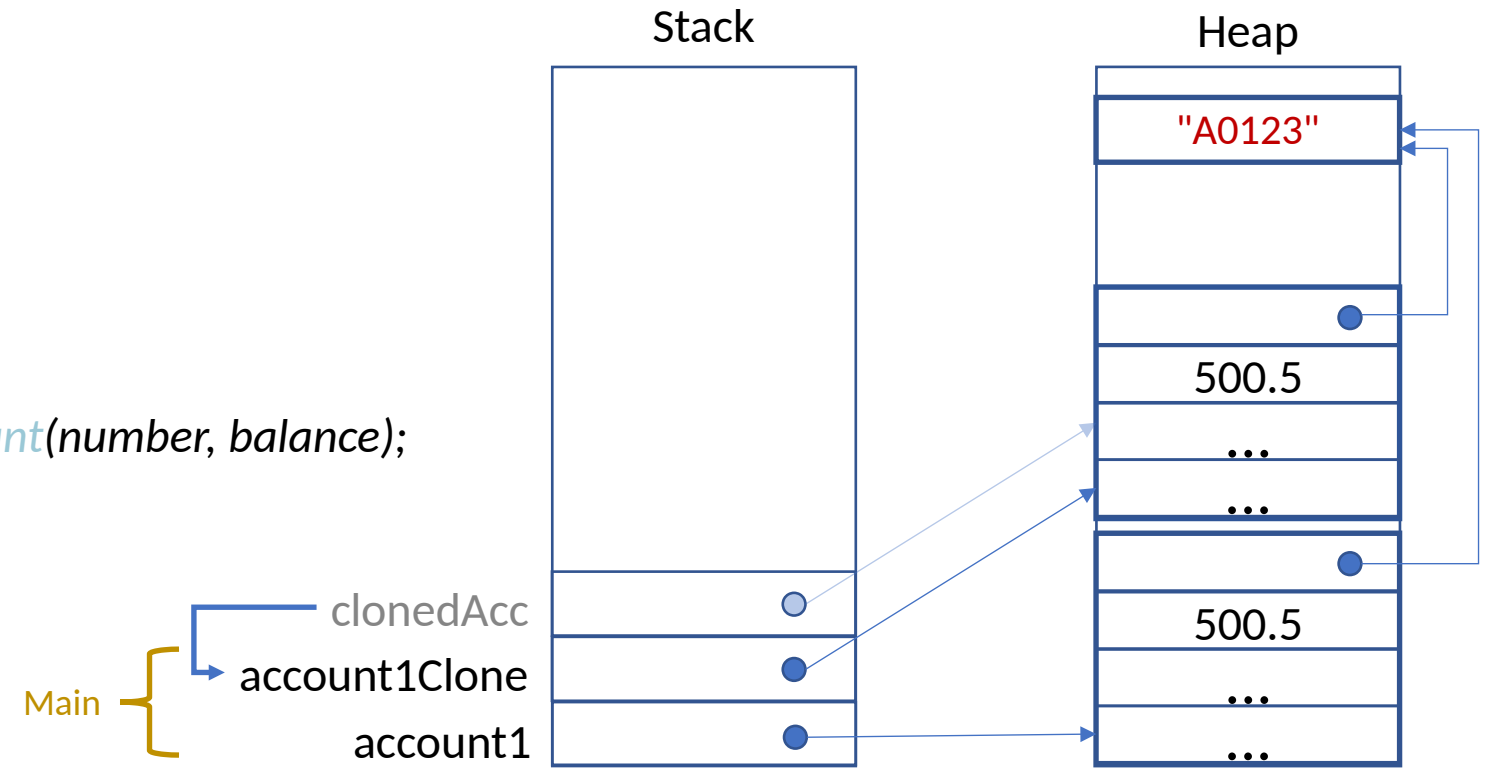
```
class BankAccount
{
    private string number;
    private double balance;

    // ... all the methods defined in the tutorial

    public BankAccount CloneAccount()
    {
        BankAccount  clonedAcc = new BankAccount(number, balance);
        // do more things
        return clonedAcc;
    }
}

class Program
{
    public static void Main(string[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        BankAccount  account1Clone = account1.CloneAccount();
        ...
}
```




*CloneAccount* should return the reference to the newly created object to the *Main*.

# Variables and Objects Lifetime

```
class BankAccount
{
    private string number;
    private double balance;

    // ... all the methods defined in the tutorial

    public BankAccount CloneAccount()
    {
        BankAccount  clonedAcc = new BankAccount(number, balance);
        // do more things
        return clonedAcc;
    }
}


class Program
{
    public static void Main(string[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        BankAccount  account1Clone = account1.CloneAccount();
        ...
}
```




The *Main* should store it inside a local reference type variable (e.g., *account1Clone*)

# Stack vs Heap: summary

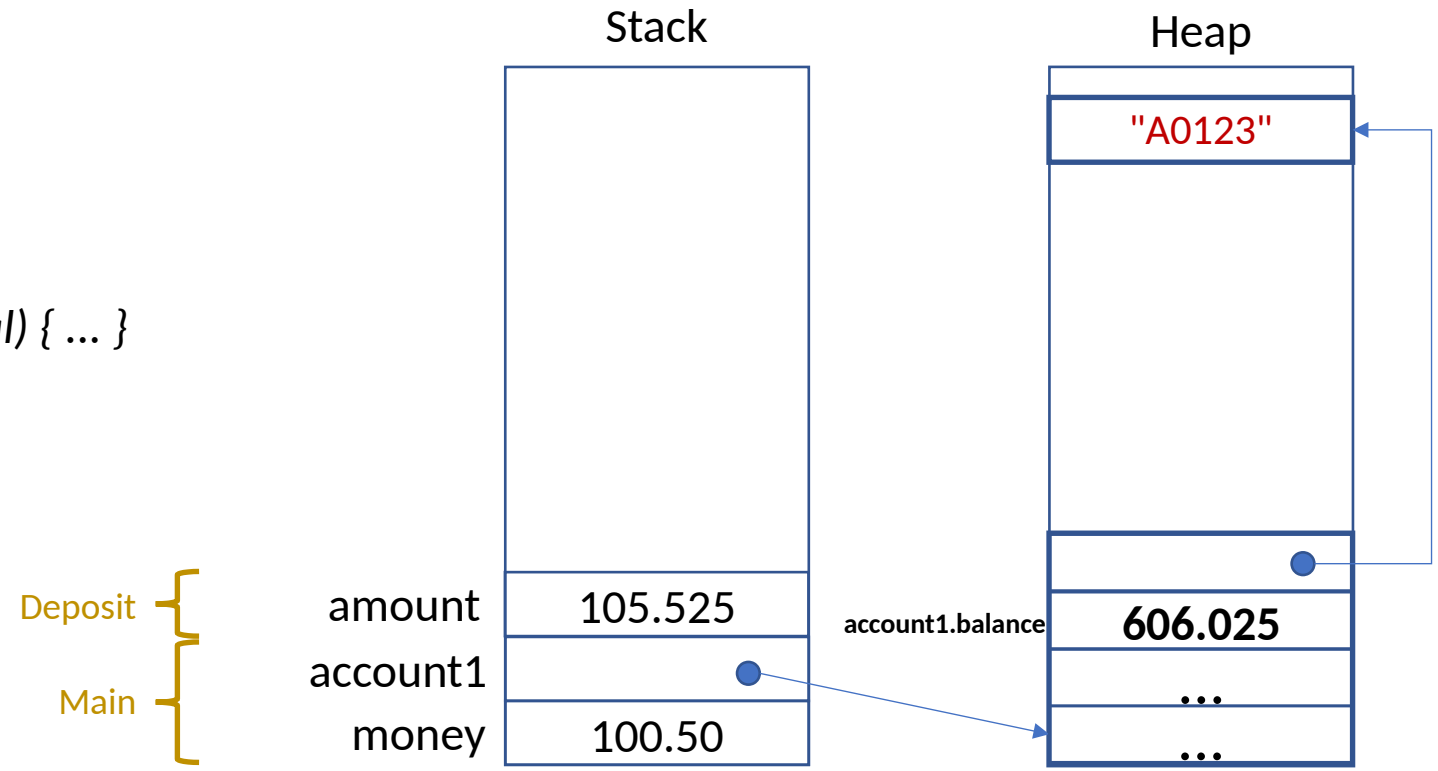
- ***Local variables*** allocated in the stack have the **same lifetime** of the method they belong to
- ***Objects*** allocated in the heap may have a **longer** lifetime than local variables
- They are removed from the heap by the *GC* only when there are **no reference type variables** in the program that **refer** to them

# Outline

- More on *Value Types* and *Reference Types*
  - '=' and '==' operators
  - Method invocation and parameter passing
- The *this* keyword
- More on *Encapsulation*

# this keyword

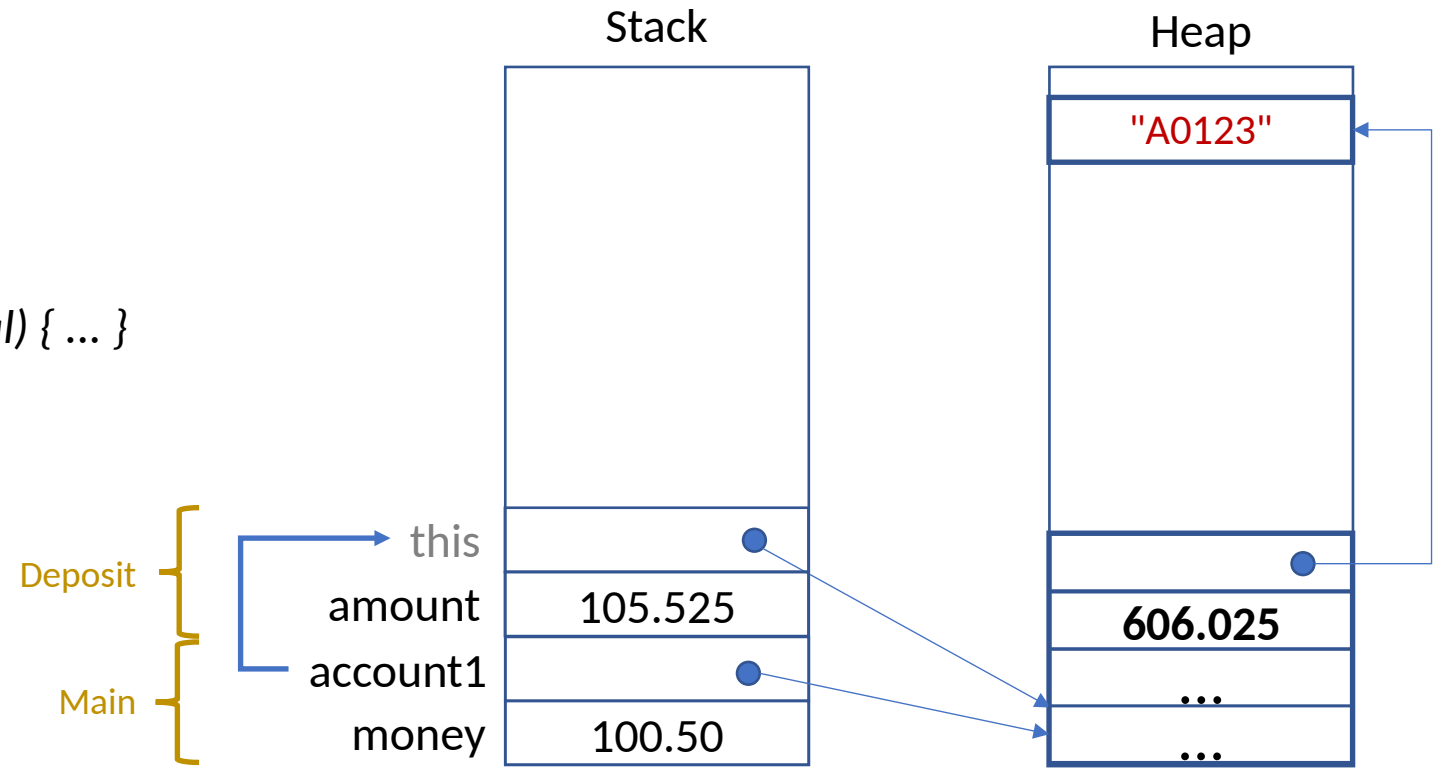
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05 // e.g., an interest rate
        balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(money);
        ...
}
```




the Deposit method changes the *balance* attribute of the object *account1*, on which it was invoked: account1.*Deposit*(money)

# this keyword

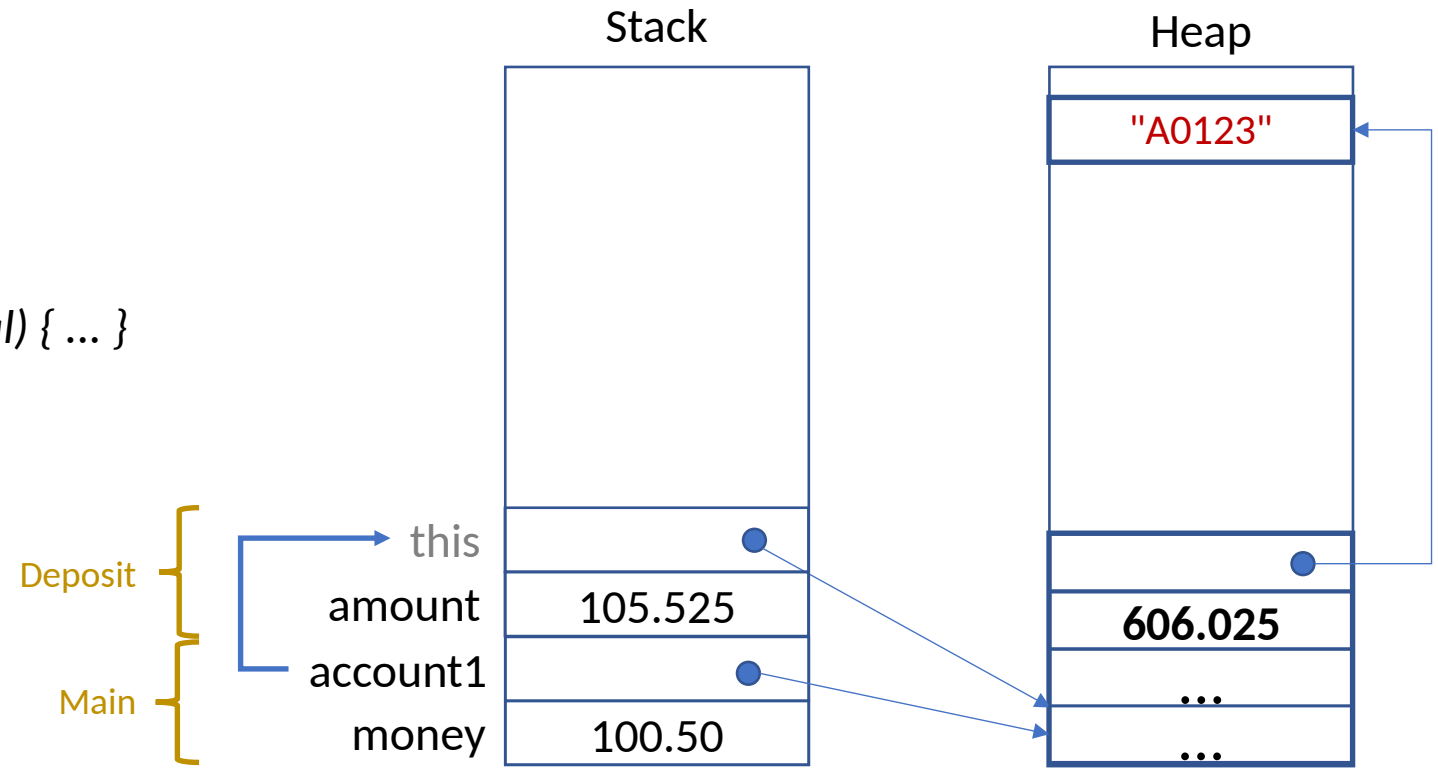
```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(this, double amount)
    {
        amount *= 1.05 // e.g., an interest rate
        this.balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(account1, money);
        ...
}
```




the Deposit method can change *amount* because it implicitly receives a reference to *account1*: this

# this keyword

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        amount *= 1.05 // e.g., an interest rate
        this.balance += amount;
    }
}

class Program
{
    public static void Main()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.Deposit(money);
        ...
}
```




the Deposit method can change *amount* because it implicitly receives a reference to *account1*: **this**

# this keyword: how can be used?

- To refer to the **current instance (object)** of the class

# this keyword: how can be used?

- To refer to the **current instance (object)** of the class
- To qualify **attributes** hidden by similar names

```
class BankAccount
{
    private string number;
    private double balance;

    public BankAccount(string number, double balance)
    {
        this.number = number;
        this.balance = balance;
    }

    ...
}
```

# this keyword: how can be used?

- To refer to the **current instance (object)** of the class
- To qualify **attributes** hidden by similar names
- To **pass** an object as a parameter to methods of other classes

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount)
    {
        Logger.LogInfo(this);
        balance += amount;
    }
}
```

```
class Logger
{                                                   account1
    public static void LogInfo(BankAccount account)
    {
        // logs object account transaction into a DB
    }
}

class Program
{
    public static void Main()
    {
        BankAccount  account1 = new BankAccount( ... );
        account1.Deposit(100.50);
}
```

# this keyword: how can be used?

- To refer to the **current instance (object)** of the class
- To qualify **attributes** hidden by similar names
- To **pass** an object as a parameter to methods of other classes

- There is one more possible usage of *this*: *constructors chaining*
- Before discussing it, let's remind ourselves of the concept of **method overloading**

# Methods: *overloading*

```
class BankAccount
{
    private string number;
    private double balance;

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

    public BankAccount(string num)
    {
        number = num;
        balance = 0;
    }
}
```

# Methods: *overloading*

```
class BankAccount
{
   private string number;
   private double balance;

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

   public BankAccount(string num)
   {
      number = num;
      balance = 0;
   }
}
```

Both these *constructor methods* have the **same name** (must be as the class name: *BankAccount*)

They have a **different number** of formal parameters (two and one)

We say they are **overloaded**

**Overloading** applies to methods in general (not only constructors)

# this keyword: constructors chaining

```
class BankAccount
{
    private string number;
    private double balance;

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

    public BankAccount(string num)
    {
        number = num;
        balance = 0;
    }
}
```

# this keyword: constructors chaining

```
class BankAccount
{
    private string number;
    private double balance;

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

    public BankAccount(string num)
    {
        number = num;
        balance = 0;
    }
}
```

```
class BankAccount
{
    private string number;
    private double balance;

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

    public BankAccount(string num)
            : this(num, 0)
    {
    }
}
```

# this keyword: constructors chaining

```
class BankAccount
{
    private string number;
    private double balance;

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

    public BankAccount(string num)
    {
        number = num;
        balance = 0;
    }
}
```

```
class BankAccount
{
    private string number;
    private double balance;

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

    public BankAccount(string num)
        : this(num, 0)
    {
    }
}
```

# Methods: *overloading*

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount) { ... }
                                                ?
    public void Deposit(double money) { ... }
```

```
...

BankAccount  account1 = new BankAccount("A0123", 500.5);

account1.Deposit(200.0);
```

These *methods* have the **same name** *Deposit*
They also have the **same number** (one) and **type** (double) of formal parameters

Which one should the compiler choose?

# Methods: *overloading*

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount) { ... }

    public void Deposit(double money,
                        double interest) { ... }
```

```
...

BankAccount  account1 = new BankAccount("A0123", 500.5);

account1.Deposit(200.0);

account1.Deposit(200.0, 0.5);
```

# Methods: *overloading*

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount) { ... }

    public void Deposit(double money,
                        double interest) { ... }
```

```
...

BankAccount  account1 = new BankAccount("A0123", 500.5);

account1.Deposit(200.0);

account1.Deposit(200.0, 0.5);
```

The choice of the method to be invoked is done at **compile time** by matching arguments with type/number of parameters

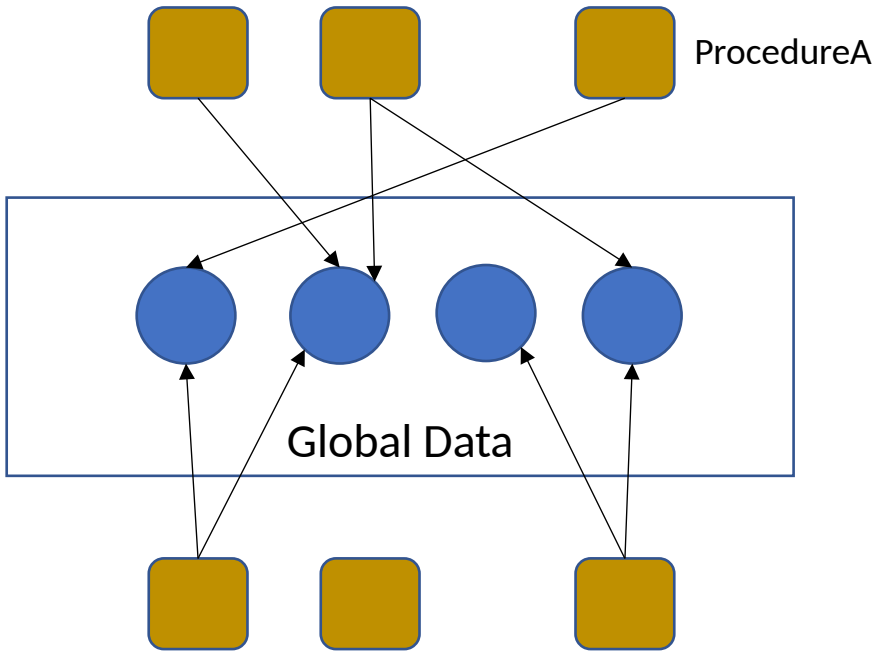
Method overloading is considered a form of *static polymorphism* or *compile-time polymorphism*

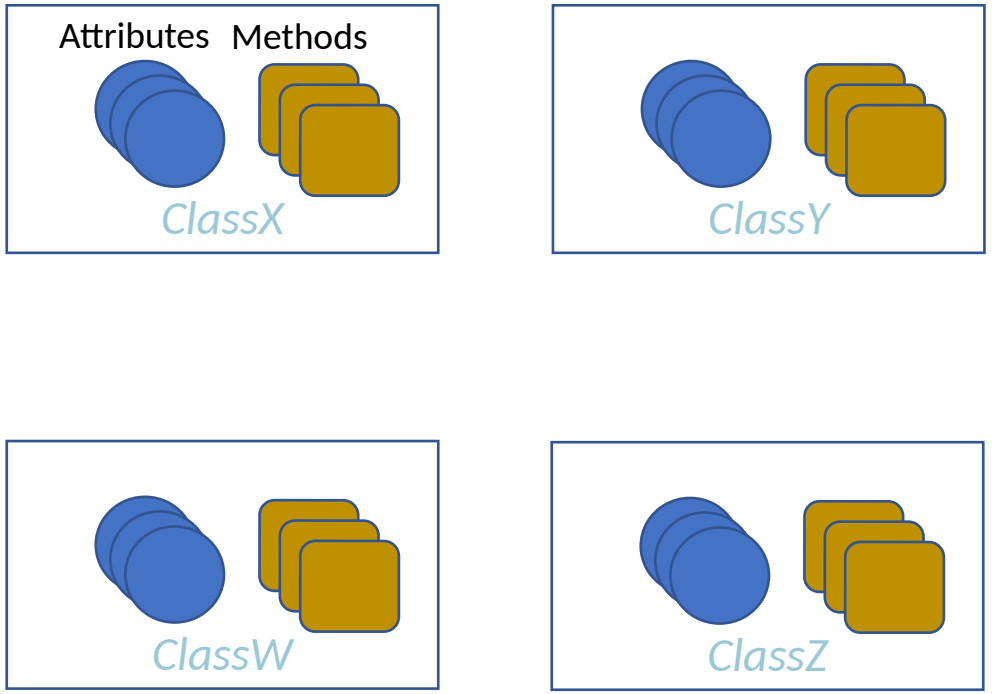
# Outline

- More on *Value Types* and *Reference Types*
  - '=' and '==' operators
  - Method invocation and parameter passing
- The *this* keyword
- More on *Encapsulation*

# Modules: Procedures vs Classes

Program Modules: Procedures



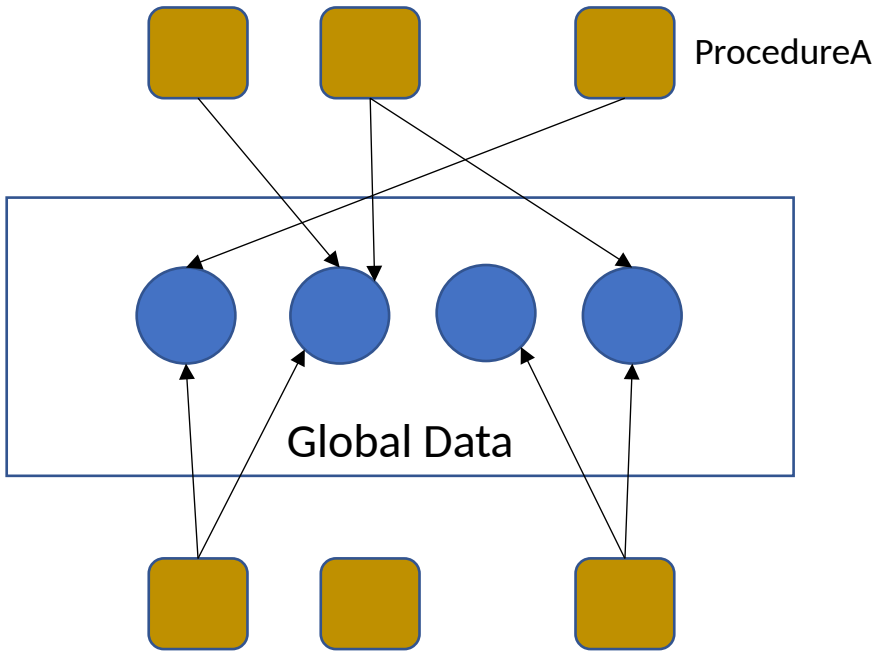



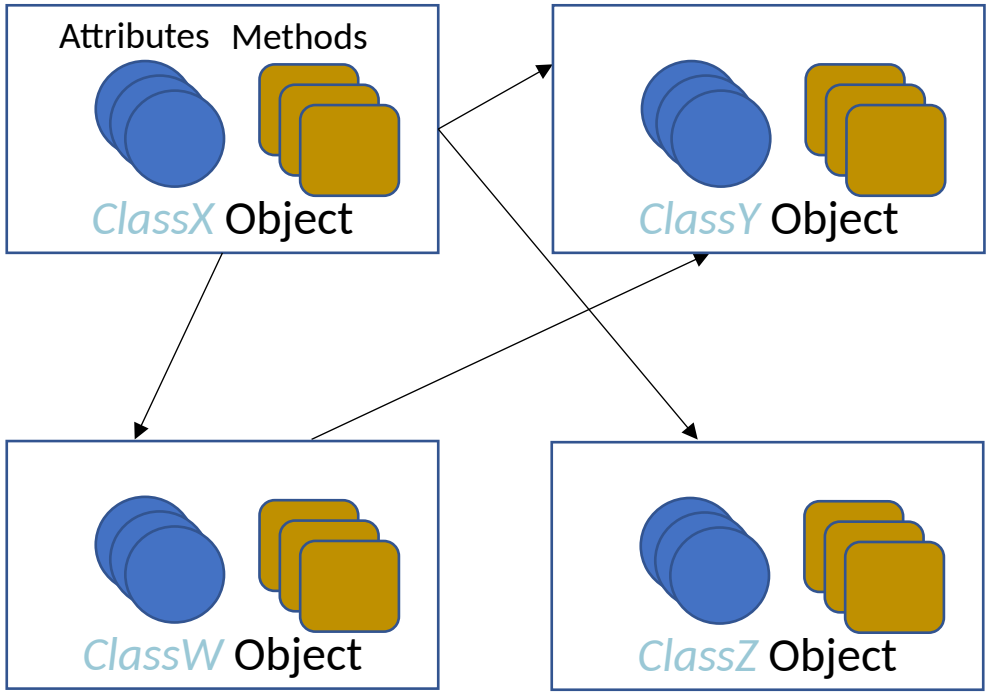

Procedures are like methods but can **operate on globally shared data**

A class defines **encapsulated** *attributes* and *methods*, along with their **visibility** to other classes

# Modules: Procedures vs Classes

Program Modules: Procedures

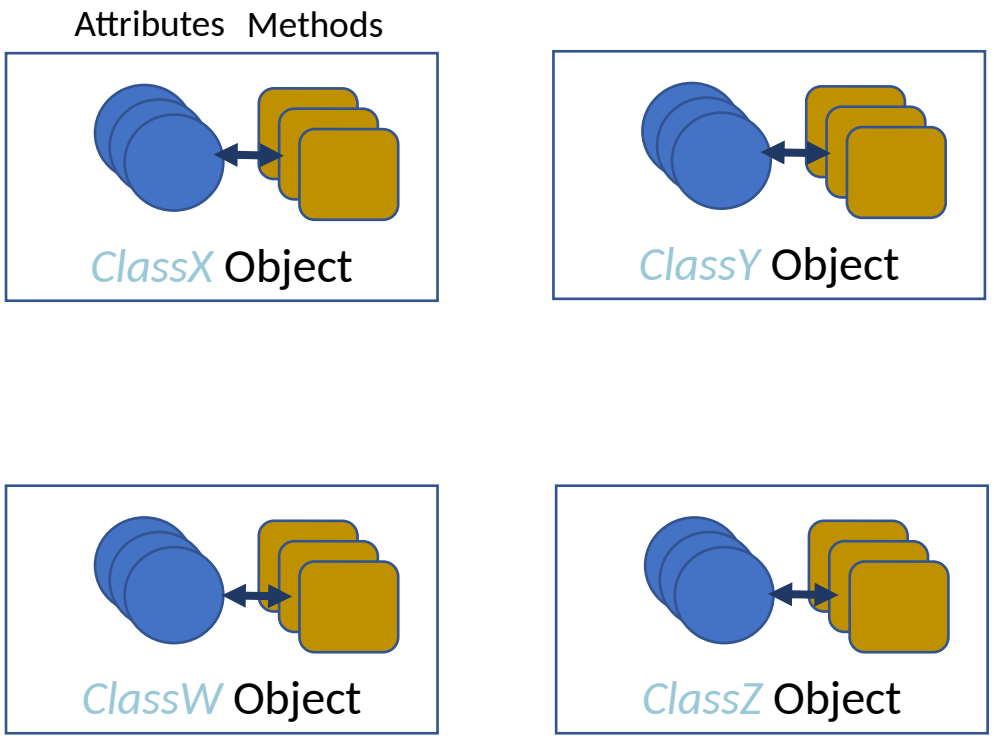



Procedures are like methods but can **operate on globally shared data**

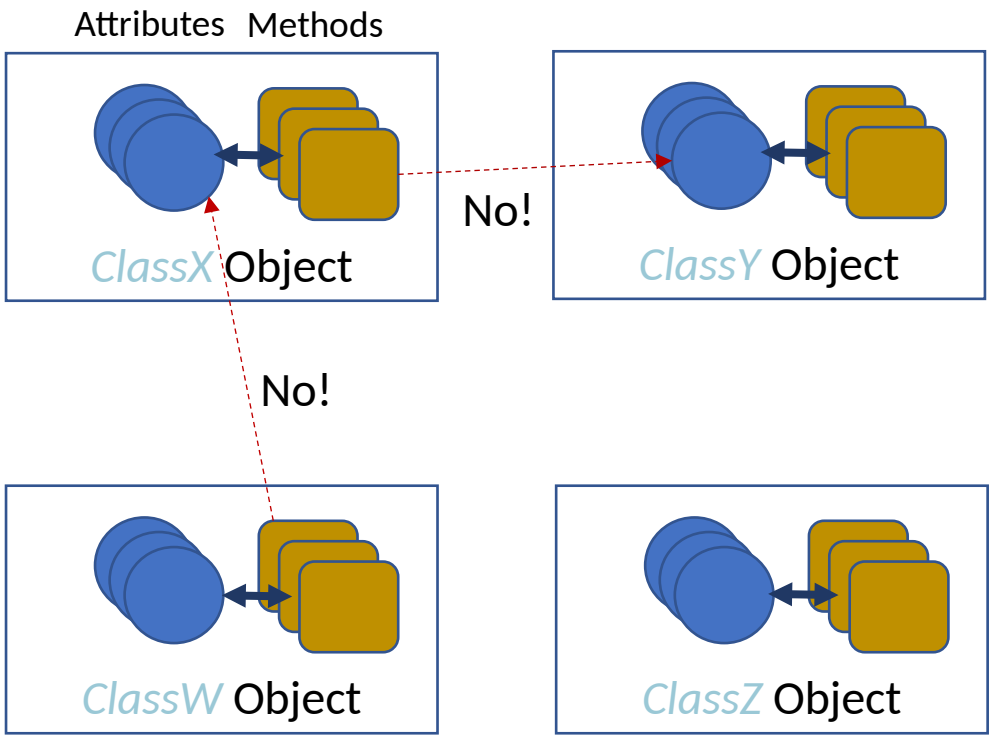
**Objects** of those classes interact with each other based on the **visibility** of *attributes* and *methods*

# Objects communication



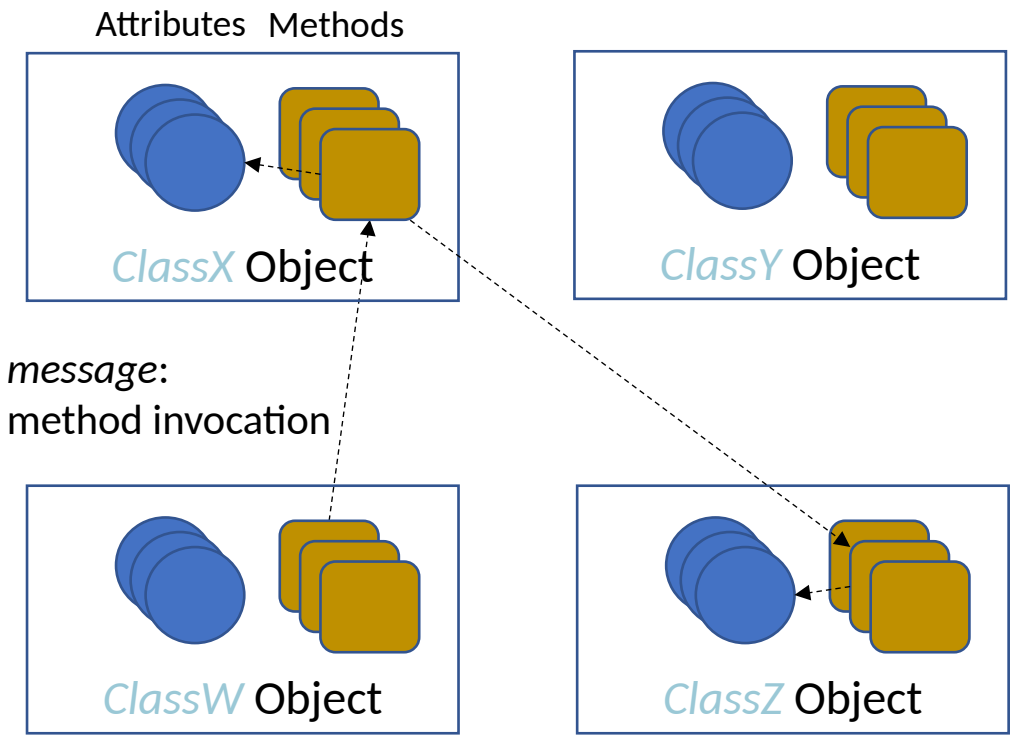

**Attributes** of an object are directly accessible by methods of objects of the same class

# Objects communication



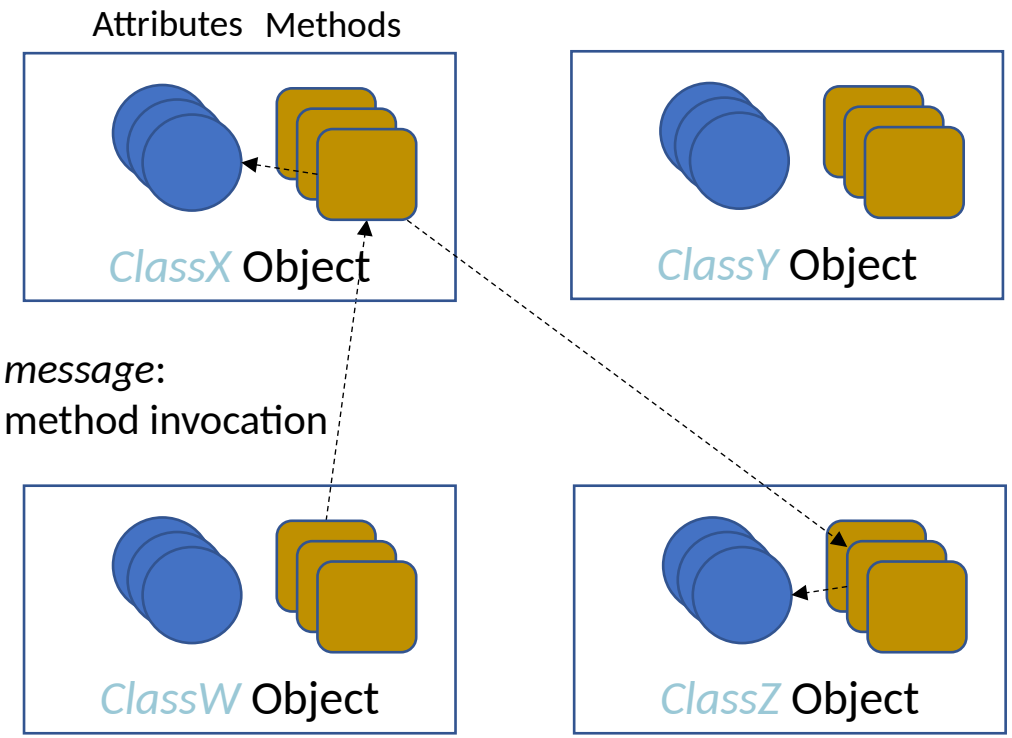

An object of a class should **not have direct access** to **attributes** of objects of other classes

# Objects communication




Objects of different classes interact with each other by invoking **methods**—they *"send messages"*

# Objects communication




Objects of different classes interact with each other by invoking **methods**—they *"send messages"*

How can (did) we enforce these *visibility* and *access control* **rules** in C#?

# Answer: access modifiers

- **Keywords** that define the *visibility* and *accessibility* of class **members** —*attributes* and *methods*—from outside the class
- *private* and *public* (so far)
- *internal* and *protected* (later)

# Access modifiers

```
class BankAccount
{
    private string number;
    private double balance;

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

    public void Deposit(double amount) { ... }

    public bool Withdraw(double amount) { ... }
```

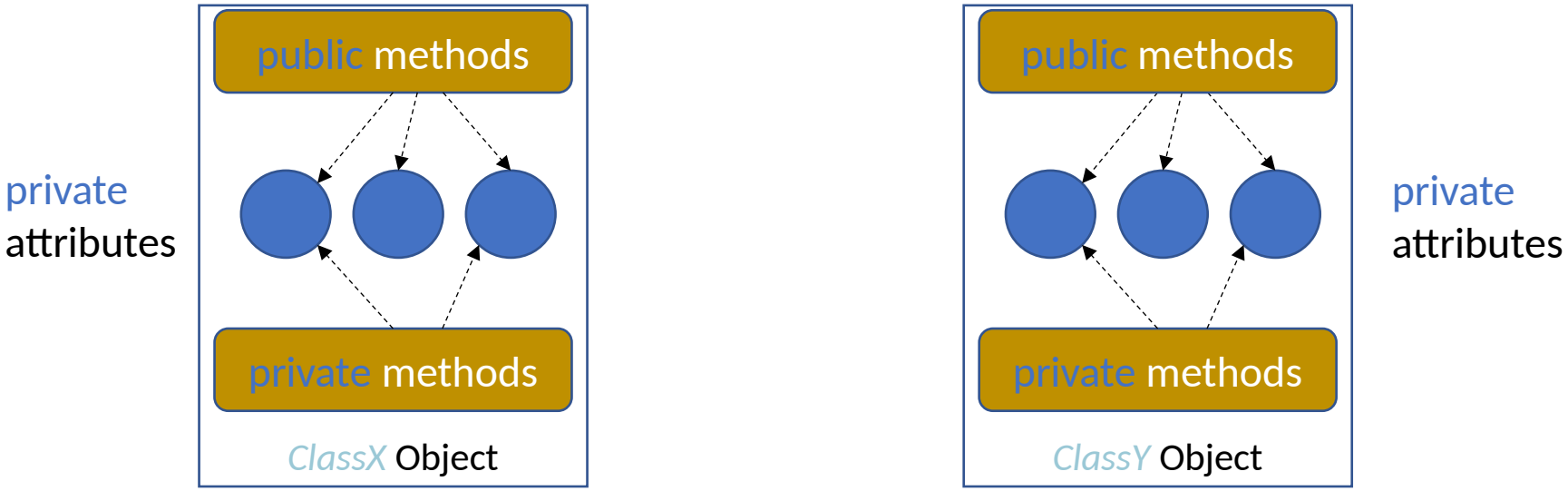
Why did we use *private* for the **attributes**?

Why did we use *public* for the **methods**?

# Access modifiers

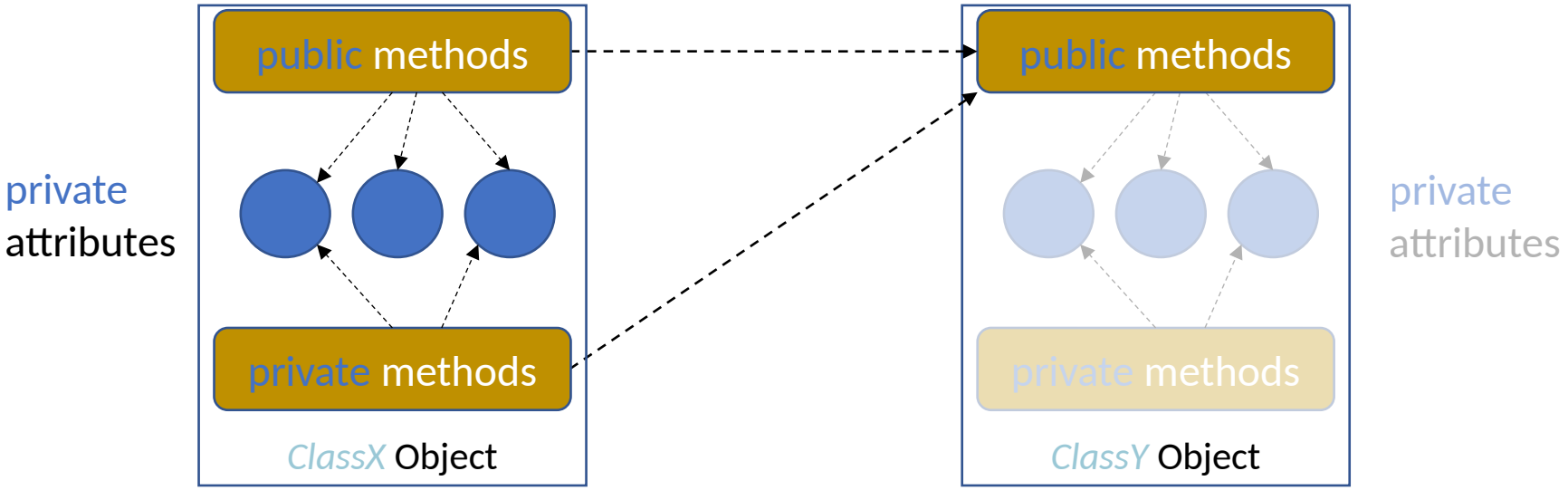
- private members are **only** accessible by the code of the class where they are declared
- public members are accessible by the code of any class of the same program

# Objects interface



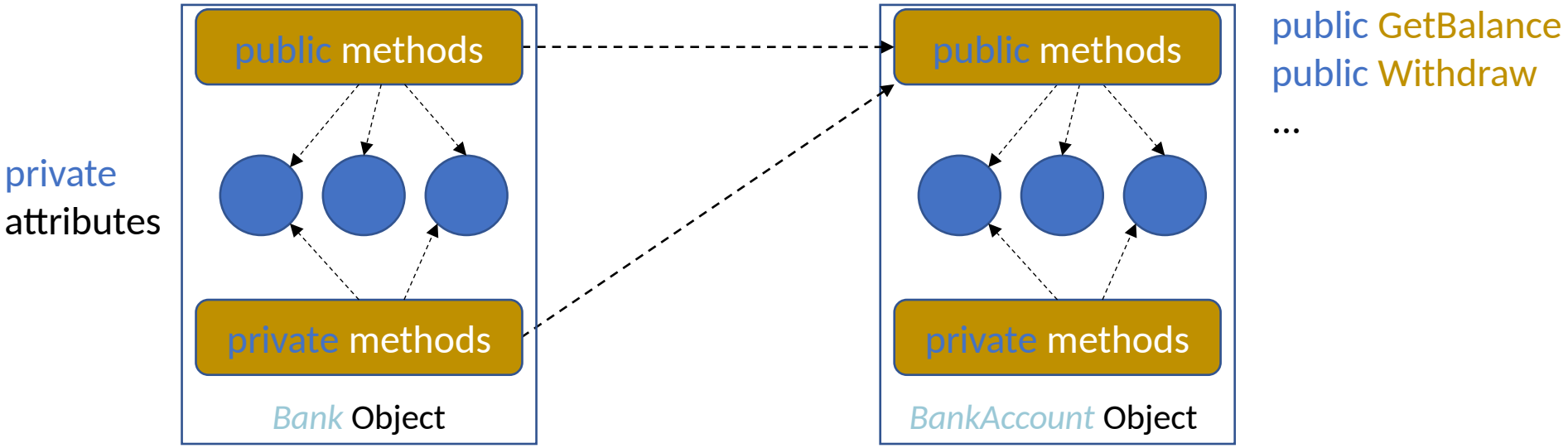

- Each object usually has private *attributes* and public *methods*, as defined by the associated class
- An object may also have private *methods*

# Objects interface



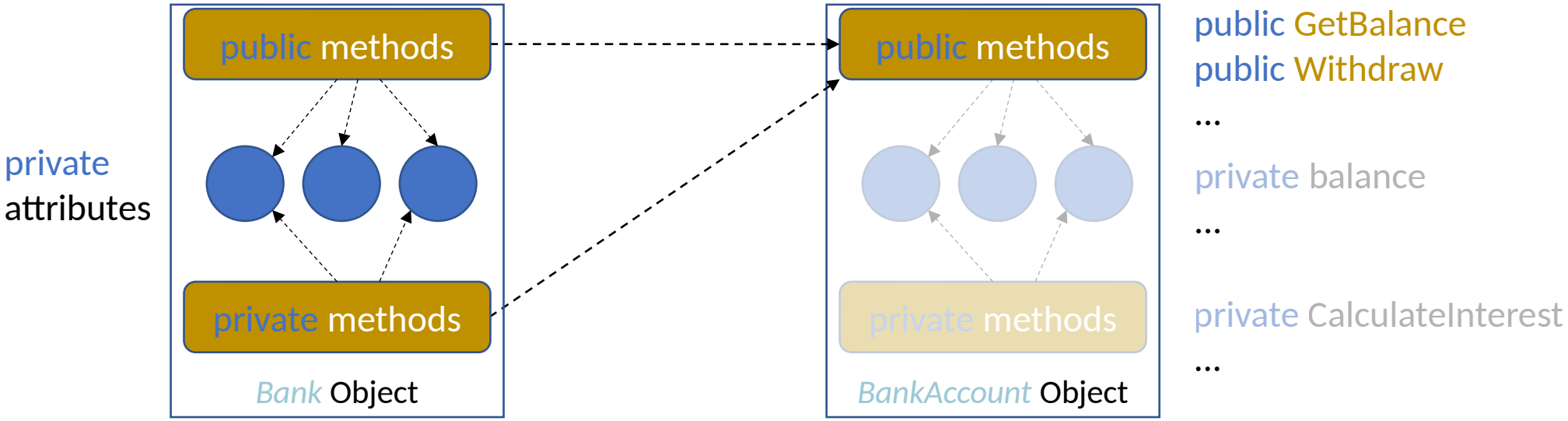

- public *methods* define the **interface** an object exposes to objects of other classes
- private *members* are **hidden** and **not accessible** by objects of other classes

# Objects interface example: *BankAccount*




- Interface of a *BankAccount* object: public methods, such as GetBalance, Withdraw, Deposit, etc.

# Objects interface example: *BankAccount*




- Interface of a *BankAccount* object: public methods, such as GetBalance, Withdraw, Deposit, etc.
- private *attributes* and *methods* are hidden from objects of other classes

# If *balance* was public

```
class BankAccount
{
    public string number;
    public double balance;
    ...
}

class Program
{
    public static void Main(string args)
    {
        BankAccount  account1 = new BankAccount("A0123", 1000);

        double amount = 1500;
        account1.balance = account1.balance - amount;
        // balance is negative: -500

}
```

it could be accessed by using the **dot** notation as we did for public methods

# If *balance* was public

- **Any** class could manipulate the *balance* attribute directly
- **Bypass** any required **checks**: **errors** and **anomalies**
- **Each** class to **implement** and **replicate** those checks

- **Encapsulation** of *attributes* and *methods*
- *balance* is private and can only be changed via the public methods of *BankAccount*

# The right way: *encapsulation*

use private access modifier

```
class BankAccount
{
   private string number;
   private double balance;

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

   public bool Withdraw(double amount)
   {
       if (amount <= balance )
       {
          balance -= amount;
          return true;
       }
      return false;
   }
}
```

# The right way: *encapsulation*

```
class BankAccount
{
    private string number;
    private double balance;

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

    public bool Withdraw(double amount)
    {
        if (amount <= balance )
        {
            balance -= amount;
            return true;
        }
        return false;
    }
}
```

- Logic to check *balance* implemented in a **single place**
- Every other class must use Withdraw
- Prevents *errors* and *anomalies* in the program

# The right way: *encapsulation*

```
class Program
{
   public static void Main(string args)
   {
      BankAccount  account1 = new BankAccount("A0123", 1000);

      double amount = 1500;
      // account1.balance = account1.balance - amount;

      if ( account1.Withdraw(amount) == true )
         Console.WriteLine("Transaction approved!");
      else
         Console.WriteLine("Transaction refused!");
}
```

It would produce an error because *balance* is private in *BankAccount*

# The right way: *encapsulation*

```
class Program
{
   public static void Main(string args)
   {
      BankAccount  account1 = new BankAccount("A0123", 1000);

      double amount = 1500;
      // account1.balance = account1.balance - amount;

      if ( account1.Withdraw(amount) == true )
         Console.WriteLine("Transaction approved!");
      else
         Console.WriteLine("Transaction refused!");
}
```

The *Program* class cannot arbitrarily modify the value of the *balance* attribute of the *account1* object

# The right way: *encapsulation*

```
class Program
{
   public static void Main(string args)
   {
      BankAccount  account1 = new BankAccount("A0123", 1000);

      double amount = 1500;
      // account1.balance = account1.balance - amount;

      if ( account1.Withdraw(amount) == true )
         Console.WriteLine("Transaction approved!");
      else
         Console.WriteLine("Transaction refused!");
}
```

The withdrawal can only be done by calling the public *Withdraw* method on the *account1* object

# Object-Oriented Programming (OOP) Principles

- Abstraction
- **Encapsulation**
- Inheritance
- Polymorphism

Objects **contain attributes and behaviours**— they can **control** how these are **accessed** and **hide** their implementation from objects of other classes.

# Object-Oriented Programming (OOP) Principles

- Abstraction
- **Encapsulation**
- Inheritance
- Polymorphism

Objects **contain attributes and behaviours**— they can **control** how these are **accessed** and **hide** their implementation from objects of other classes to prevent *errors* and *anomalies*.

# Object-Oriented Programming (OOP) Principles

- Abstraction
- **Encapsulation**
- Inheritance
- Polymorphism

Objects **contain attributes and behaviours**— they can **control** how these are **accessed** and **hide** their implementation from objects of other classes to prevent *errors* and *anomalies*.

# Encapsulation: implementation hiding

```
class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private string address; // implemented as a single string attribute

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

    // other methods...

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


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

From Week 5 Tutorial

# Encapsulation: Getter and Setter methods

- private attributes that need to be accessed externally typically have associated Get<Attribute> and Set<Attribute> methods

- The *address* attribute of the *Person* class
  - GetAddress() { … } getter method: returns *address*
  - SetAddress(string addr) { … } setter method: assigns addr to *address*

- *Getter* and *Setter* methods allow enforcing *information* and *implementation hiding*

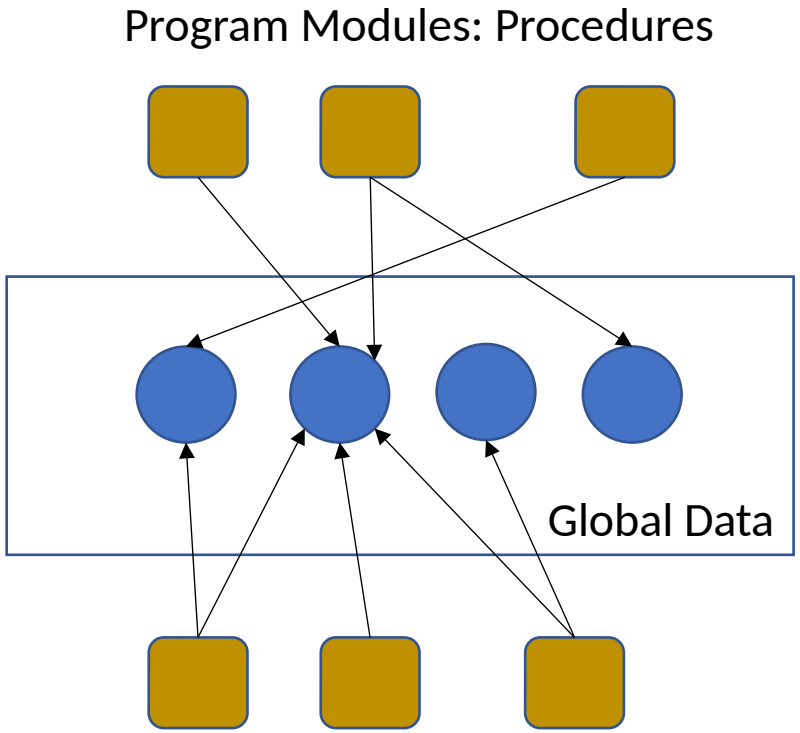
# Encapsulation: implementation hiding

- **Keep** the same interface (public methods signatures) of an object
- Can **change** the implementation of what is *underneath* that interface
- **No need to modify** the **implementation** of any objects **that use that interface**

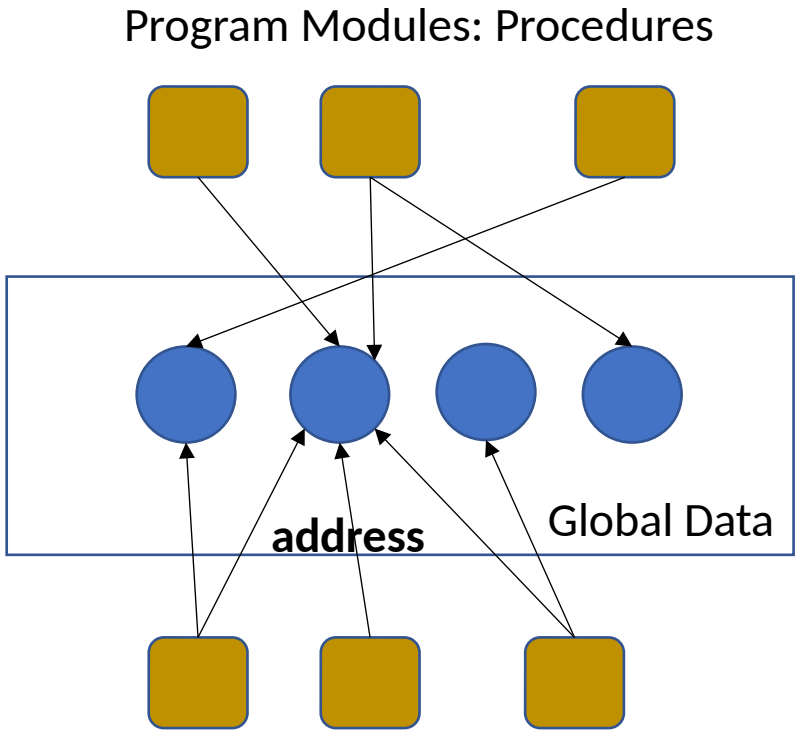
*Example*

- **Change the type** of the *address* attribute of the *Person* class
- Should **all** the classes that use *Person* be **modified**?

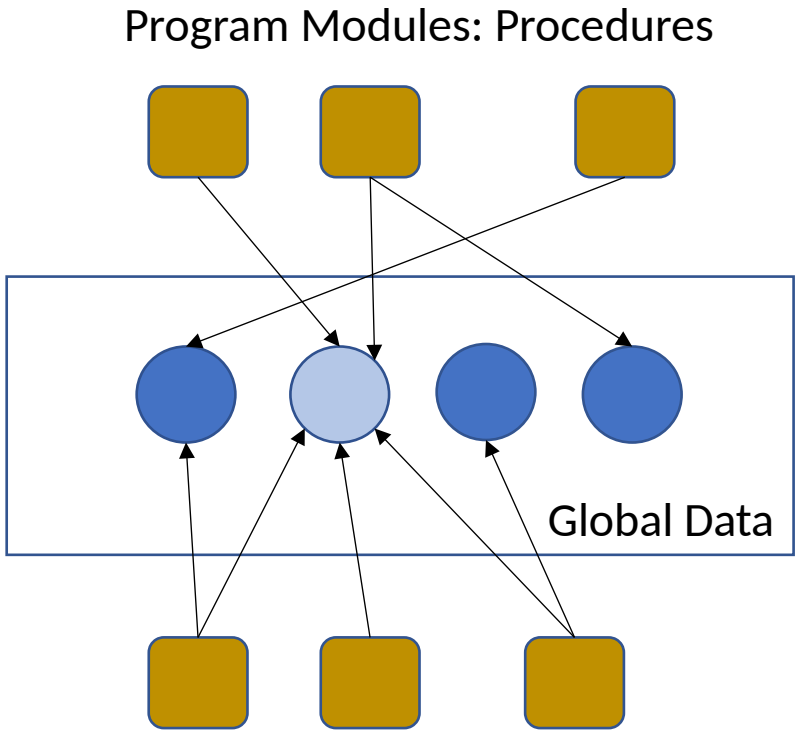
# Encapsulation: implementation hiding



# Encapsulation: implementation hiding

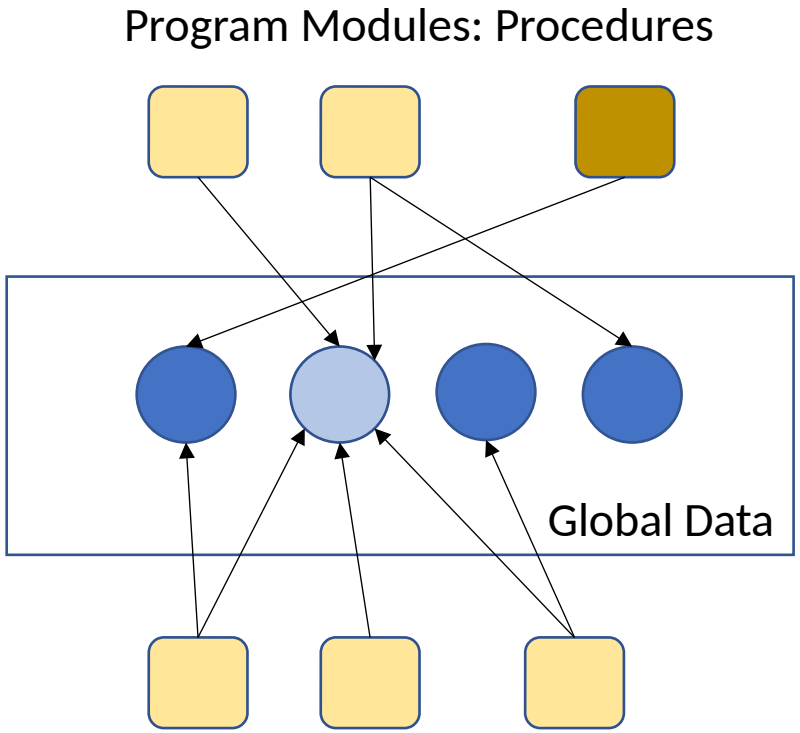


# Encapsulation: implementation hiding



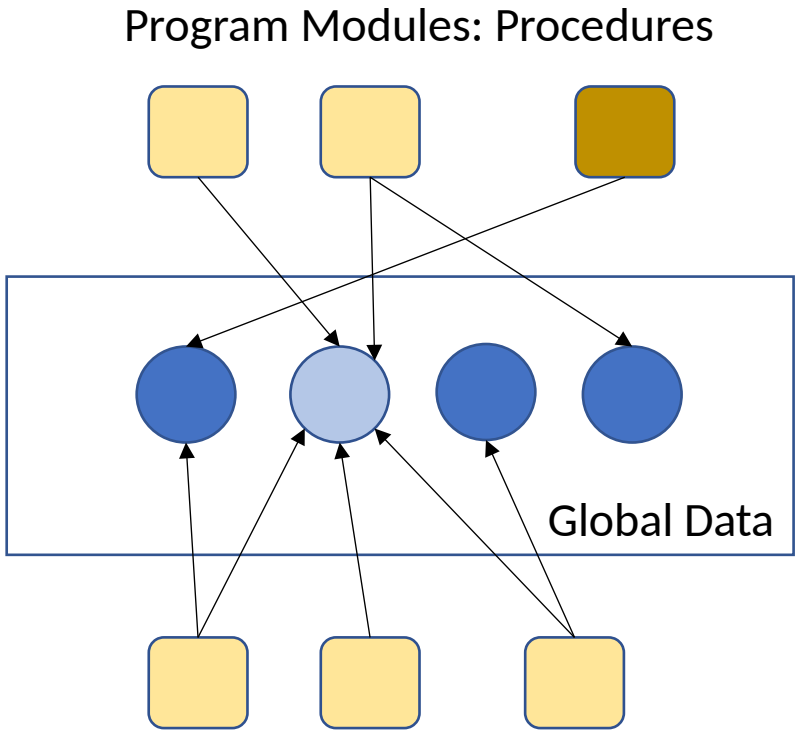

If changes are made to the definition of *address*

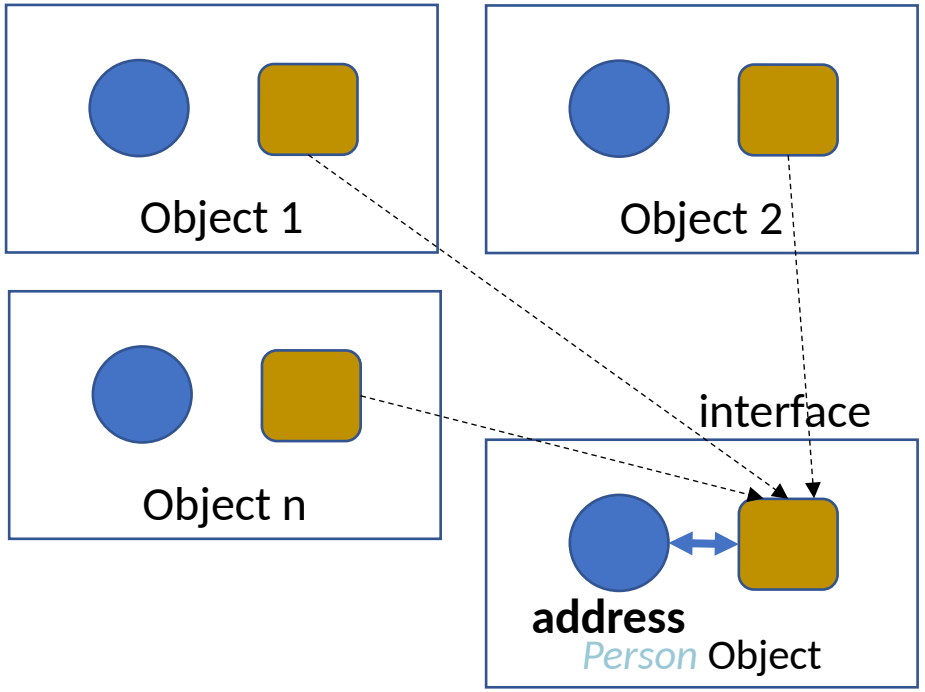
# Encapsulation: implementation hiding




**All** the procedures that use *address* may need to be changed

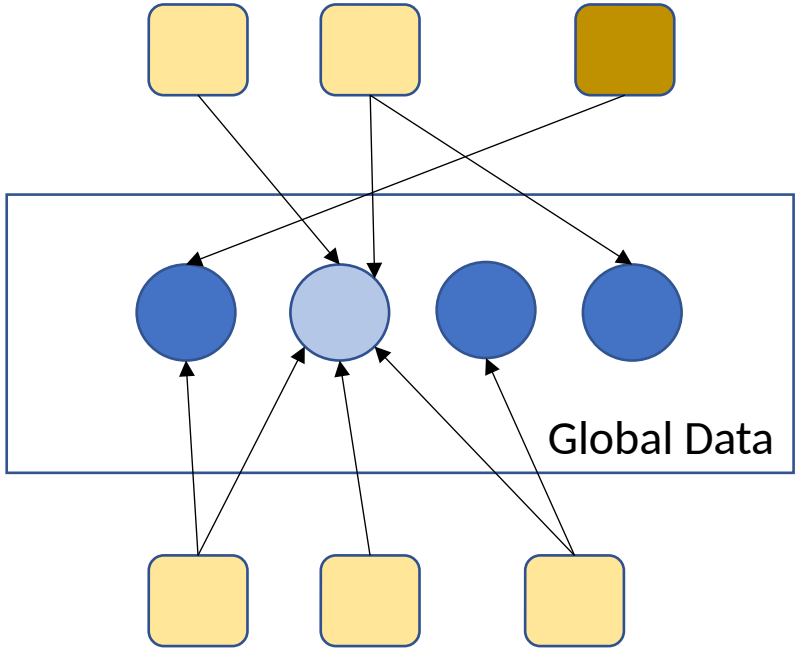
# Encapsulation: implementation hiding

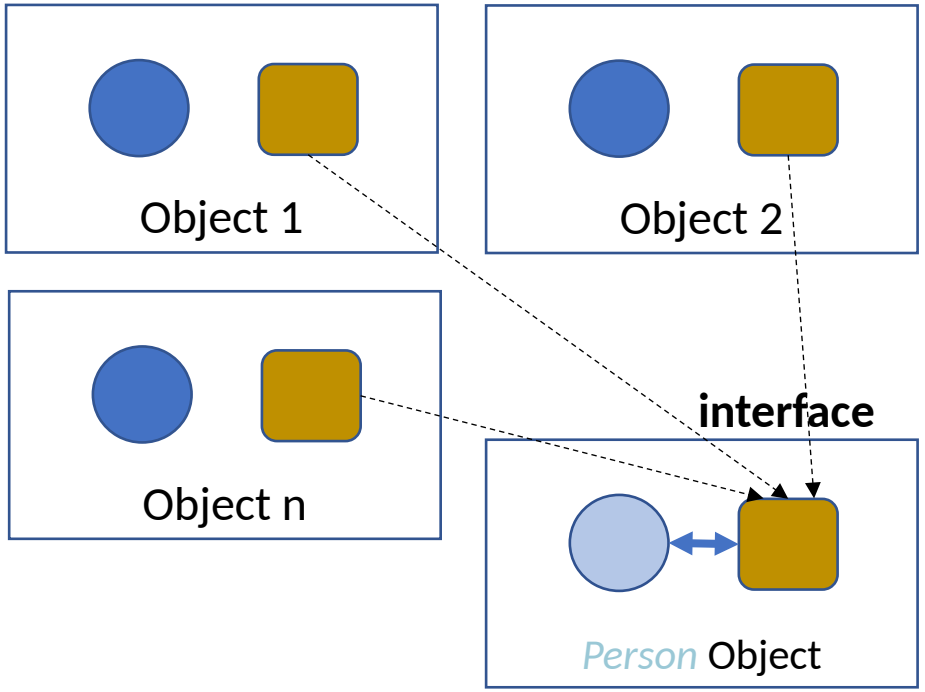


# Encapsulation: implementation hiding

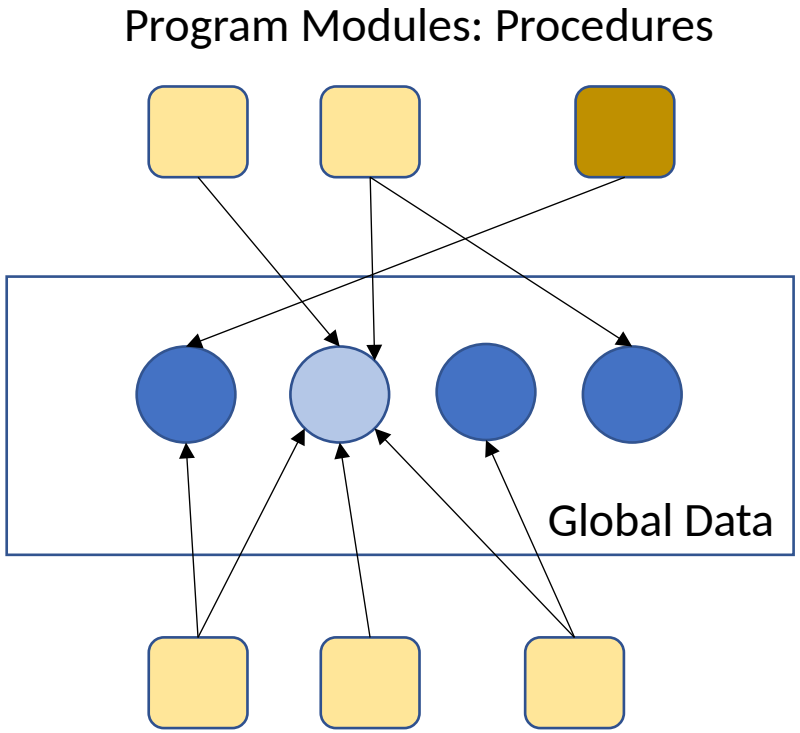
Program Modules: Procedures

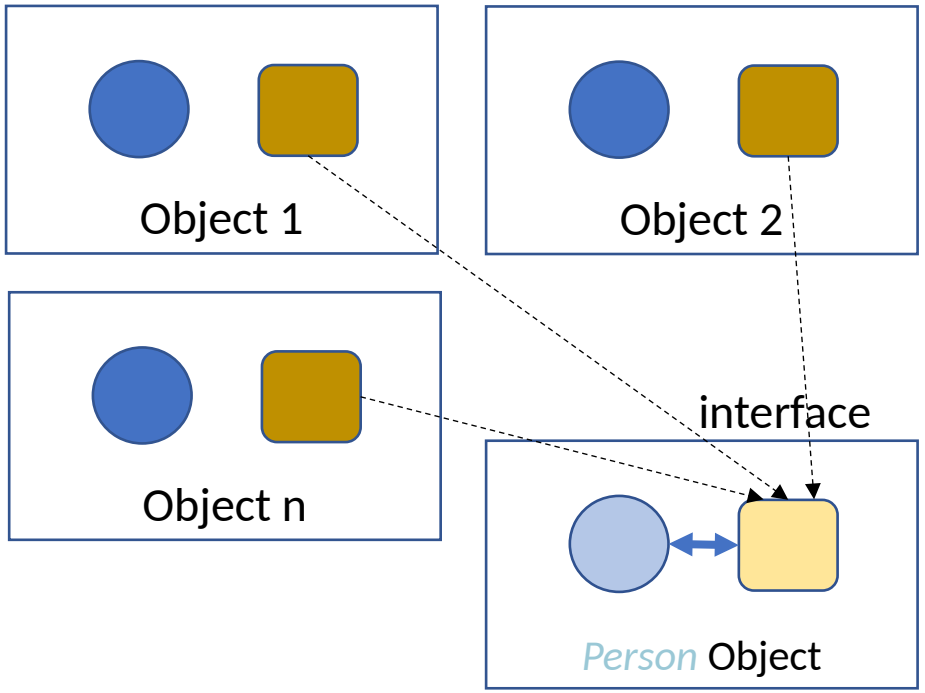



If the definition of the *address* changes but the **interface** does not…

# Encapsulation: implementation hiding




...only the **internal implementation** of the methods of *Person* may need to change

# Encapsulation: 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;
    }
}
```

# Encapsulation: 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;
    }
}
```

Instead of defining *address* as a string, we use a class we define called *Address*.

The class parses the address from a string, checks whether the postcode is correct, etc.

# Encapsulation: 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;
    }
}
```

```
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();
    }
}
```

# Encapsulation: 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;
    }
}
```

same interface

same interface

same interface

```
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();
    }
}
```

# Encapsulation: 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();
    }
}
```

# Encapsulation: implementation hiding

```
class Program
{
   public static void Main()
   {
      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**

# Object-Oriented Programming (OOP) Principles

- Abstraction
- **Encapsulation**
- Inheritance
- Polymorphism

Objects **contain attributes and behaviours**— they can **control** how these are **accessed** and **hide** their implementation from objects of other classes to prevent *errors* and *anomalies* and promote code *maintainability* and *flexibility*.