7SENG011W Object Oriented Programming

More on Methods, Value types and Reference types, UML class Diagrams

Dr Francesco Tusa

Readings

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

- Hands-On Object-Oriented Programming with C#
  - Chapter: Hello OOP—Classes and Objects
- Programming C# 10
  - Chapter 3: Types

Online

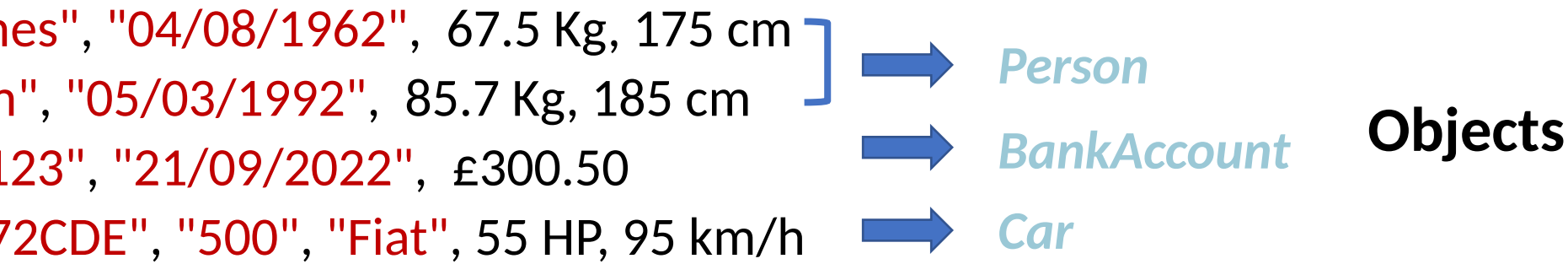
- Value Types
- Reference Types
- Classes
- Methods
- Passing Parameters
- Passing Value Types by Value

Outline

- Summary of last week and more on methods
- Value types and Reference types, Stack and Heap
- UML class diagrams

What are Objects?

- *Different types of data* can be grouped to model **entities** of a **problem** we would like to solve



What are Classes?

- Defines a set of **attributes** (data) and **methods** (behaviours) that are common for some Objects

class Person

attributes: *string name, string dateOb, double weight, int height*

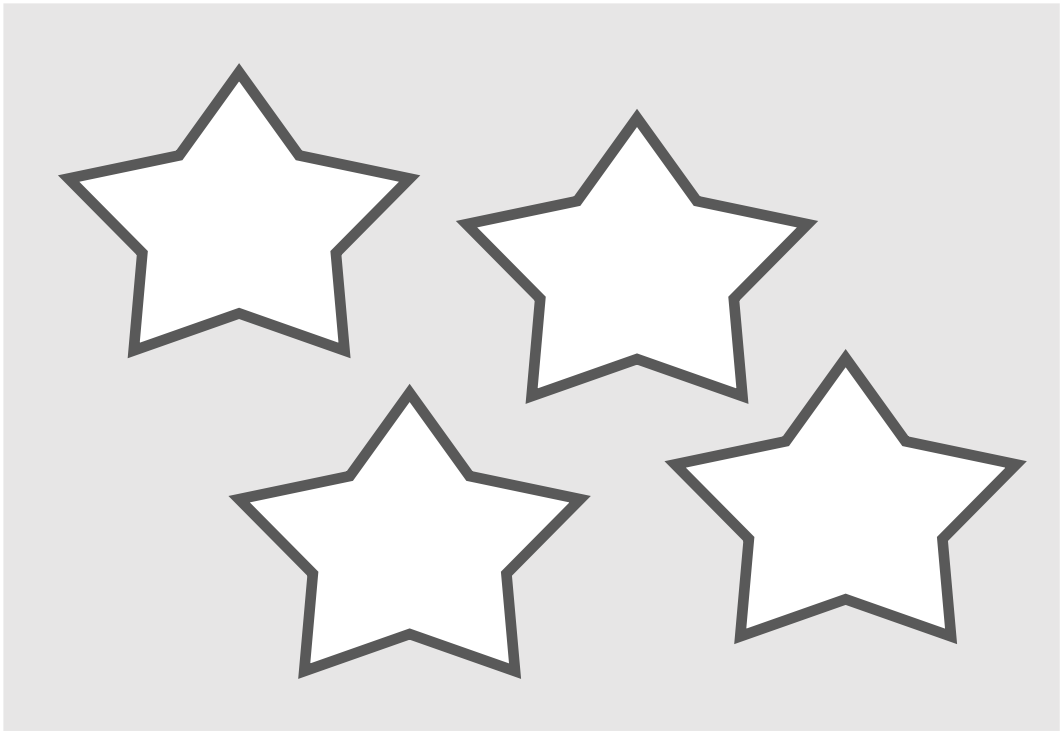
methods: *SayName, Eat, GetAge, …*

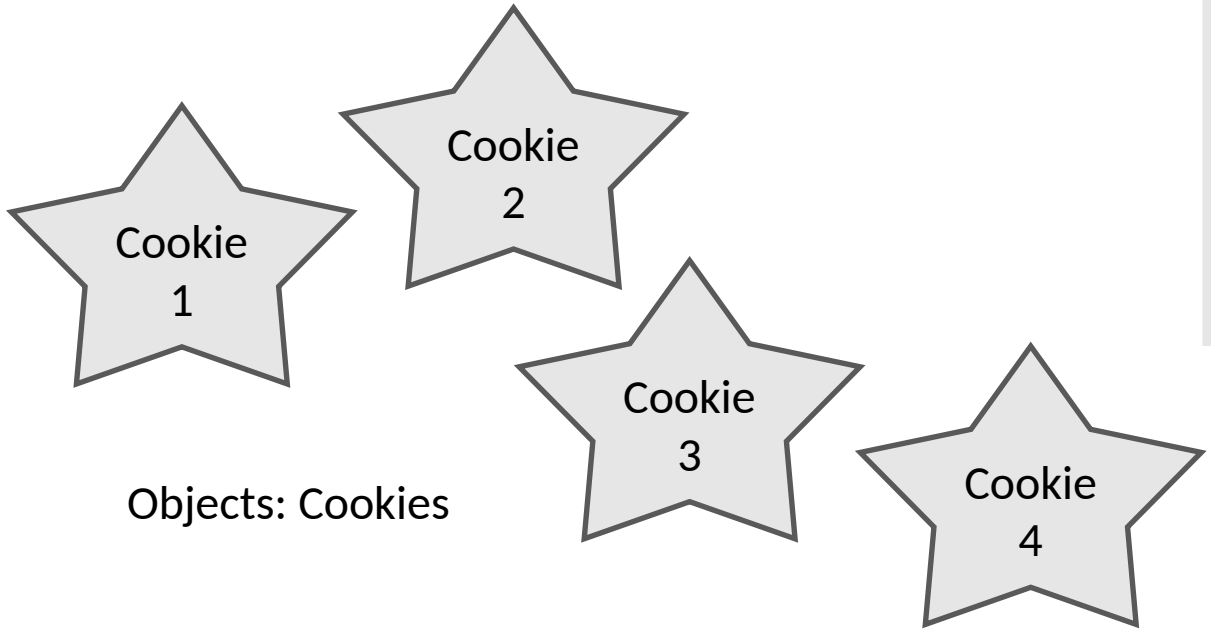
- The **blueprint** is defined **once—multiple Objects** with similar features can be created

Classes and Objects

Class Template

Cookie Dough





Objects: Cookies

Primitive variables vs Objects

```
class Program
{
    static void Main(string[] args)
    {
        int p1X = 6;
        int p1Y = 4;

        int p2X = 8;
        int p2Y = 2;

        Console.WriteLine($"p1 = ({p1X} , {p1Y})");
        Console.WriteLine($"p2 = ({p2X} , {p2Y})");
    }
}
```

versus

```
class Program
{
    static void Main(string[] args)
    {
        Point p1 = new Point(6, 4);
        Point p2 = new Point(8, 2);

        p1.Display();
        p2.Display();
    }
}
```

Custom Type: *Point* class definition

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display()
        {
            Console.WriteLine($"({x}, {y})");
        }
}
```

```
class Program
{
        static void Main(string[] args)
        {
                Point p1 = new Point(6, 4);
                Point p2 = new Point(8, 2);

                p1.Display();
                p2.Display();
        }
}
```

Custom Type: *Point* class definition

```
class Point
{
    private int x;
    private int y;

    public Point(int xarg, int yarg )
    {
        x = xarg;
        y = yarg;
    }

    public void Display()
    {
        Console.WriteLine($"({x}, {y})");
    }
}
```

Name of the class—custom type

Custom Type: *Point* class definition

```
class Point
{
    private int x;
    private int y;

    public Point(int xarg, int yarg )
    {
        x = xarg;
        y = yarg;
    }

    public void Display()
    {
        Console.WriteLine($"({x}, {y})");
    }
}
```

Attributes: variables that define the data of each object instance—different Objects have different copies of the attributes

Custom Type: *Point* class definition

```
class Point
{
    private int x;
    private int y;

    public Point(int xarg, int yarg )
    {
        x = xarg;
        y = yarg;
    }

    public void Display()
    {
        Console.WriteLine($"({x}, {y})");
    }
}
```

Methods: define the operations that an object instance can perform (on the attributes)

Custom Type: *Point* class definition

```
class Point
{
    private int x;
    private int y;

    public Point(int xarg, int yarg )
    {
        x = xarg;
        y = yarg;
    }

    public void Display()
    {
        Console.WriteLine($"({x}, {y})");
    }
}
```

Constructor: a method that initialises the attributes with some values when a new object instance is created—has the same name of the class

Method

- Allows the execution of **operations**—actions objects of a class can perform (on the attributes)
- *0*, *1*, or *n* **parameters** can be used *inside the body* of the method

Method (type1 par1, type2, par2, …)

Examples:

Display() { … } // 0 parameters

Point(int xarg, int yarg) { … } // 2 parameters

Blocks: from Lecture 1

- A block is a region of code delimited by a pair of curly braces *{}*
- Program *Main* entry-point
- Instructions within an *if*, *else* or *switch* statement
- Instructions within a *for*, *while*, or *do while* loop
- Blocks can be *nested*

- **Methods** are also blocks

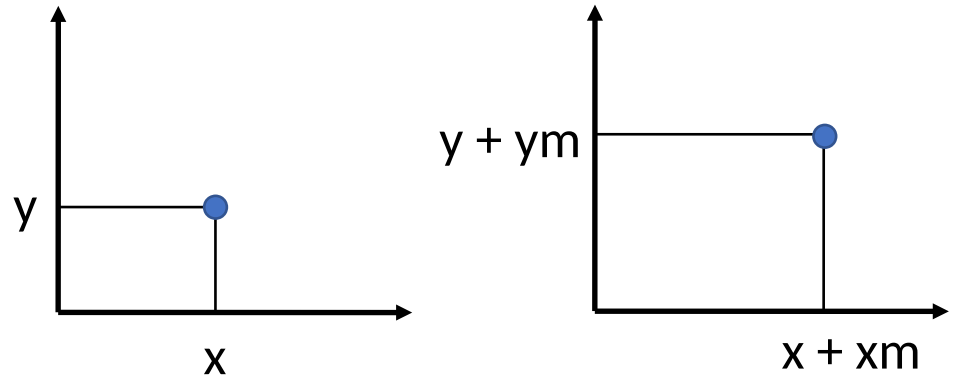
Methods: variables scope

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display()
        {
            Console.WriteLine($"({x}, {y})");
        }
}
```


The code inside the methods **can access** the class attributes *x* and *y*

Methods: variables scope

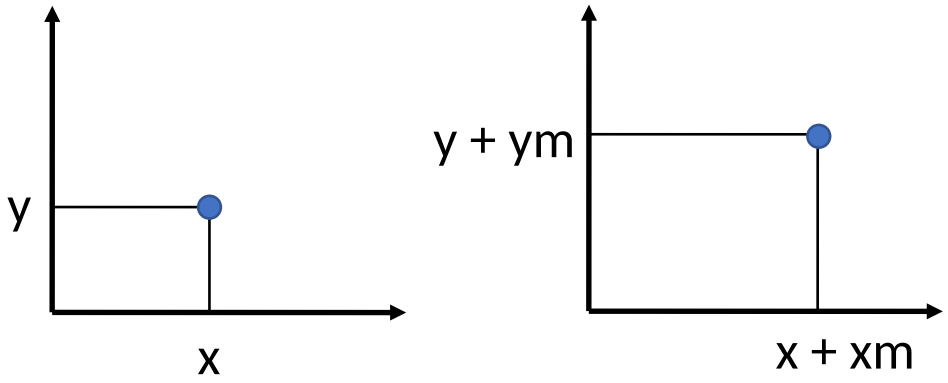
```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
         }
}
```



New method—*Move*—shifts the point of *xm* and *ym* on the x and y axis

Methods: variables scope

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
         }
}
```



New method—*Move*—shifts the point of *xm* and *ym* on the x and y axis

Methods: variables scope

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
         }
}
```

The code inside each method **can access** the class attributes *x* and *y*

Methods: variables scope

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
         }
}
```

Each method can access its parameters—they are local variables not available anywhere else in the class

Methods: variables scope

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
            int sum = xm + ym;
         }
}
```

The same applies to other *variables* declared inside a method

Methods: Parameters and Arguments

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
            int sum = xm + ym;
         }
}
```

```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);

        int xt = 1, yt = 1;
        p1.Move(xt, yt);
    }
}
```

When a method is called, the instructions defined inside that method are executed

Methods: Parameters and Arguments

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
            int sum = xm + ym;
         }
}
```

accepted input **parameters**

(6, 4)

```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);

        int xt = 1, yt = 1;
        p1.Move(xt, yt);
    }
}
```

actual values passed, **arguments**

The values of the **arguments** (a, b) are copied into the **parameters** (xarg, yarg)

Methods: Parameters and Arguments

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
            int sum = xm + ym;
         }
}
```

```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);

        int xt = 1, yt = 1;
        p1.Move(xt, yt);
    }
}
```

What happens to the *x* and *y* attributes of *p1* when the method *Move* is called?

Methods: invocation within the same class

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { ... }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
            int sum = xm + ym;
            Display();
         }
}
```

A method can call any other methods defined in the same class using the method name

Methods: invocation within the same class

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
            int sum = xm + ym;
            Display();
         }
}
```

```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);

        int xt = 1, yt = 1;
        p1.Move(xt, yt);
    }
}
```

p1.Move will call *Display()* on the same object (*p1*)

Methods: void or return

When called, a **method** can:

- Perform some operations and ***terminate*—void**
- Perform some operations and **return** a ***value*** to the caller

Methods: void

When called, a **method** can:

- Perform some operations and ***terminate*—void**
- Perform some operations and **return** a ***value*** to the caller

```
public void methodName(type arg1, type arg2, ...)
{
        // statements
}
```

Methods: void

When called, a **method** can:

- Perform some operations and ***terminate*—void**
- Perform some operations and **return** a ***value*** to the caller

```
class Point
{
      // x and y attributes declaration

      public void Display()
      {
         // prints x and y of the Point
      }
}
```

Methods: void

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display()
        {
            Console.WriteLine($"({x}, {y})");
        }
}
```

```
class Program
{
        static void Main(string[] args)
        {
                Point p1 = new Point(6, 4);
                Point p2 = new Point(8, 2);

                p1.Display();
                p2.Display();
        }
}
```

When *Display*() is invoked, the *Main* does not receive anything back from the execution—no **values** are produced

Methods: return

When called, a **method** can:

- Perform some operations and *terminate*—**void**
- Perform some operations and **return** a ***value*** to the caller

```
public TYPE methodName(type arg1, type arg2, ...)
{
        // statements
        return EXPRESSION
}
```

Methods: return

When called, a **method** can:

- Perform some operations and *terminate*—**void**
- Perform some operations and **return** a ***value*** to the caller

```
//In the Circle class

public double Area()
{
        double area = Math.PI * radius * radius;
        return area;
}
```

Methods: return – Random class example

```
Random r = new Random();

int randomInt = r.Next(1, 7);
```

- The method *Next* of the *Random* class returns a random int value in the range specified via the provided arguments

```
double randomDouble = r.NextDouble();
```

- The method *NextDouble* of the *Random* class returns a random *double* value in the range 0-1 (no arguments can be provided)

Methods: *Point* class example

Let's add two new methods to our *Point* class

- *GetX*(): when called, returns the content of the object's *x* attribute
- *GetY*(): when called, returns the content of the object's *y* attribute

Methods: *Point* class example

Let's add two new methods to our *Point* class

- *GetX*(): when called, returns the content of the object's *x* attribute
- *GetY*(): when called, returns the content of the object's y attribute

In OOP, these methods are referred to as *Getter Methods*

Similarly, *Setter Methods* are used to set the value of an attribute

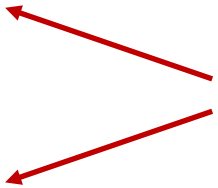
Methods: Point class example

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg) { ... }

        public void Display() { ... }
        public void Move(int xm, int ym) { ... }

        public int GetX()
        {
            return x;
        }

        public int GetY()
        {
            return y;
        }
}
```


The methods *Display*() and *Move*() perform some operations and terminate—**void**

Methods: Point class example

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg) { ... }

        public void Display() { ... }
        public void Move(int xm, int ym) { ... }

        public int GetX()
        {
            return x;
        }

        public int GetY()
        {
            return y;
        }
}
```



The methods *GetX*() and *GetY*() *return* the content of the attributes *x* and *y* as **int** values

Methods: Point class example

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg) { ... }

        public void Display() { ... }
        public void Move(int xm, int ym) { ... }

        public int GetX()
        {
            return x;
        }

        public int GetY()
        {
            return y;
        }
}
```

```
class Program
{
        public static void Main(string[] args)
        {
            Point p1 = new Point(2, 3);
                int xt =2, yt = 1;
            p1.Move(xt, yt);
            int x1 = p1.GetX();

            Point p2 = new Point(6, 2);
            int x2 = p2.GetX();

            if (x2 > x1)
            {
                Console.WriteLine("x of p2 greater than x of p1");
            }
        }
}
```

Methods: Constructors

- The *constructor*'s name must be **identical** to the class *Name*
- *Constructors* have **no return type**—implicitly void

```
class Point
{
        public (void) Point(int xarg, int yarg) { … }
        …
}
```

- All classes need **at least one** constructor
- If you do not write one, then ***Name*** **() { }** is created by default

Outline

- Summary of last week and more on methods
- Value types and Reference types, Stack and Heap
- UML class diagrams

Question

On C# memory management

Answer on PollEveryWhere

https://pollev.com/francescotusa

Value and Reference Types

- **Primitive** types are basic C# types:
  - int, double, bool, char, etc.

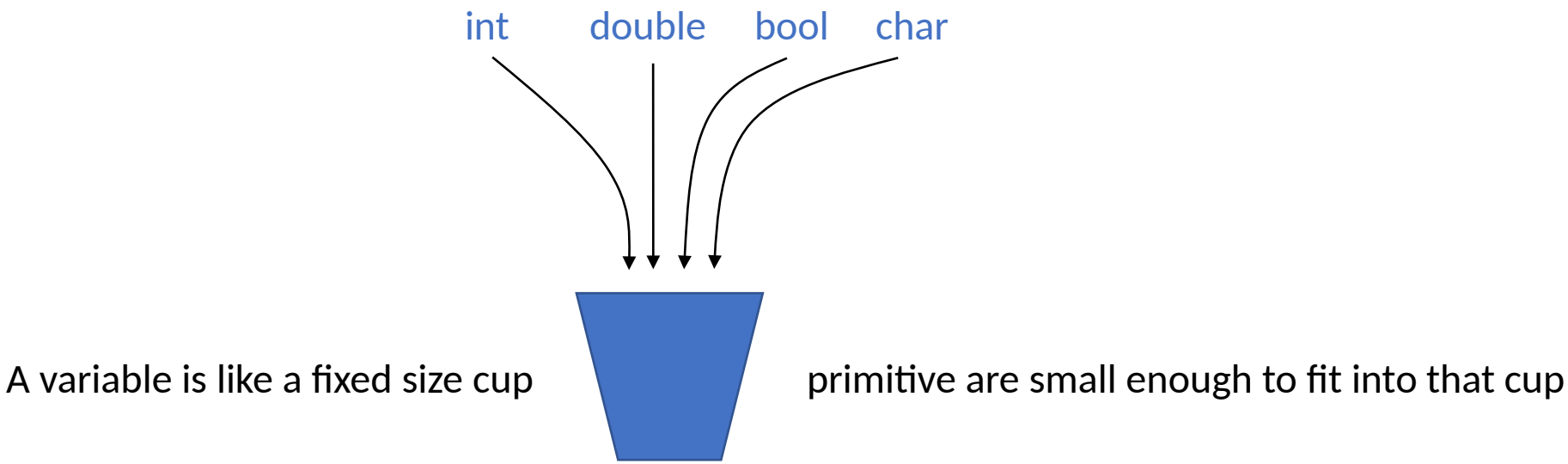
- **Reference** types are *arrays* and *objects*:
  - string, int[], double[], *Point*, *Circle*, *BankAccount*, etc.

Value and Reference Types

- **Primitive** types are basic C# types:
  - int, double, bool, char, etc.
  - the actual data **values are stored inside** the variable: **value type**
- **Reference** types are *arrays* and *objects*:
  - string, int[], double[], *Point*, *Circle*, *BankAccount*, etc.
  - the actual data values **are not stored inside** the variable
  - the variable contains a **reference** to the data (object's address)

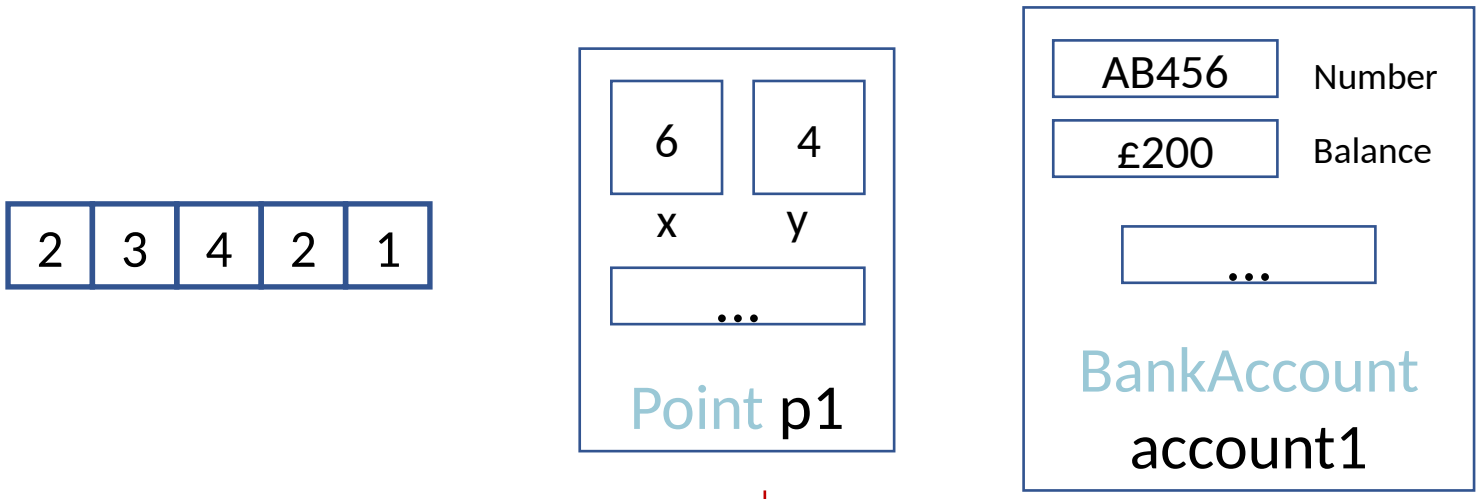
How primitive types are stored

- **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)



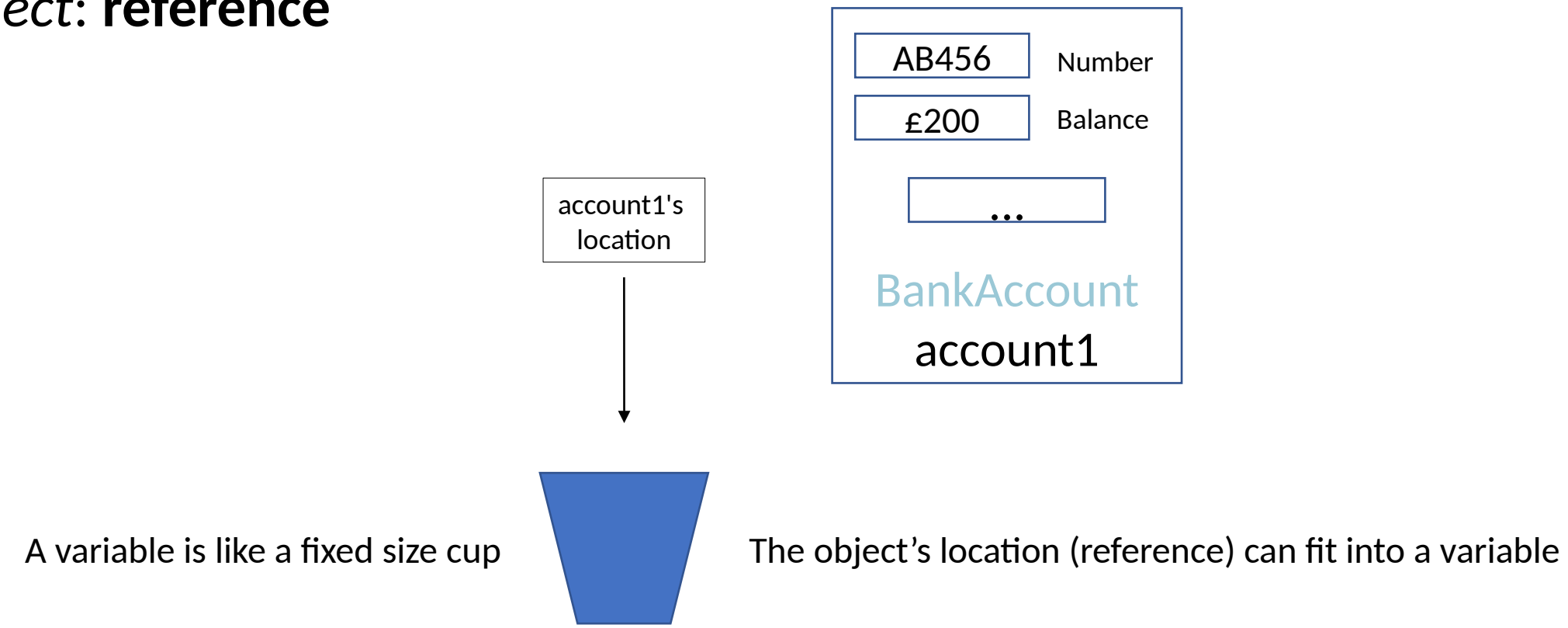
How reference types are stored

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



How reference types are stored

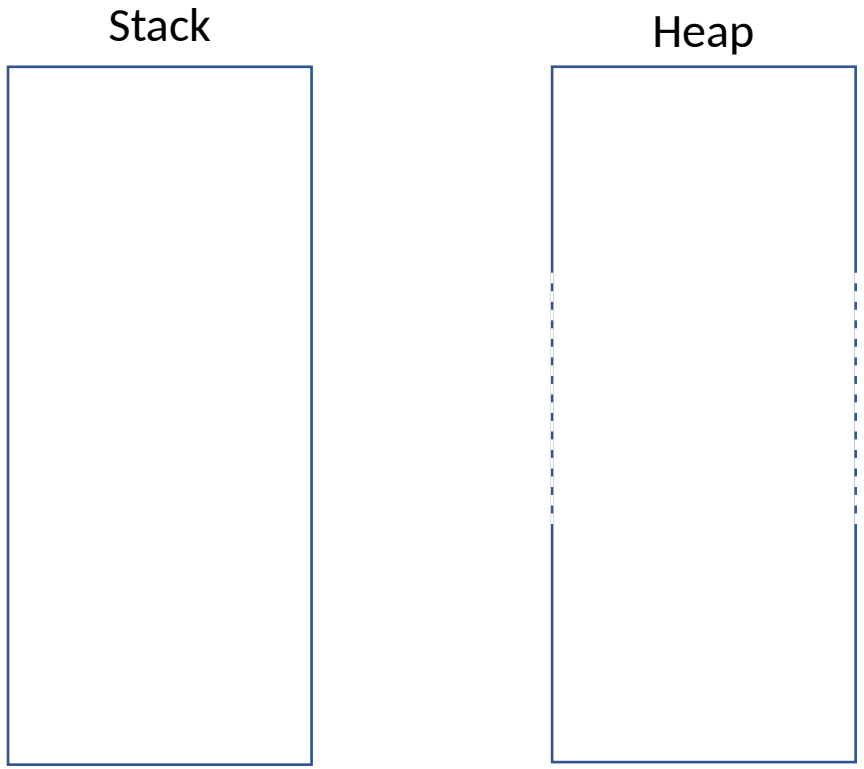
- The data (object) **is not** stored inside the **variable**
- The **variable** stores a number (address) that locates that *object*: **reference**



Memory stack and heap

- When a program is executed, it is given an initial amount of **memory** by the *operating system*
- A part of this memory is the ***program stack***
- Another part of this memory is the ***program heap***

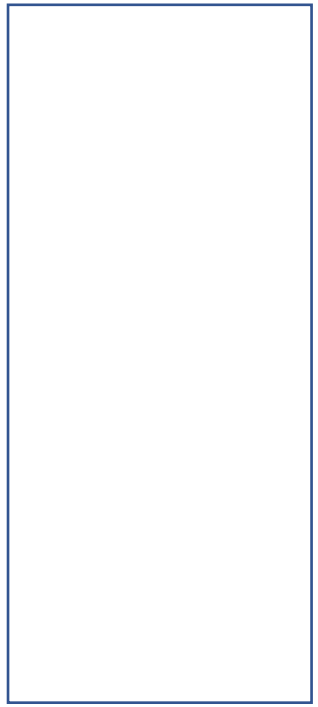
Memory stack and heap



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

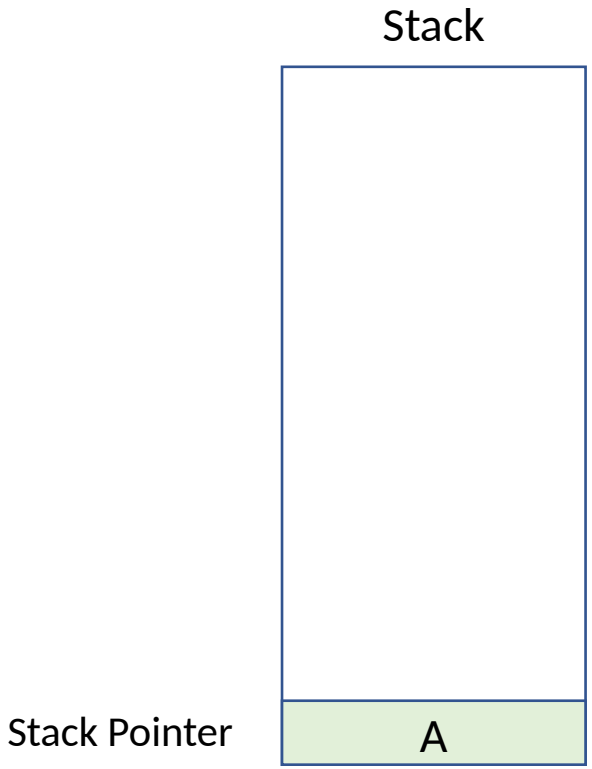
Memory **stack** and heap

Stack



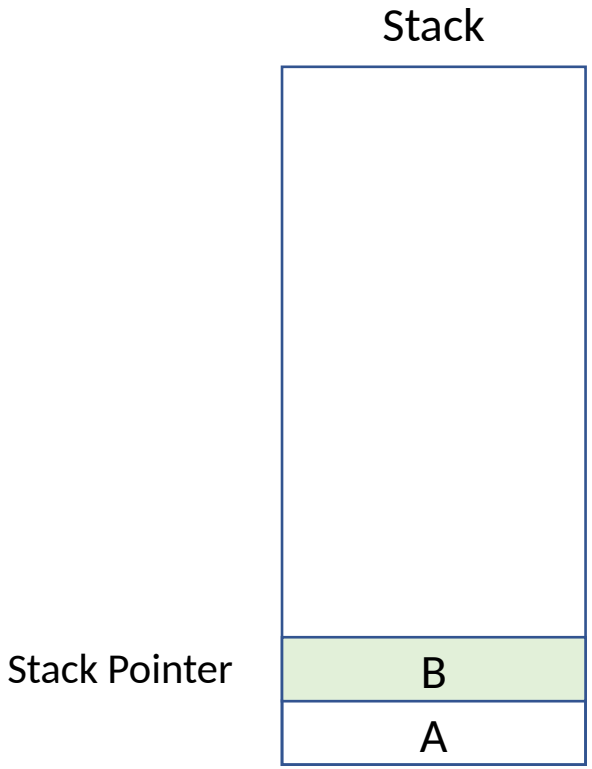
Adding element: **Push** A

Memory **stack** and heap



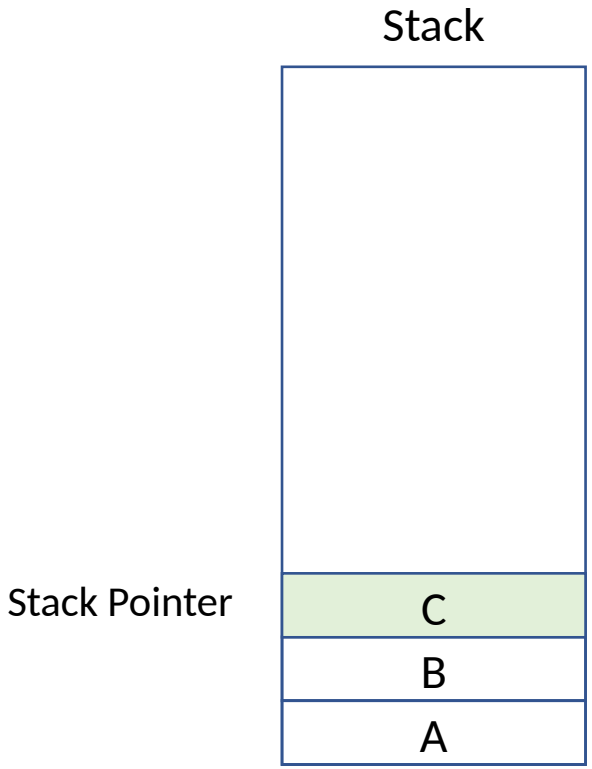
Added element: A

Memory **stack** and heap



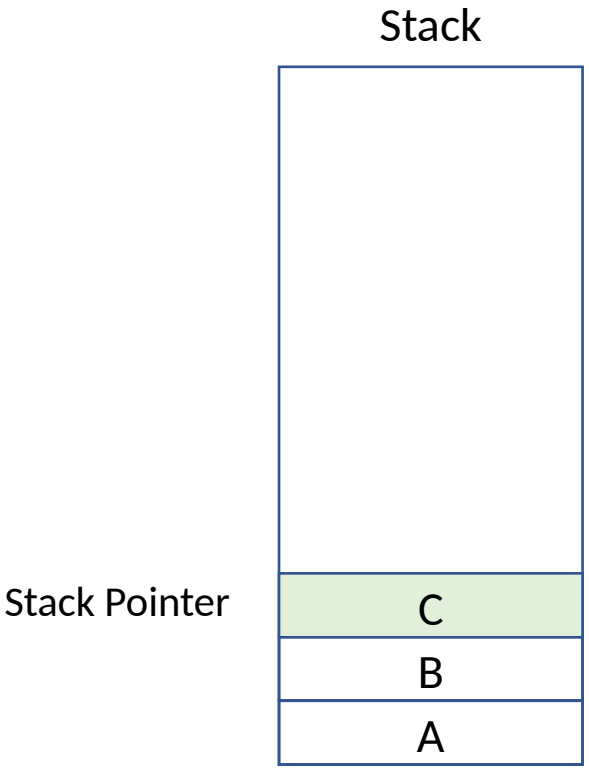
Added element: B

Memory **stack** and heap



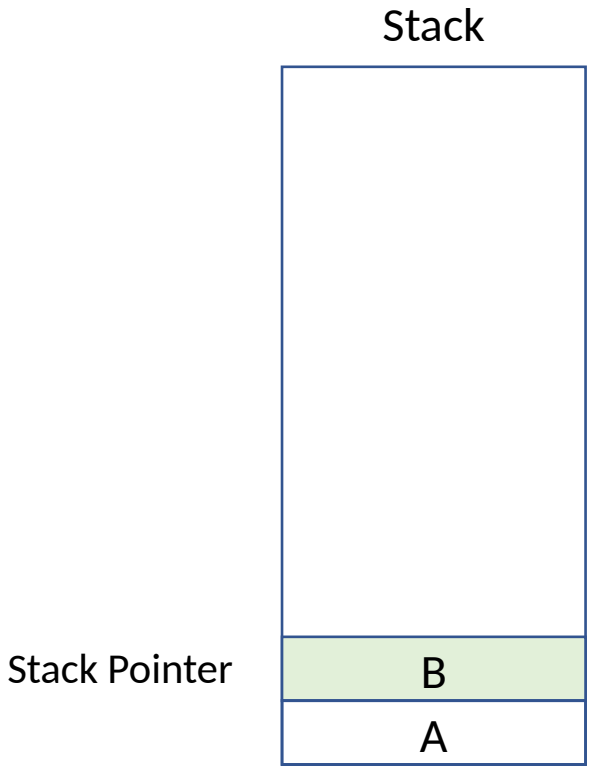
Added element: C

Memory **stack** and heap



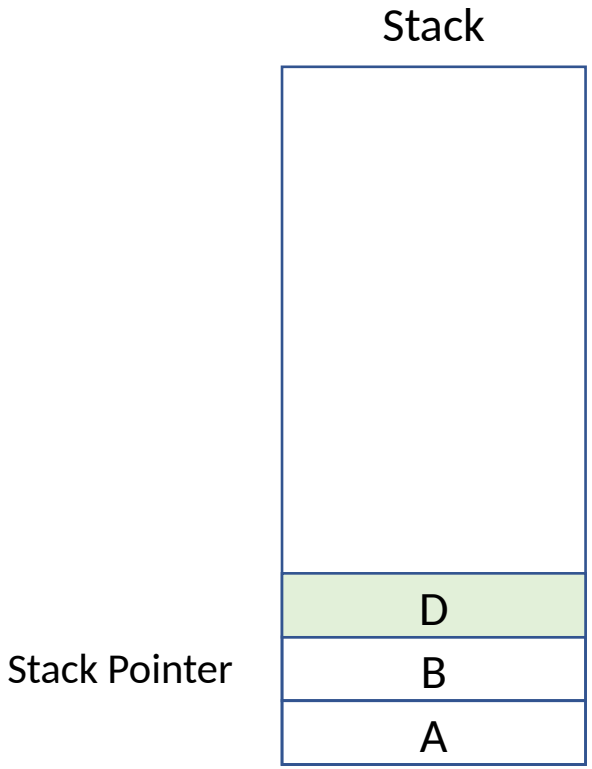
Removing element: **Pop** C

Memory **stack** and heap



Removed element: C

Memory **stack** and heap

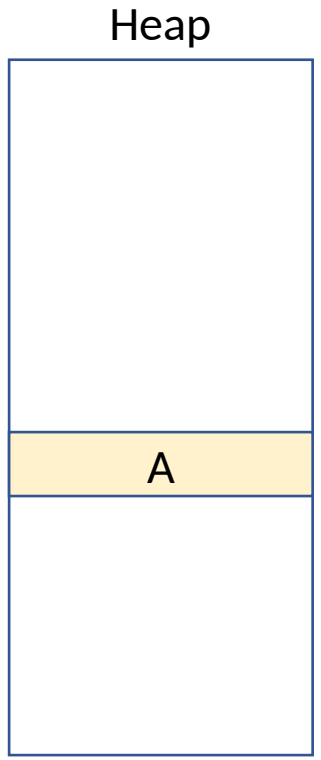


Added element: D

Stack memory allocation pattern: LIFO

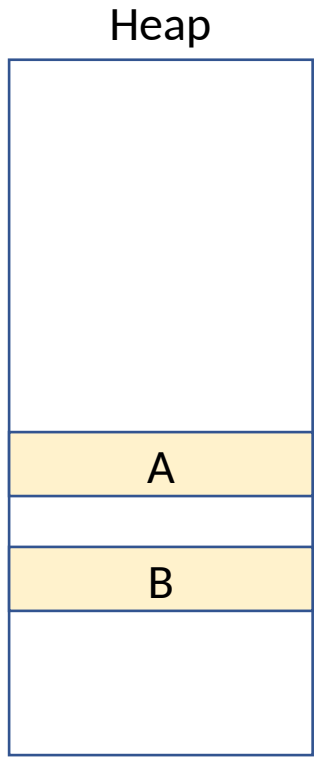
Memory stack and **heap**

Added element: A



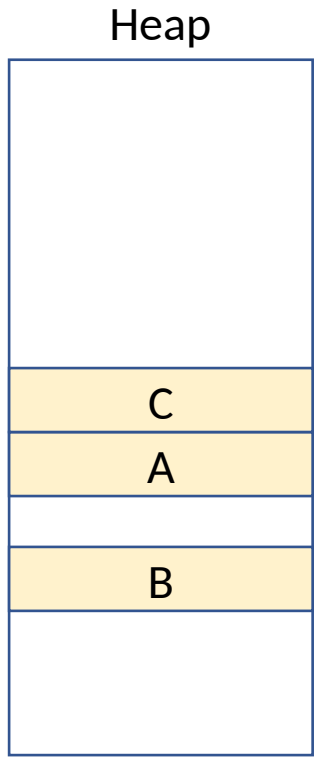
Memory stack and **heap**

Added element: B



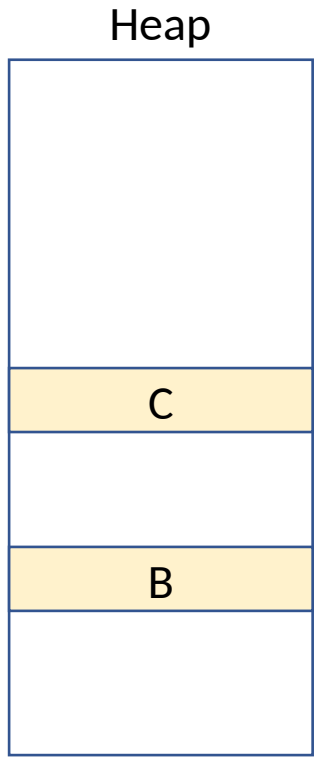
Memory stack and **heap**

Added element: C



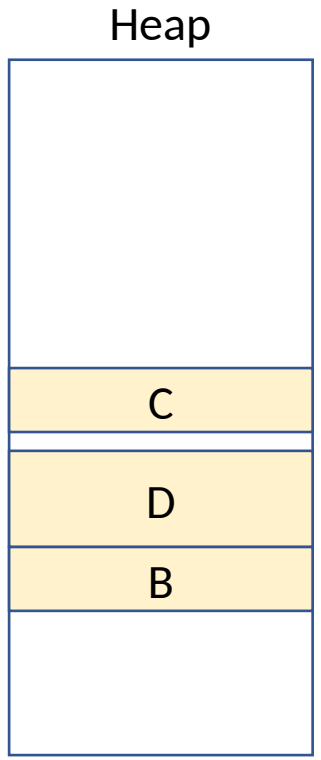
Memory stack and **heap**

Removed element: A



Memory stack and **heap**

Added element: D



Heap memory allocation: pattern-free, complex and may lead to fragmentation

Method Invocation

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
            int sum = xm + ym;
         }
}
```

```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);

        int xt = 1, yt = 1;
        p1.Move(xt, yt);
    }
}
```

What happens inside the memory when *Move* is called from the *Main*?

Method Invocation

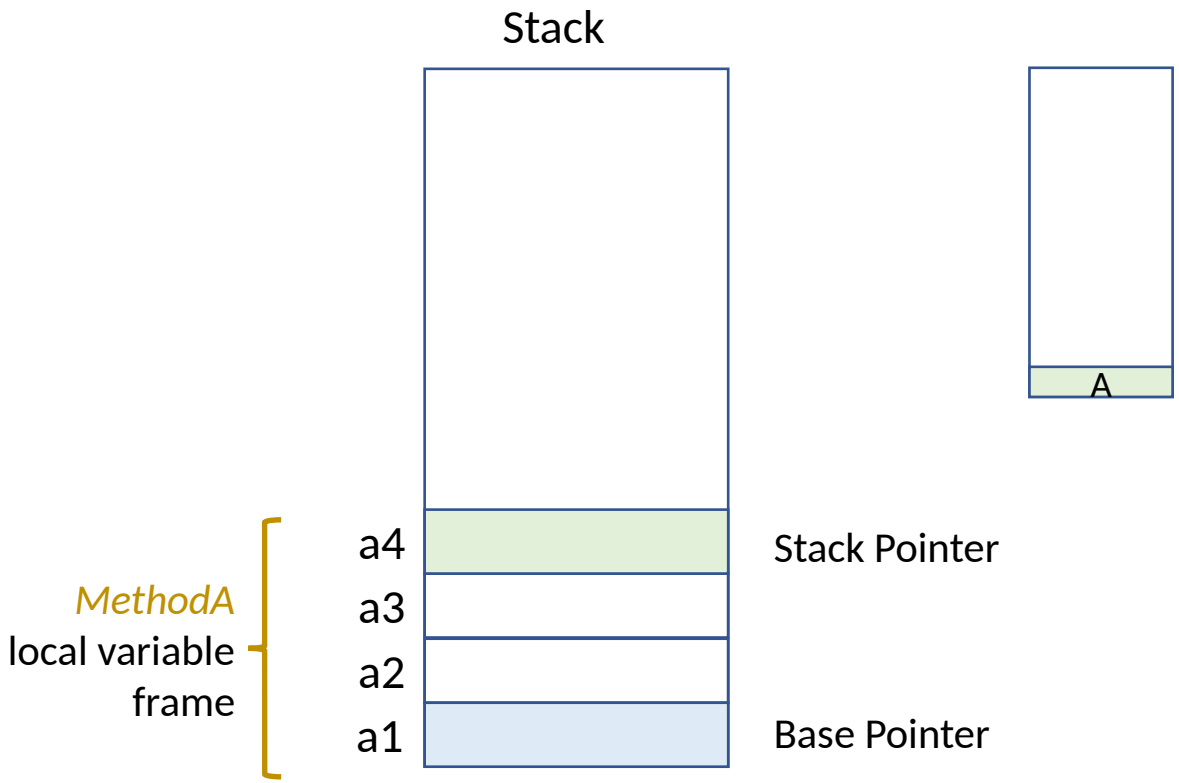
```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg )
        {
            x = xarg;
            y = yarg;
        }

        public void Display() { … }

        public void Move(int xm, int ym)
        {
            x = x + xm;
            y = y + ym;
            int sum = xm + ym;
         }
}
```

```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);

        int xt = 1, yt = 1;
        p1.Move(xt, yt);
    }
}
```

The *Move* method's *parameters* and *local variables* need to be allocated

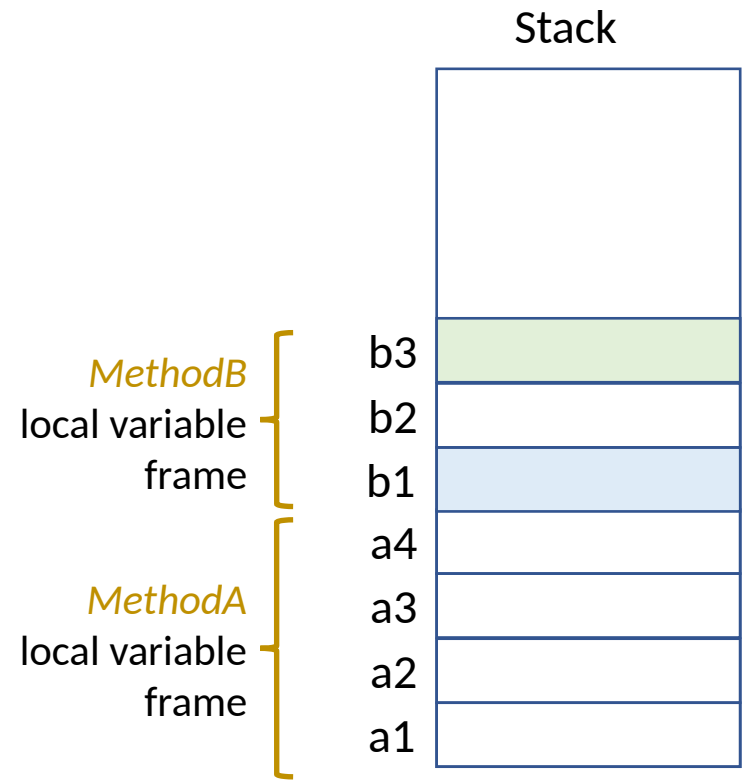
Method Invocation

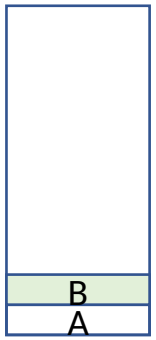
```
void MethodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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



Method Invocation

```
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);
}
```

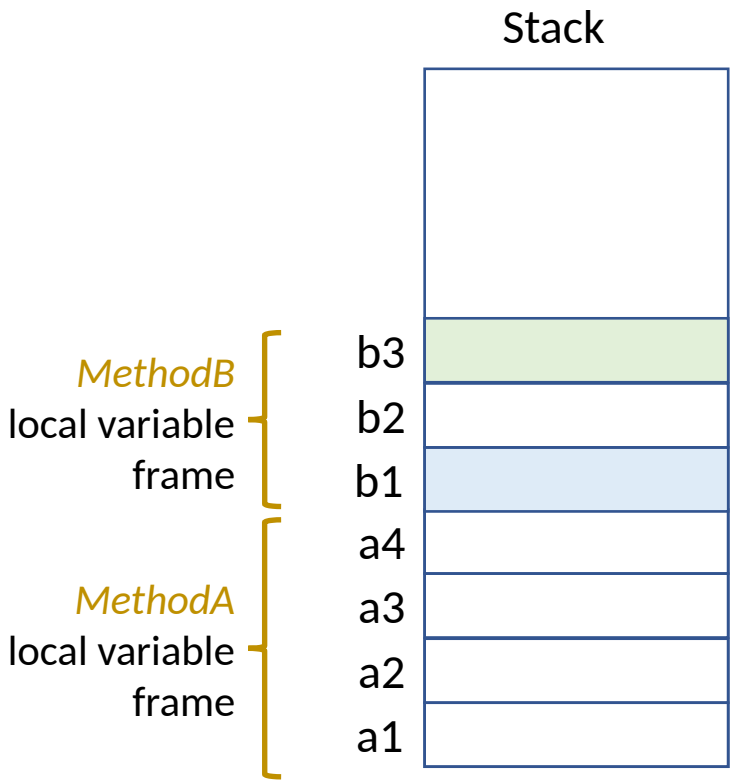




Method Invocation

If *MethodA* is called using argument values 3.0 and 4, what the content of *b2* will be in *MethodB*?

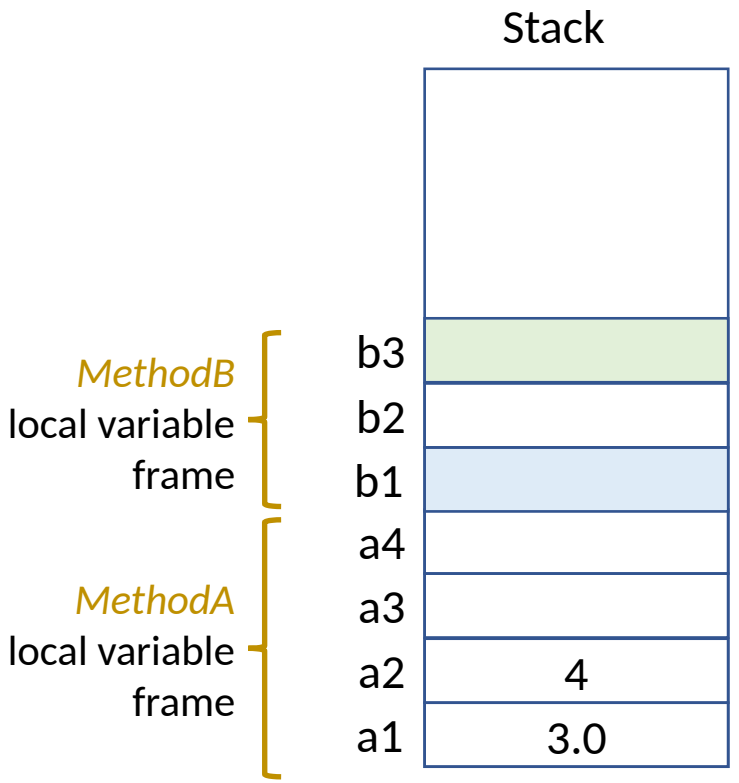
```
void MethodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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



Method Invocation

If *MethodA* is called using argument values 3.0 and 4, what the content of *b2* will be in *MethodB*?

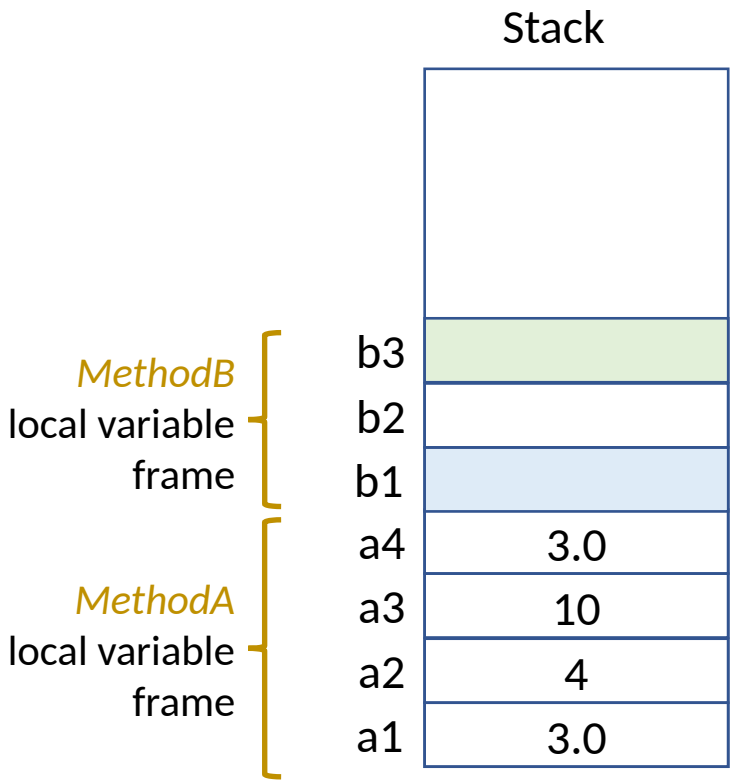
```
void MethodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

                                3.0      4
void MethodA(double a1, int a2)
{
   int a3 = 10;
   double a4 = a1;
   MethodB(a4);
}
```



Method Invocation

If *MethodA* is called using argument values 3.0 and 4, what the content of *b2* will be in *MethodB*?

```
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);
}
```

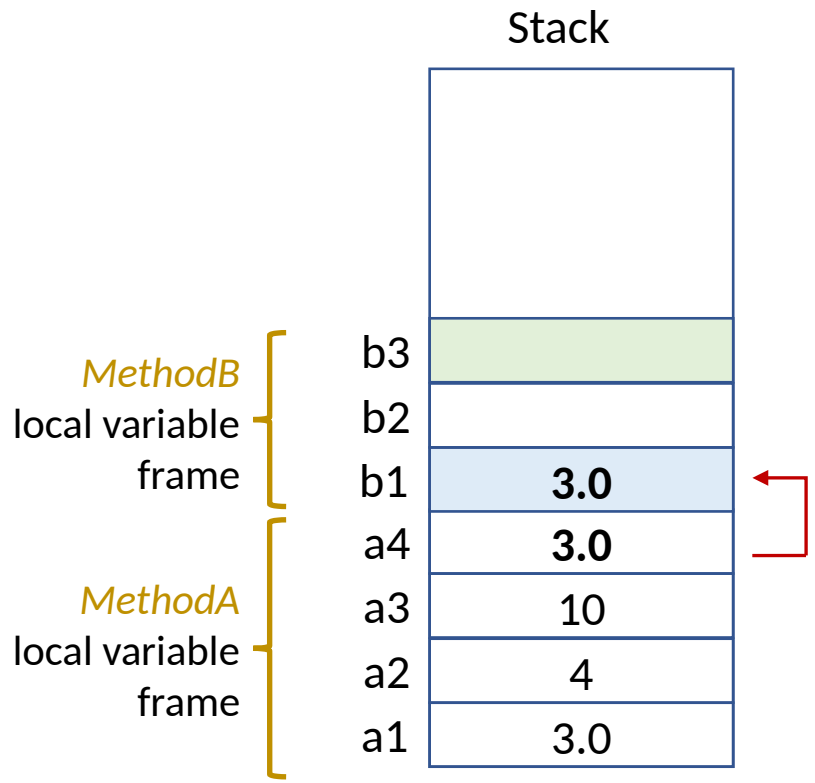


Method Invocation

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*

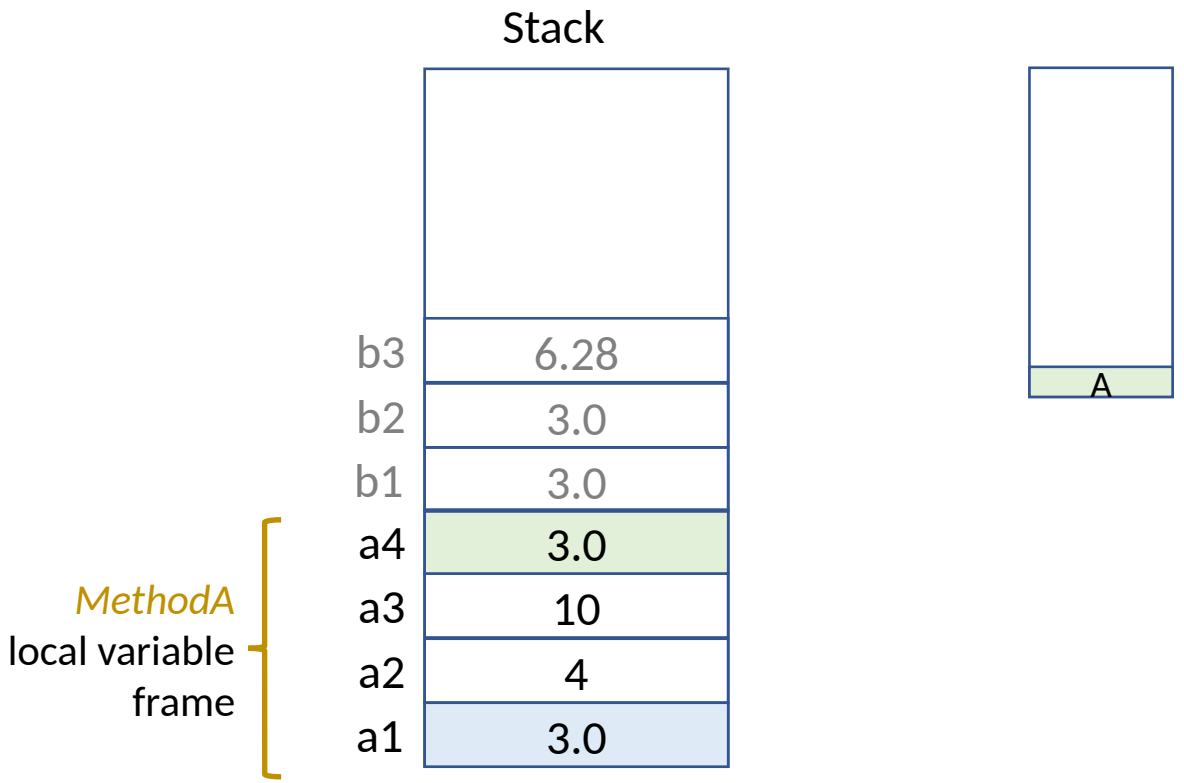
```
void MethodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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



Method Invocation

When *MethodB* terminates its local variables are no longer needed

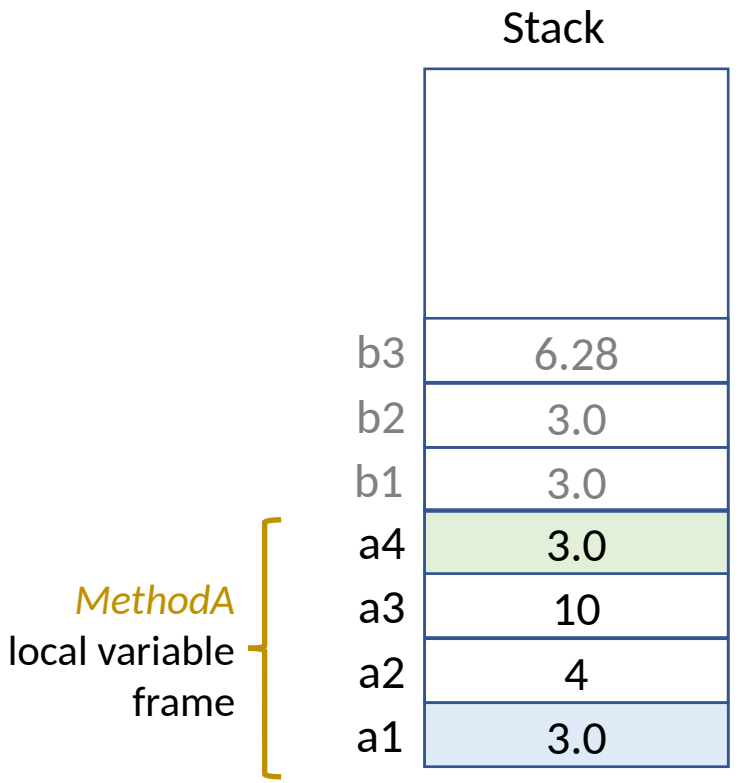
```
void MethodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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



Method Invocation

What happens if another method, e.g., *MethodC* is called inside *MethodA*?

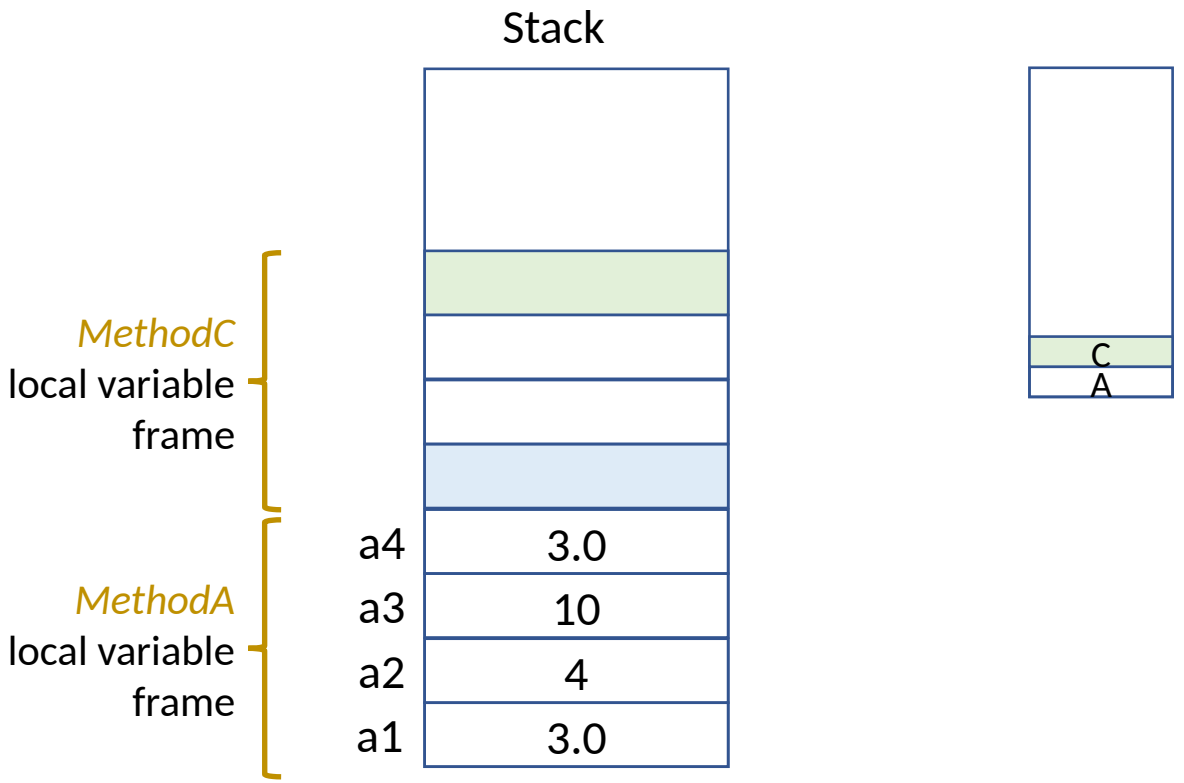
```
void MethodC( ... )
{
   ...

}

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



Method Invocation

MethodC's local variable frame is created at the top of the stack and will overwrite *MethodB* one

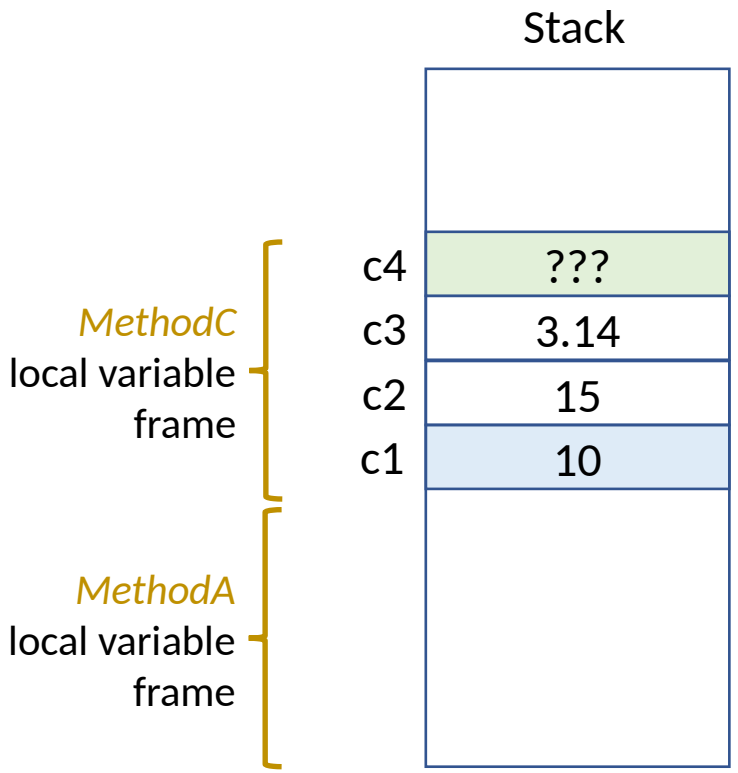
```
void MethodC( … )
{
   …

}

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



Objects allocation

What happens when a reference-type variable, e.g., an object of *ClassX*, is created inside a method?

```
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);
}
```



Memory stack and heap

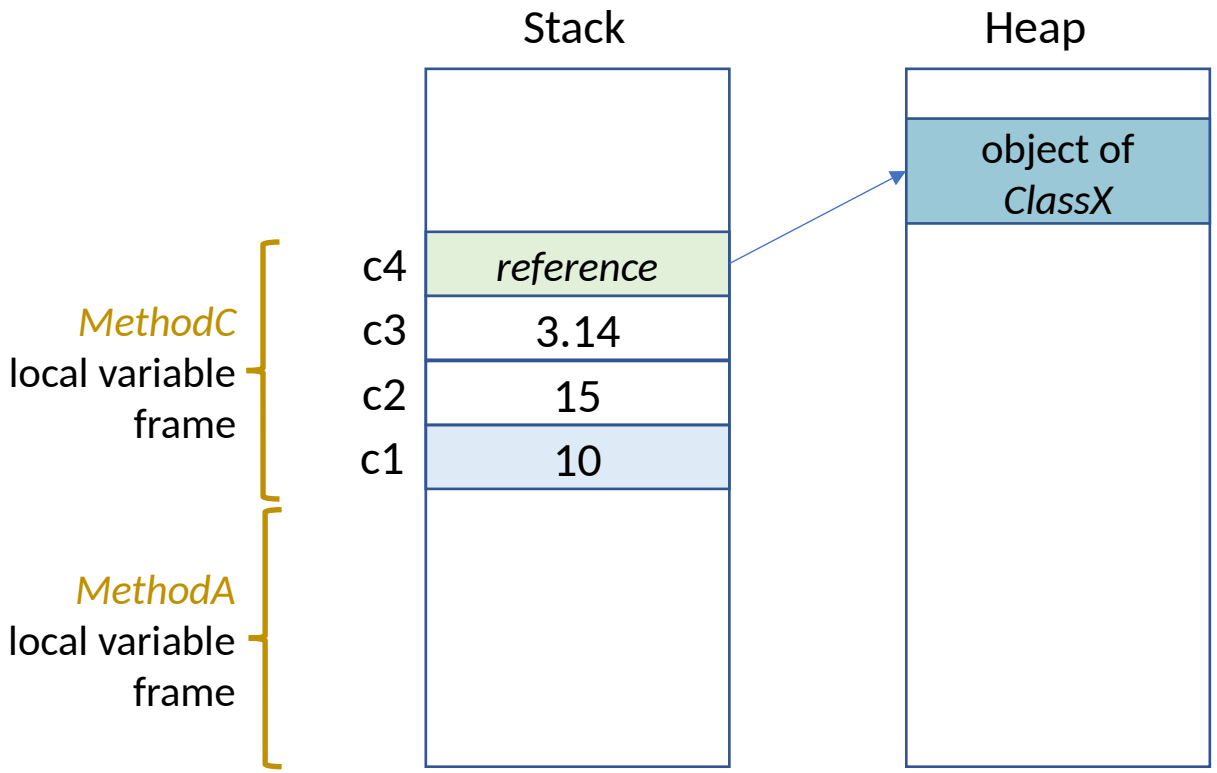
- *Primitive types* (**value types**) methods' *parameters* and local variables declared inside methods are allocated on the ***stack***

- An *object* or *array* (**reference types**) created via the new operator
  - Is allocated on the ***heap*** – also its attributes of *primitive types*
  - The **reference** variable for that object is allocated on the ***stack***

Objects allocation

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

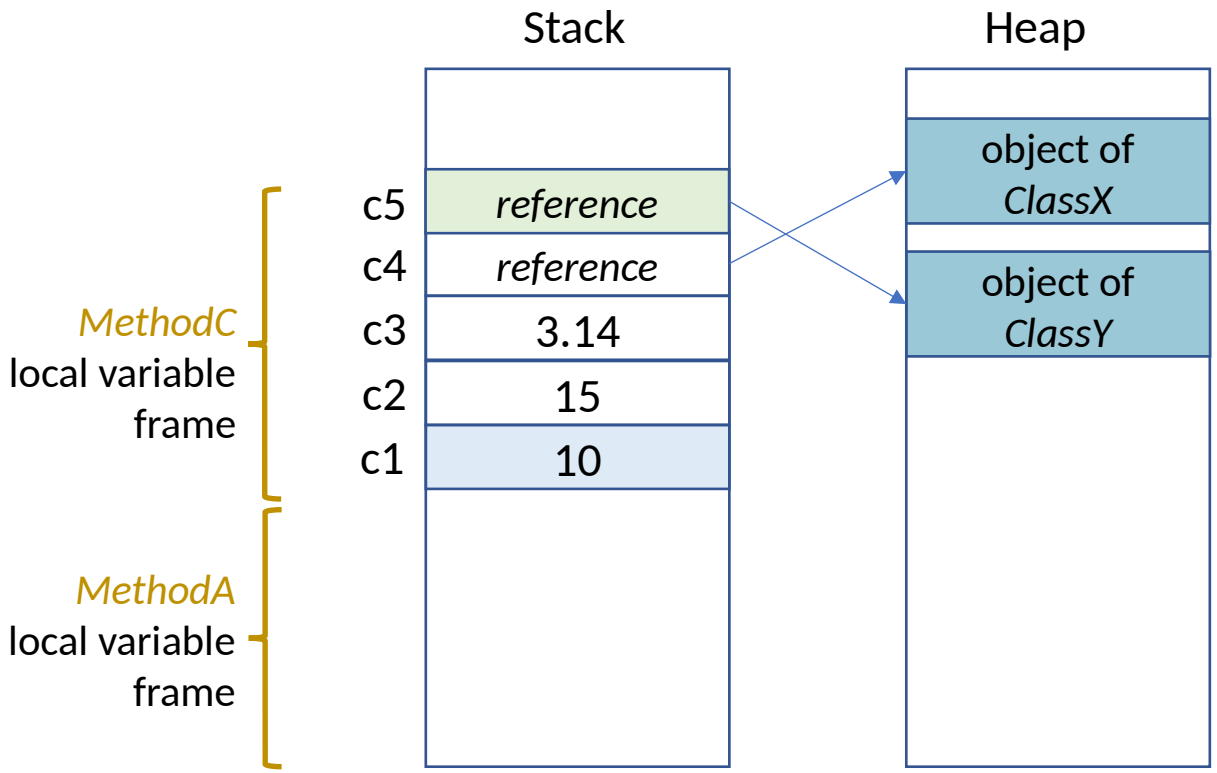
```
void MethodC(int c1)
{
   int c2 = 15;
   double c3 = 3.14;
   ClassX c4 = new ClassX();
}

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();
   ClassX c5 = new ClassY();
}

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

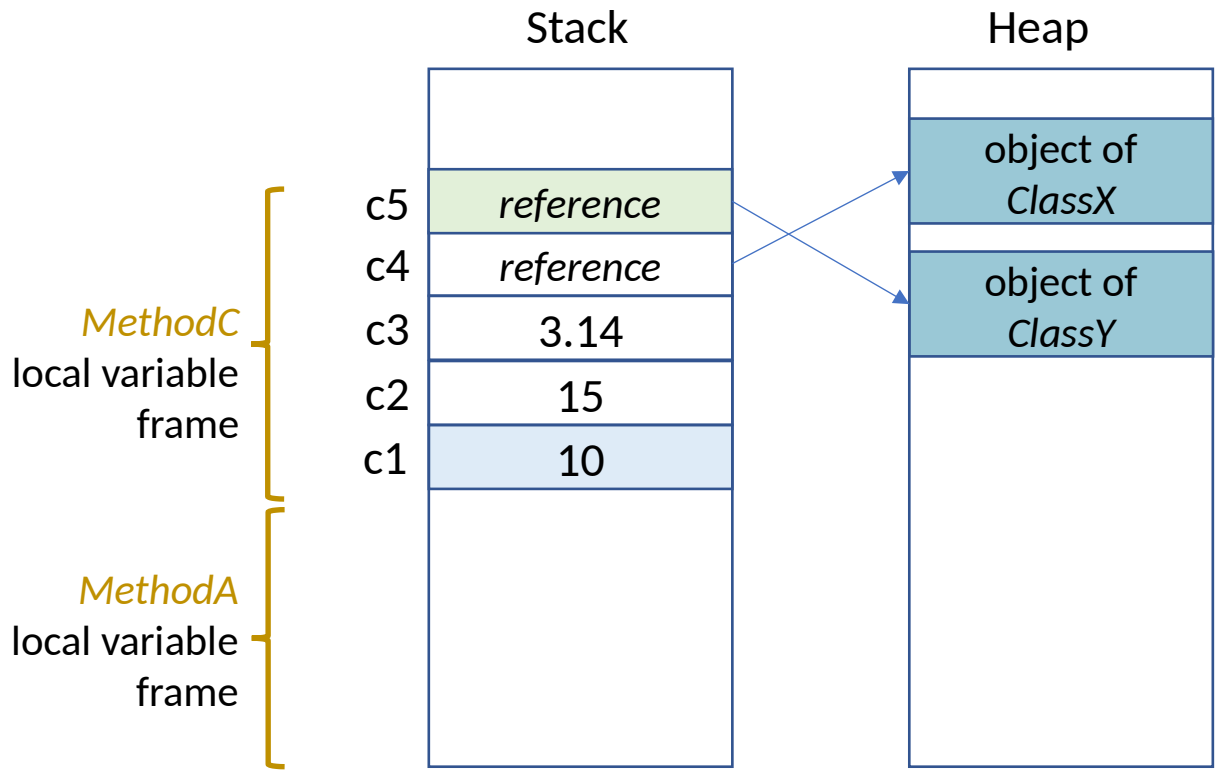


Objects allocation

Where will the *value type* attributes of *c4* be stored? Example coming soon…

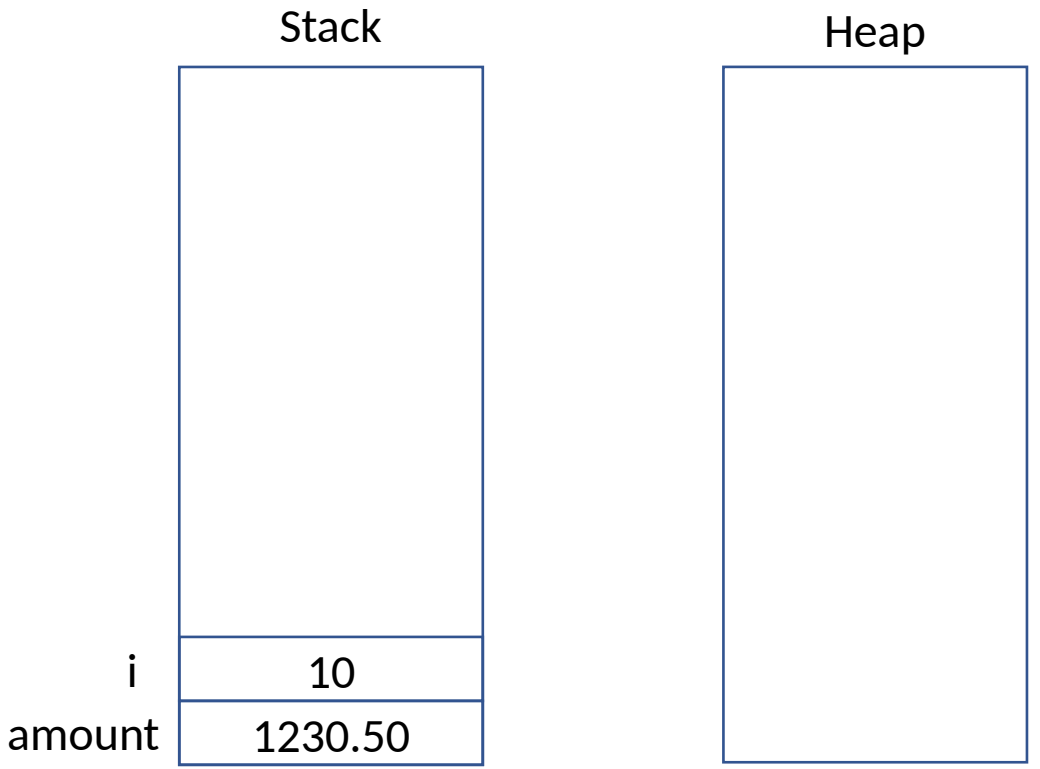
```
void MethodC(int c1)
{
  int c2 = 15;
  double c3 = 3.14;
  ClassX c4 = new ClassX();
  ClassX c5 = new ClassY();
}
```

```
class ClassX
{
  int attr1;
  int attr2;
  public ClassX() { … }
}
```



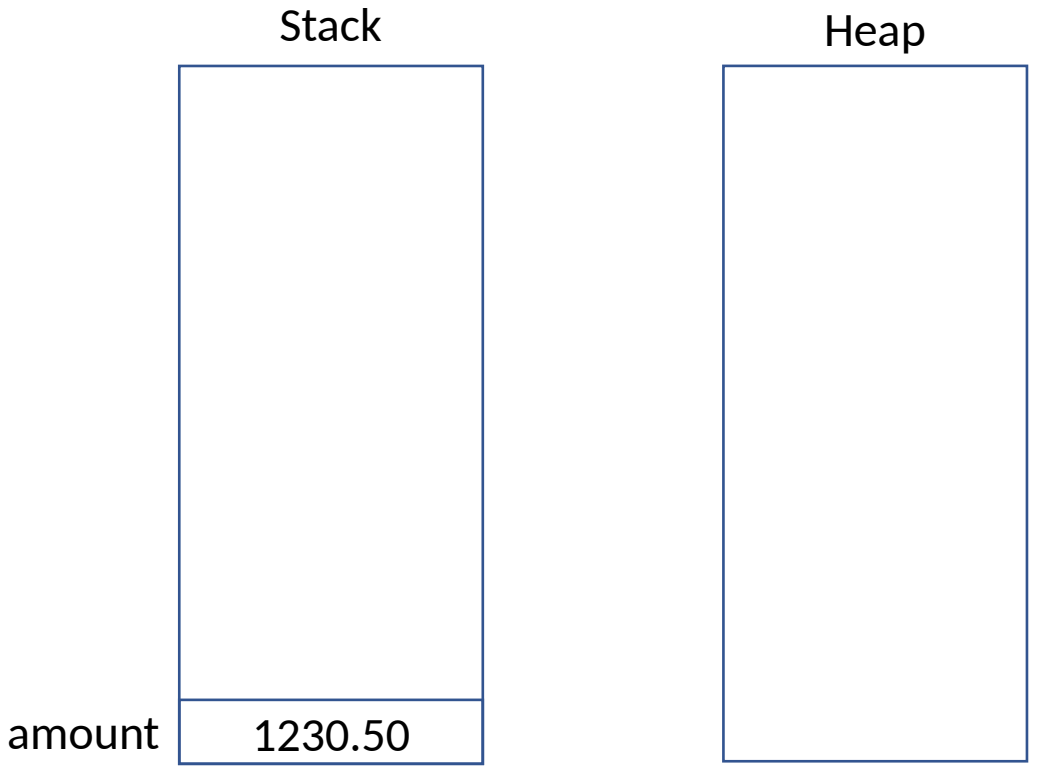
Example 1: double and int

```
class Program
{
    public static void Main()
    {
        double amount = 1230.50;
        int i = 10;
    }
}
```



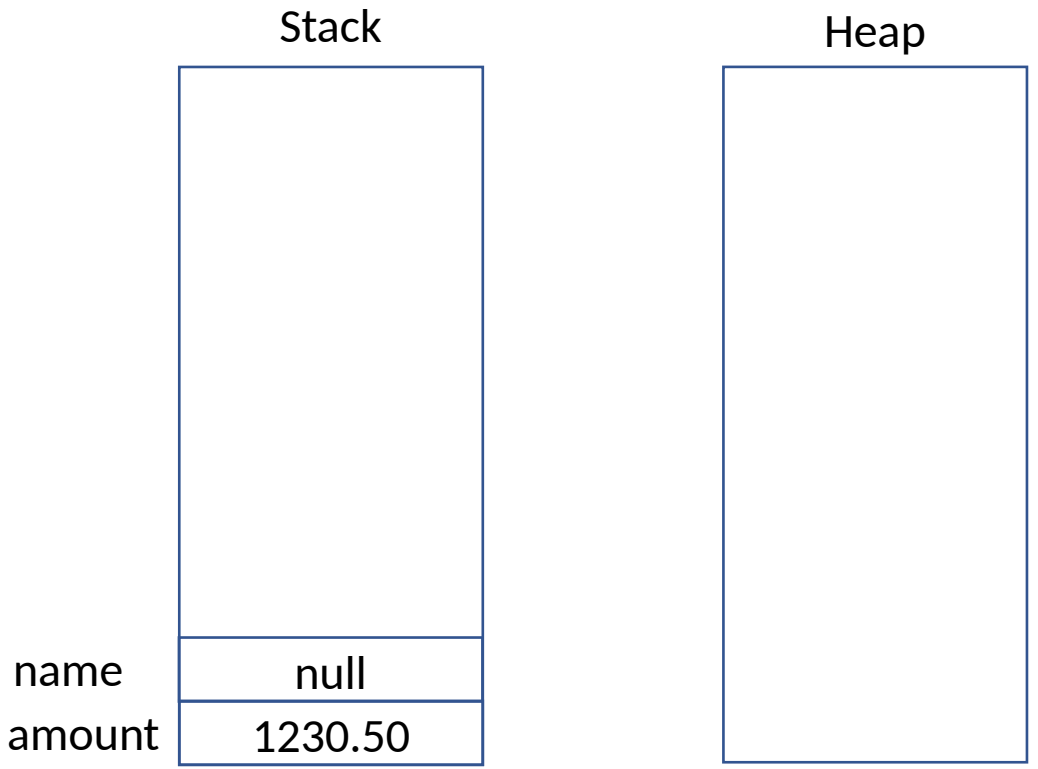
Example 2: double and string

```
class Program
{
    public static void Main()
    {
        double amount = 1230.50;

    }
}
```



Example 2: double and string

```
class Program
{
    public static void Main()
    {
        double amount = 1230.50;
        string name;
    }
}
```



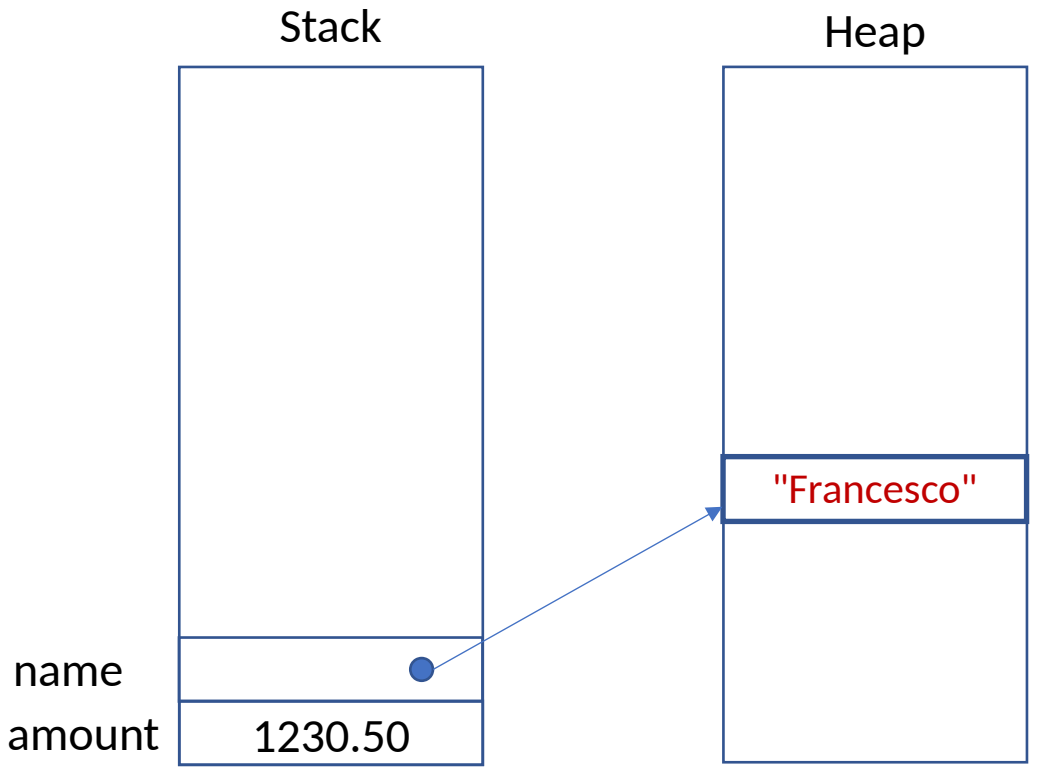
A string is an object—*name* holds a reference

null means that no object is referenced

it is the default value for a *non-initialised* reference variable

Example 2: double and string

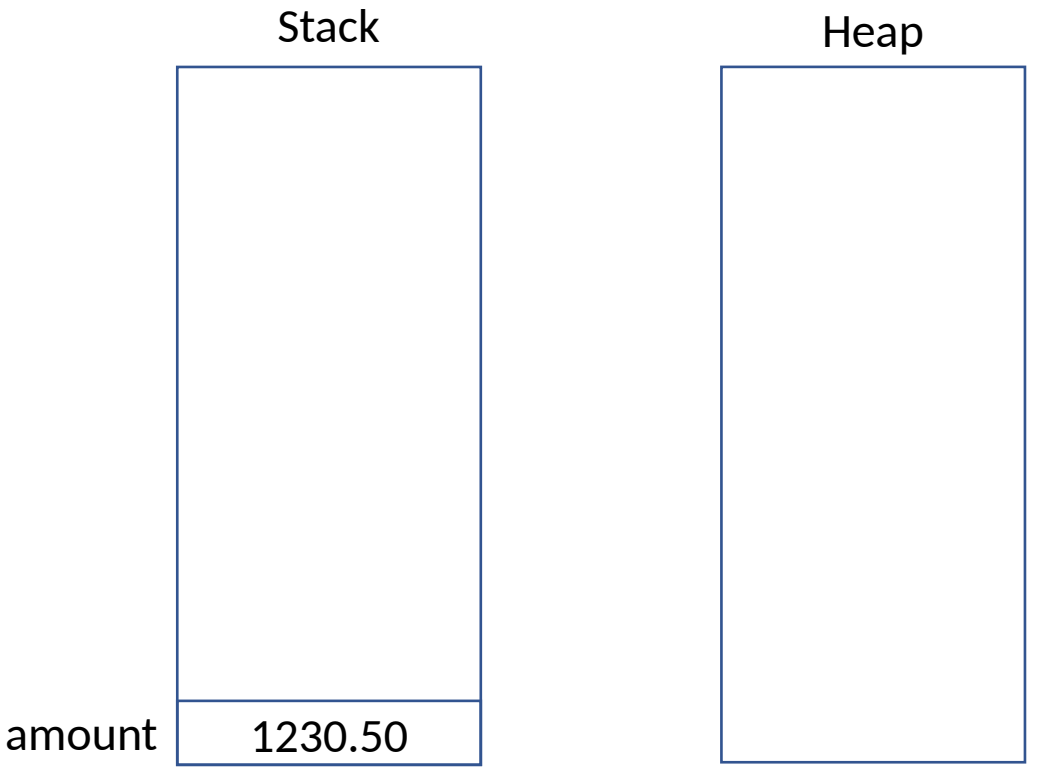
```
class Program
{
    public static void Main()
    {
        double amount = 1230.50;
        string name = "Francesco";
    }
}
```



name now references the memory location of the **heap** that contains the string object

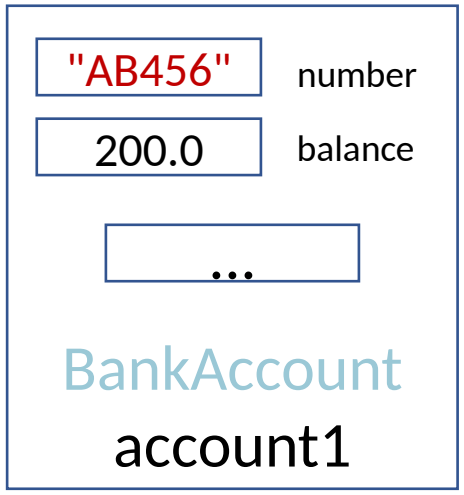
Example 3: double and *BankAccount*

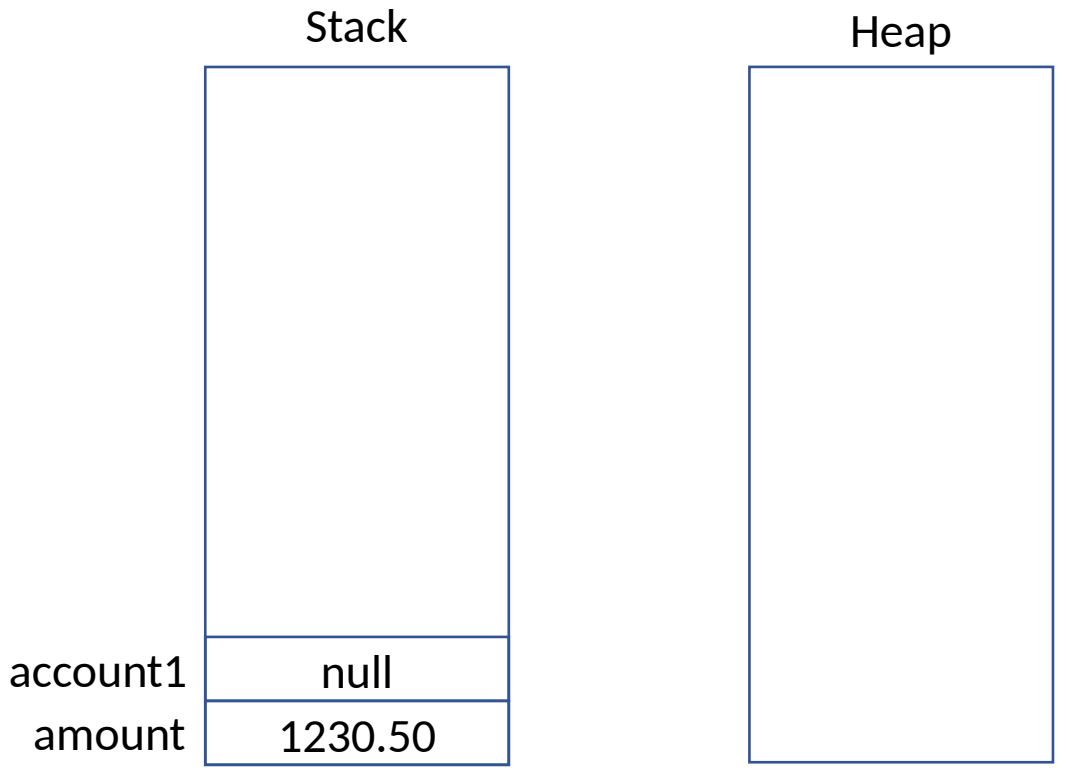
```
class Program
{
    public static void Main()
    {
        double amount = 1230.50;


    }
}
```



Example 3: double and *BankAccount*

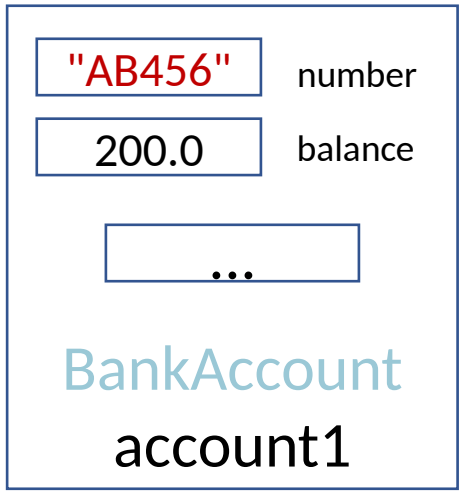
```
class Program
{
   public static void Main()
   {
      double amount = 1230.50;
      BankAccount  account1;

   }
}
```

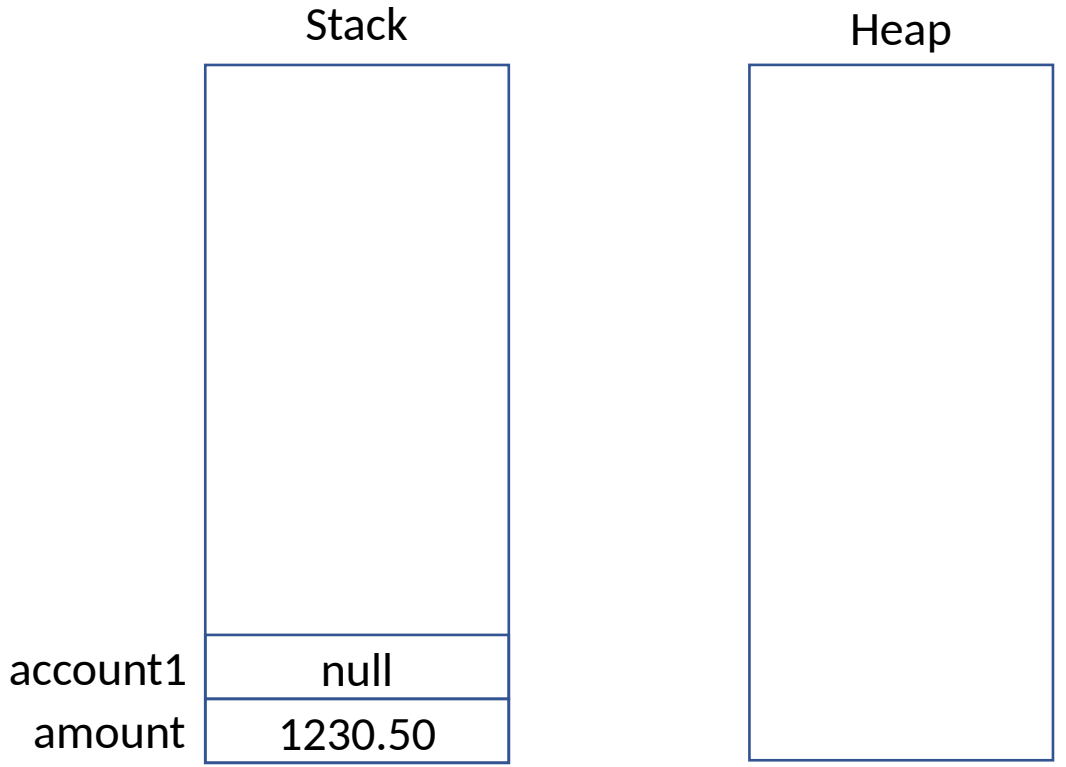




Example 3: double and *BankAccount*

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

   }
}
```

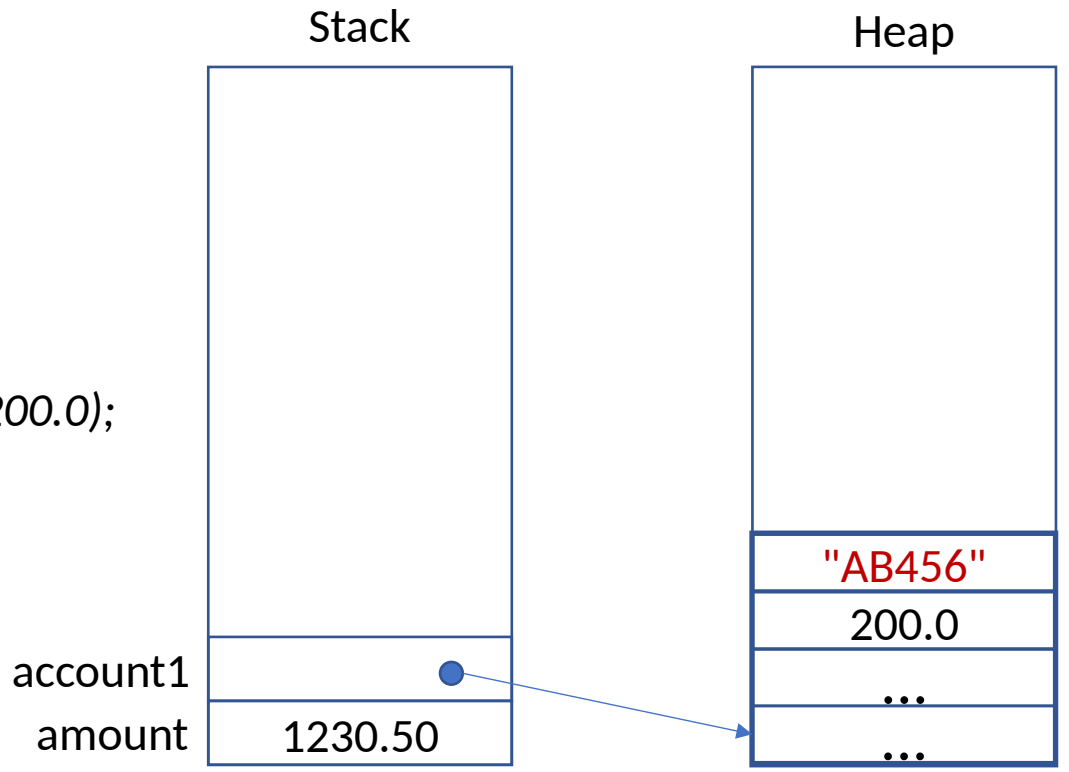




When the object is created, where will the *attributes* be stored?

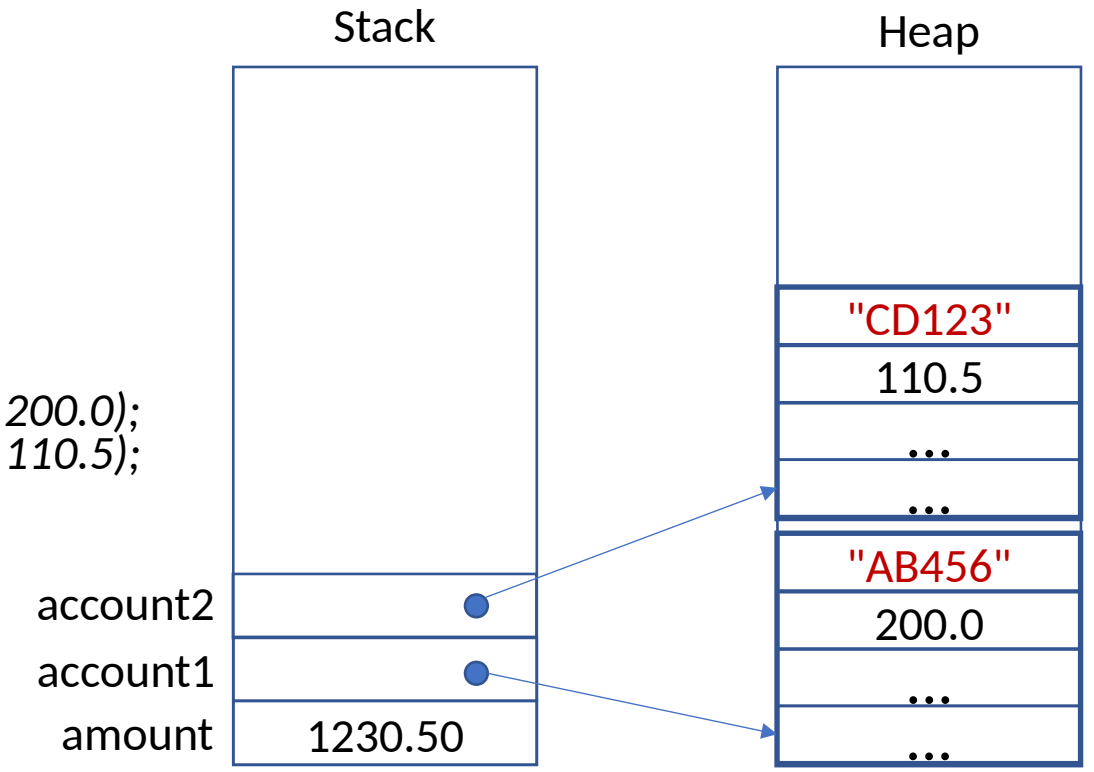
Example 3: double and *BankAccount*

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

    }
}
```



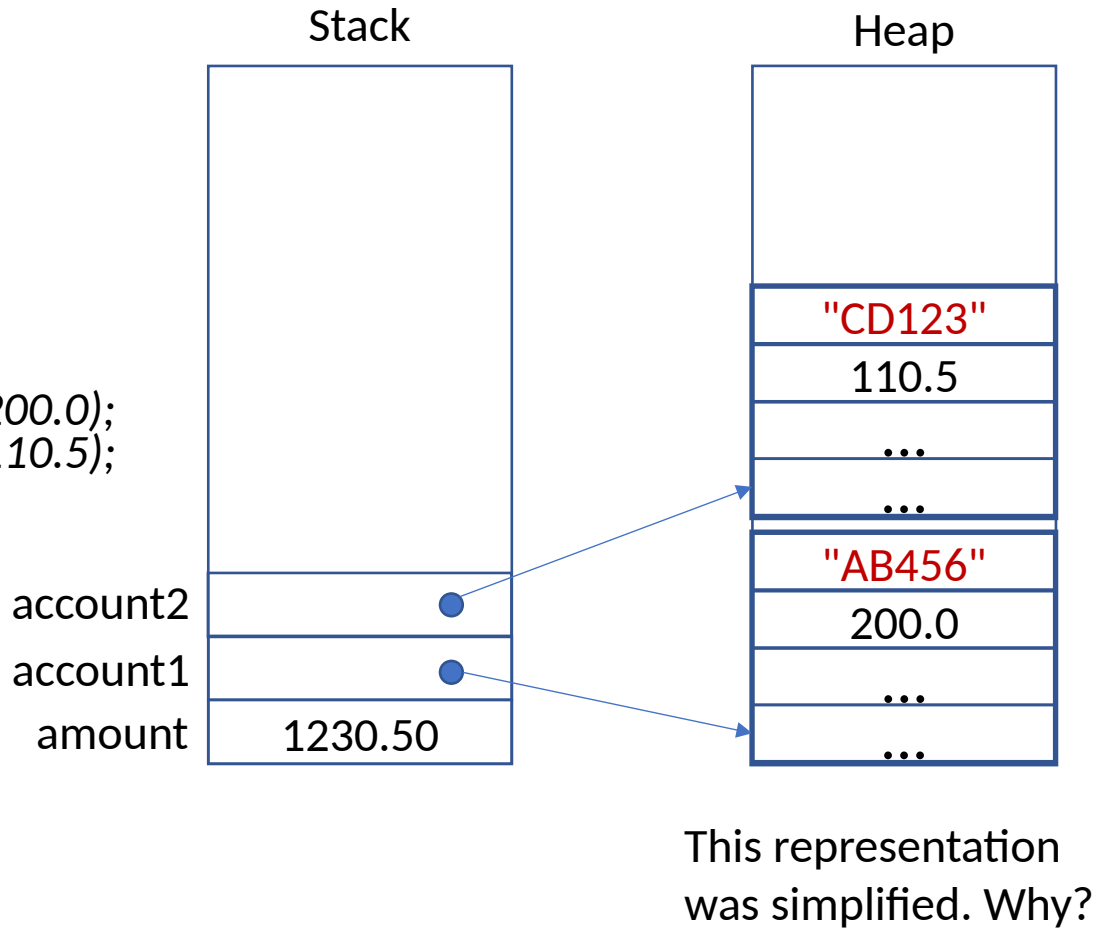
Example 3: double and *BankAccount*

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



Example 3: double and *BankAccount*

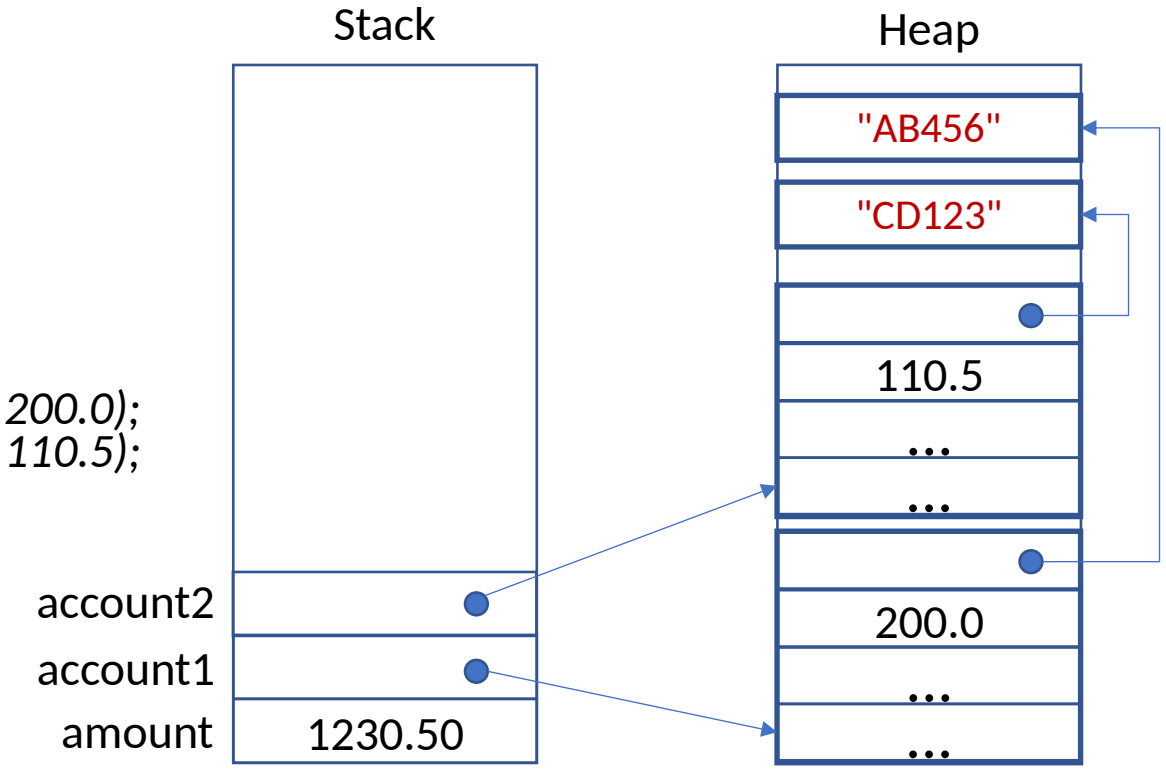
```
class Program
{
    public static void Main()
    {
        double amount = 1230.50;
        BankAccount  account1 = new BankAccount("AB456", 200.0);
        BankAccount  account2 = new BankAccount("CD123", 110.5);
    }
}
```



This representation was simplified. Why?

Example 3: 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);
   }
}
```



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

its content may be stored somewhere else on the heap

Methods parameters: reference types

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg) { ... }

        public void Display() { ... }
        public void Move(int xm, int ym) { ... }
        public int GetX() { ... }
        public int GetY() { ... }

        public double DistanceFrom(Point p)
        {
            double distance = Math.Sqrt(Math.Pow(p.x - x, 2) + Math.Pow(p.y - y, 2));
            return distance;
        }
}
```

Methods parameters: reference types

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg) { ... }

        public void Display() { ... }
        public void Move(int xm, int ym) { ... }
        public int GetX() { ... }
        public int GetY() { ... }

        public double DistanceFrom(Point p)
        {
            double distance = Math.Sqrt(Math.Pow(p.x - x, 2) + Math.Pow(p.y - y, 2));
            return distance;
        }
}
```

*What happens when a **reference type** (e.g., a Point object) is passed by value to a method?*

Methods parameters: reference types

```
class Point
{
        private int x;
        private int y;

        public Point(int xarg, int yarg) { ... }

        public void Display() { ... }
        public void Move(int xm, int ym) { ... }
        public int GetX() { ... }
        public int GetY() { ... }

        public double DistanceFrom(Point p)
        {
            double distance = Math.Sqrt(Math.Pow(p.x - x, 2) + Math.Pow(p.y - y, 2));
            return distance;
        }
}
```

*A **copy of the reference** to the object, not a copy of the object, will be passed…*

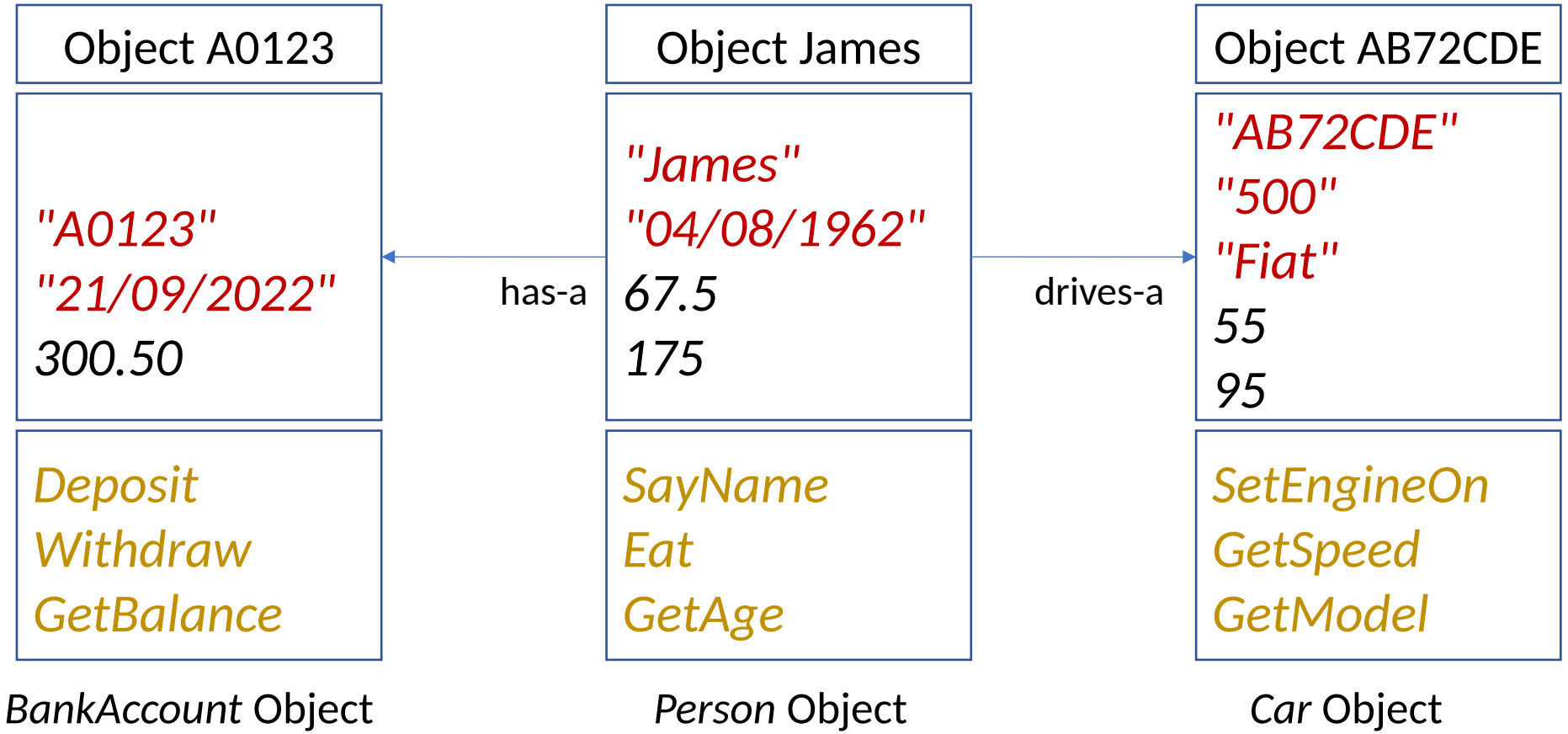
Reflect on the following (before week 7)

- What happens with ***reference type*** *variables* when:
  - They are checked for ***equality*** (==)
  - They are used in ***assignment*** (=) instructions
  - They are ***passed by value*** to a *method*

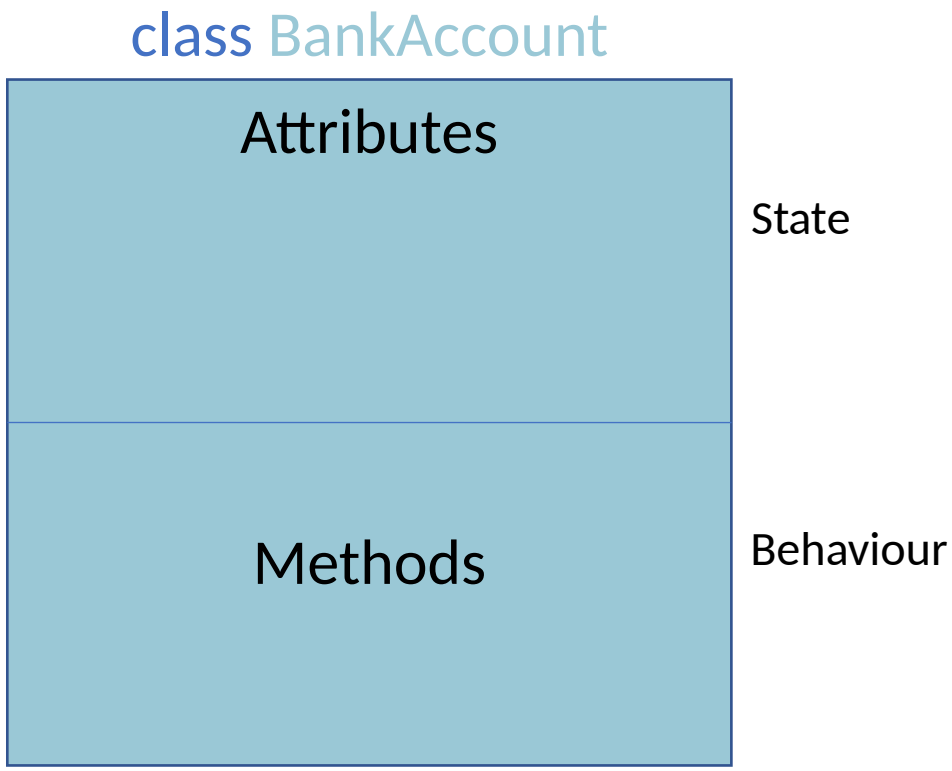
Outline

- Summary of last week and more on methods
- Value types and Reference types, Stack and Heap
- UML class diagrams

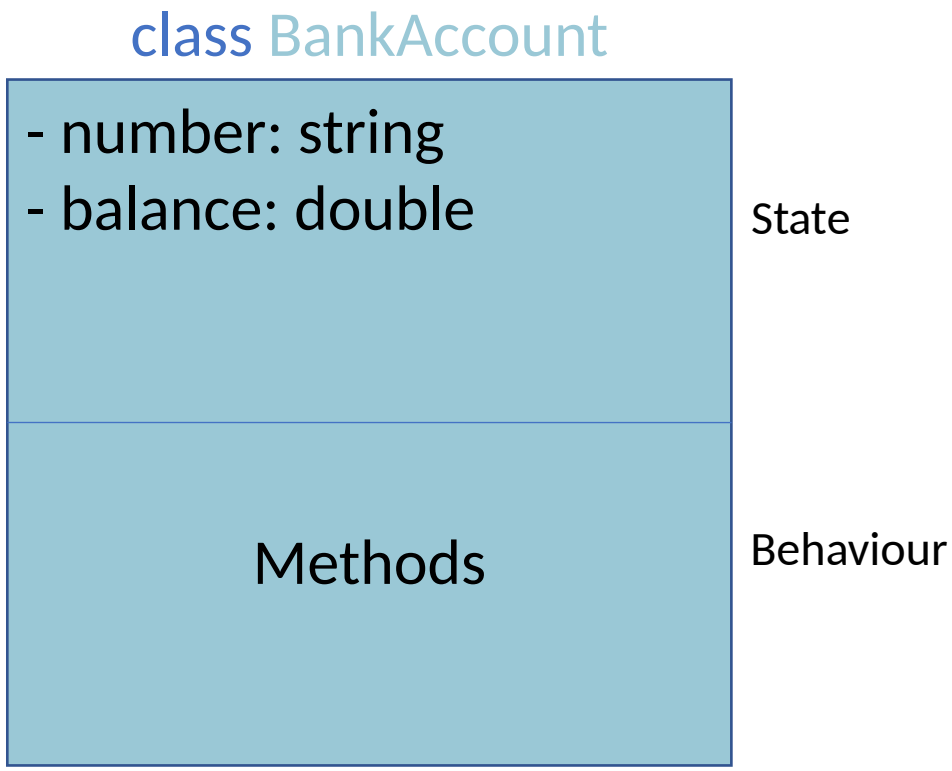
Objects Representation



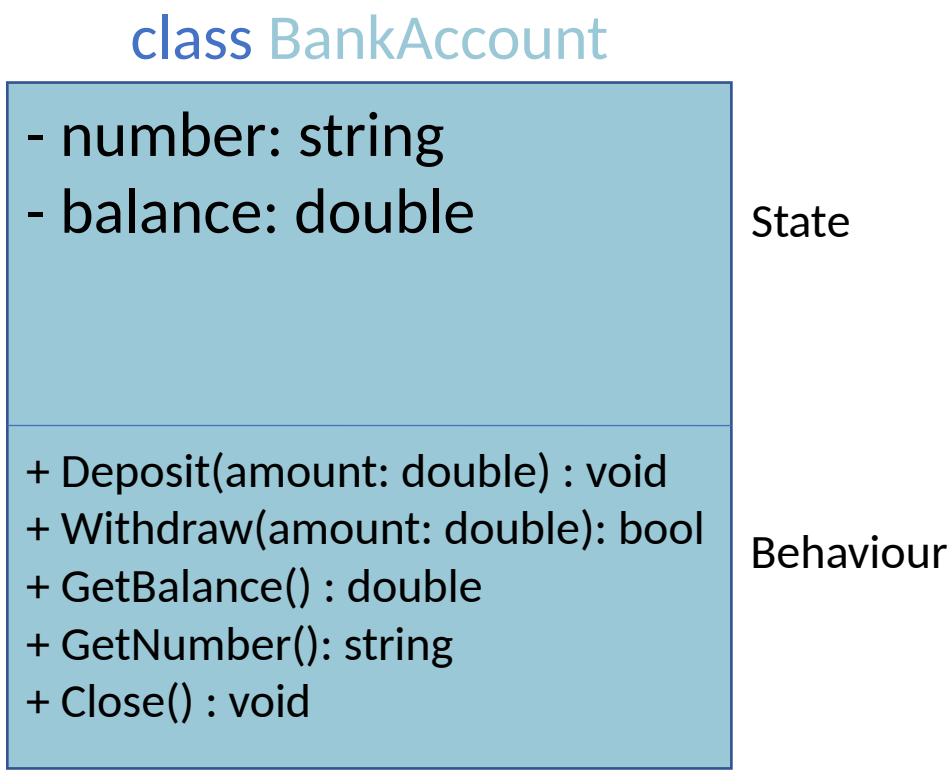
Bank account



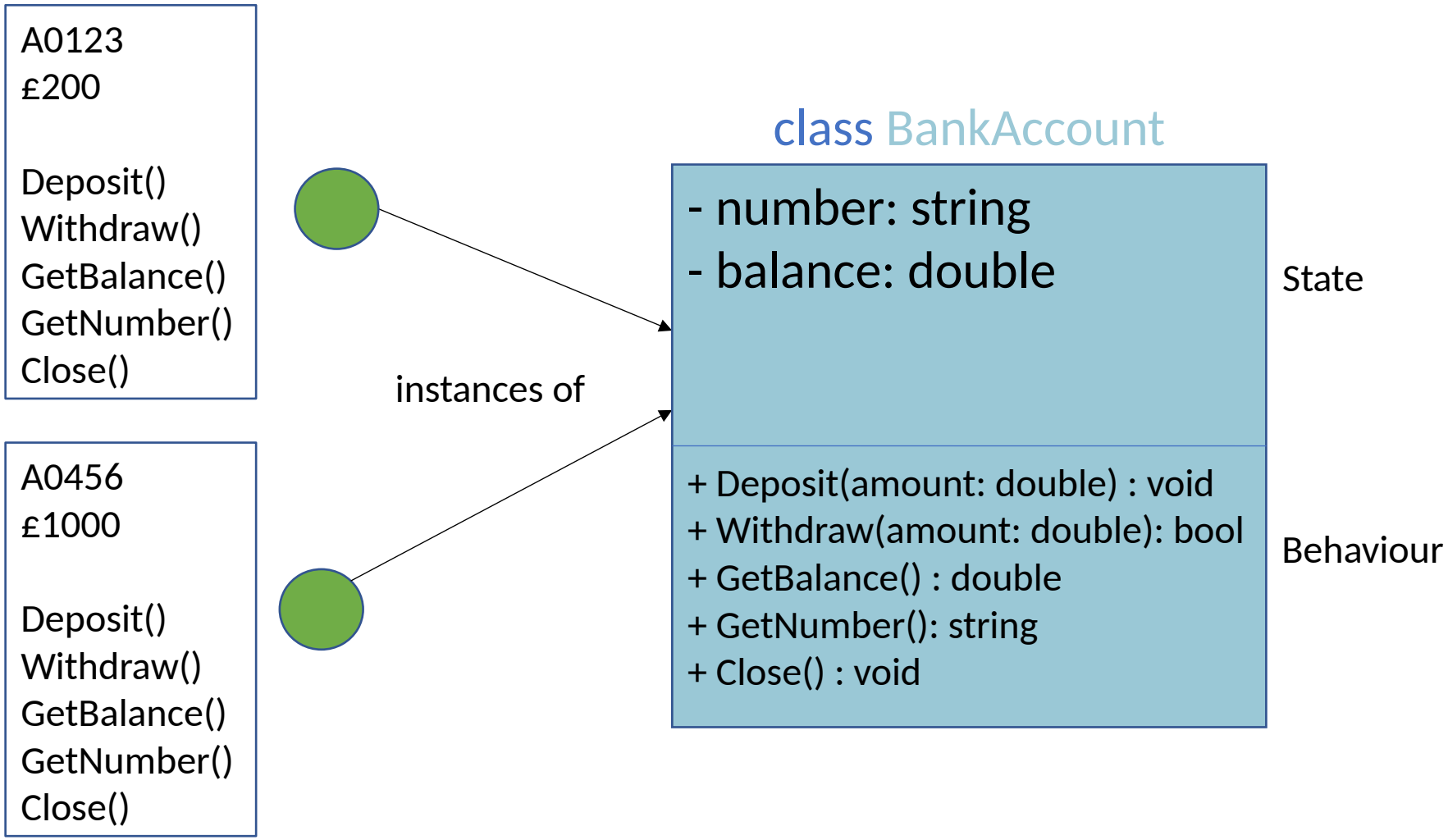
Bank account: attributes



Bank account: methods



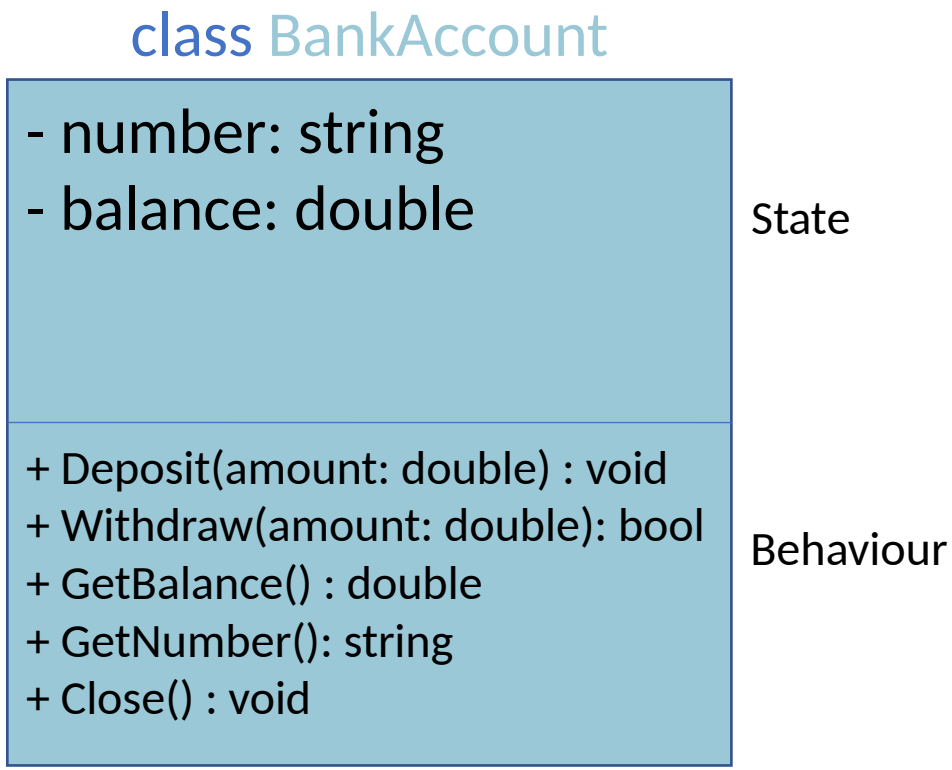
Bank account: object instances



Bank account: Class Diagram

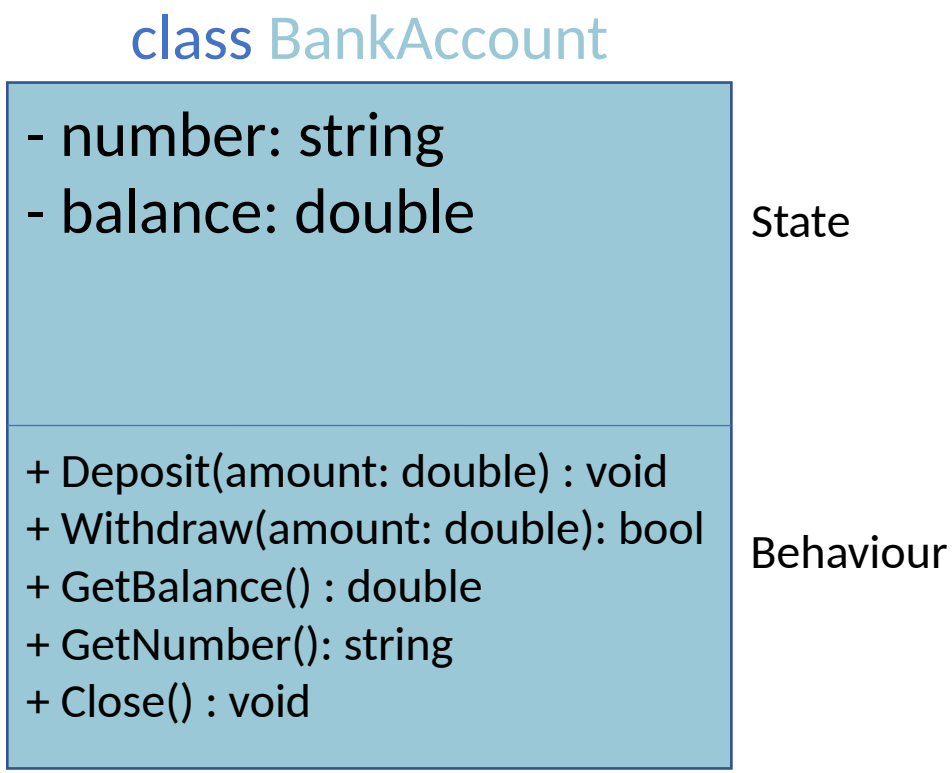
UML (Unified Modelling Language)

Class Diagram



Bank account: Class Diagram

+ denotes a public member
- denotes a private member



Bank account: Class Diagram

Think about a possible definition of a BankAccount class based on this diagram.

More on this later in the lab!