7SENG011W Object Oriented Programming

More on Memory Management, Static Attributes and Static Methods

Dr Francesco Tusa

Readings

Books

- Head First Java
  - Chapter 10
- Beginning Programming with Java For Dummies
  - Chapter 14

Online

- Java Language Specification – Chapter 8. Classes

Outline

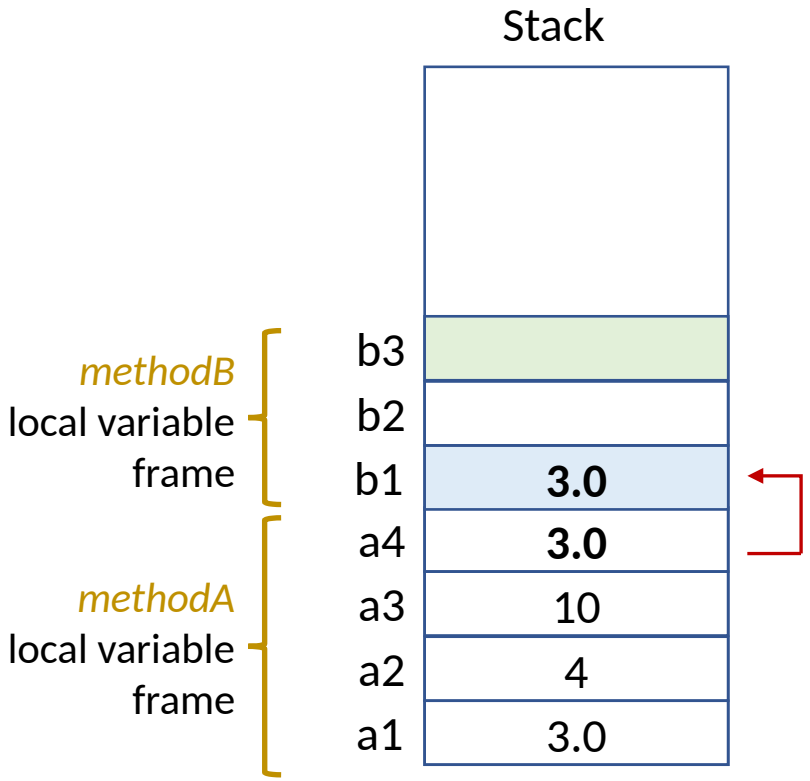
- More on Memory Management
  - Parameter Passing
  - Objects lifetime
  - 'this' keyword
- Static
  - Attributes
  - Methods
  - main method

Method Invocation

Java'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);
}
```



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 and stored in the corresponding *parameters*
- The above stack area is **deallocated** when the method **terminates**

Methods parameters: value types

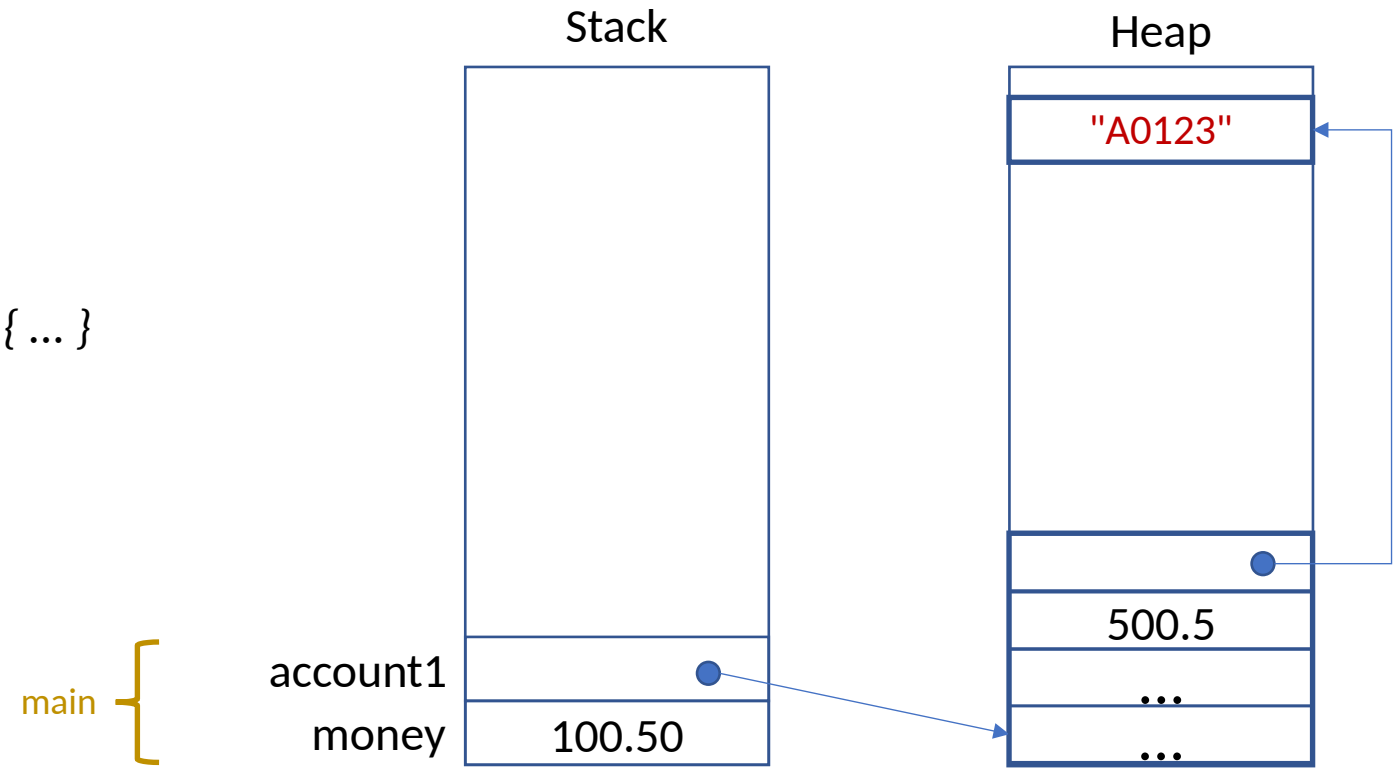
```
class BankAccount
{
    private String number;
    private double balance;

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

    public void deposit(double amount)
    {
        amount *= 1.05; // 5% interest rate
        balance += amount;
    }
}

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


}
```

In the following examples we use a simplified version of *BankAccount* with no error handling.

Here, the *deposit* method accepts one parameter and always applies a fixed interest of 5%.

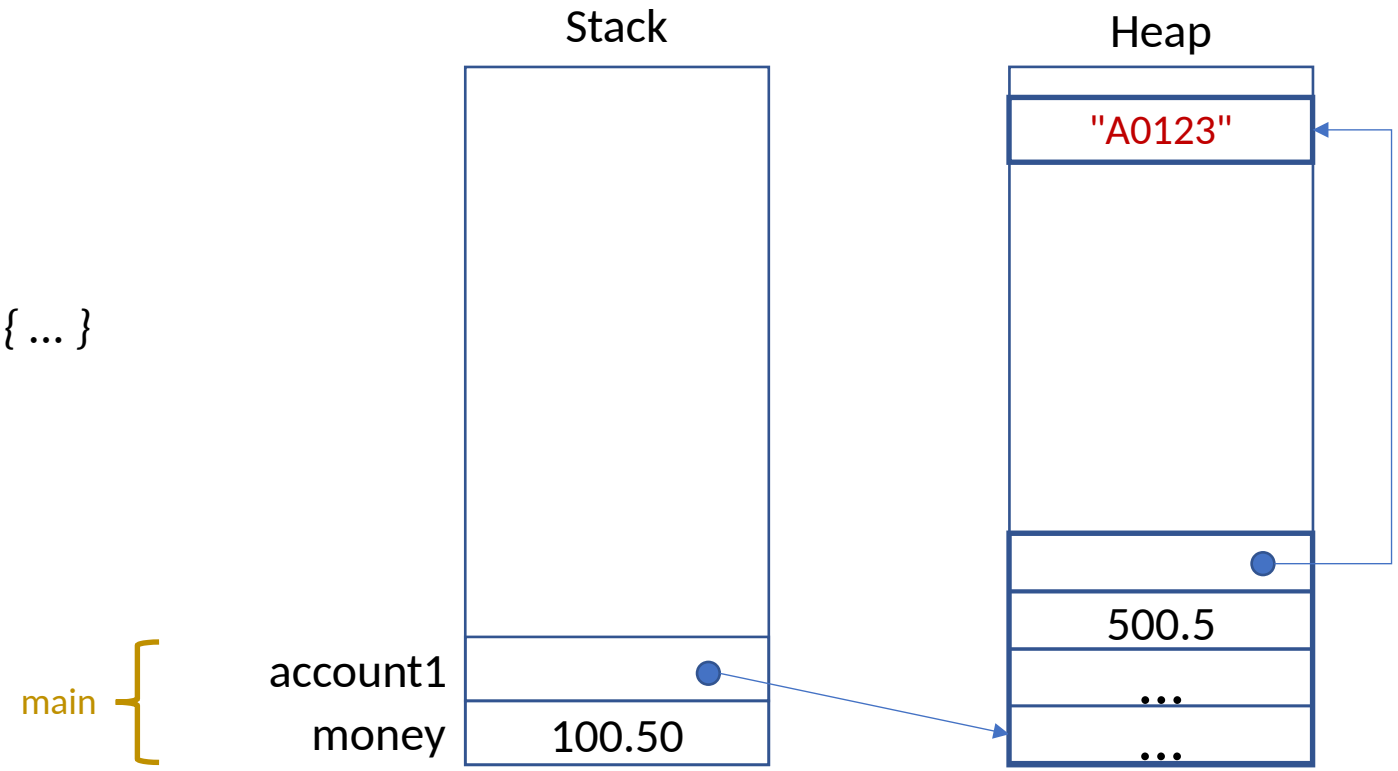
Methods parameters: value types

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

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

    public void deposit(double amount)
    {
        amount *= 1.05;
        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;
        balance += amount;
    }
}

class Program
{
    public static void main(String[] args)
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);

}
```



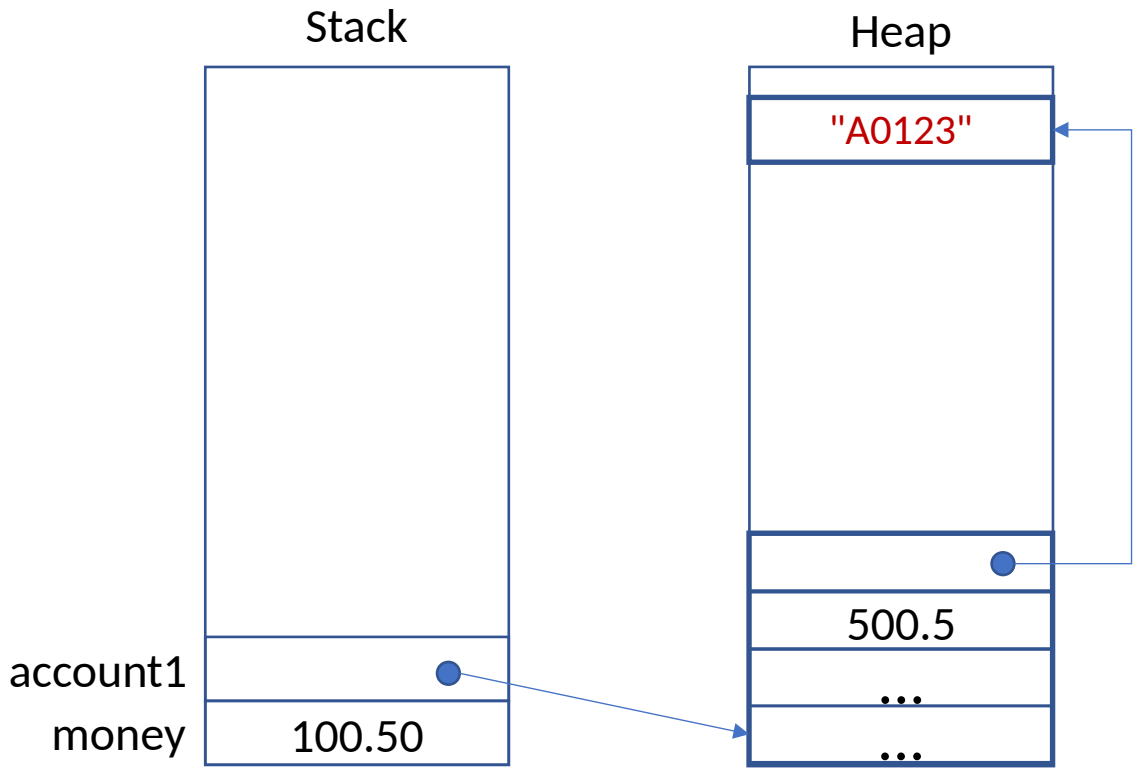
The *args* array is not shown, where should it go?

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;
        balance += amount;
    }
}

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

deposit

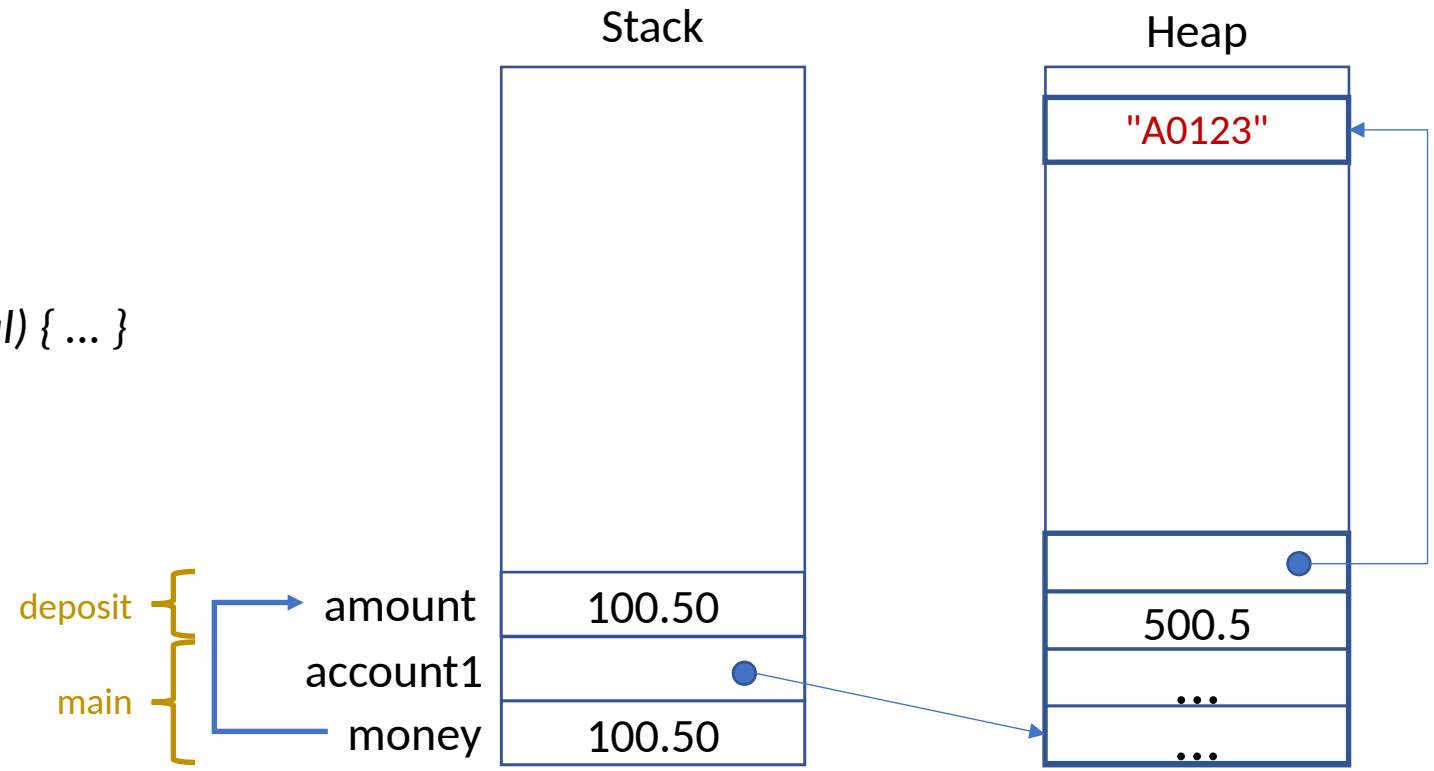
main



- What happens on the stack and heap memory when the deposit method is called on *account1*?
- What will be the content of *money* after deposit terminates?

Methods parameters: value types

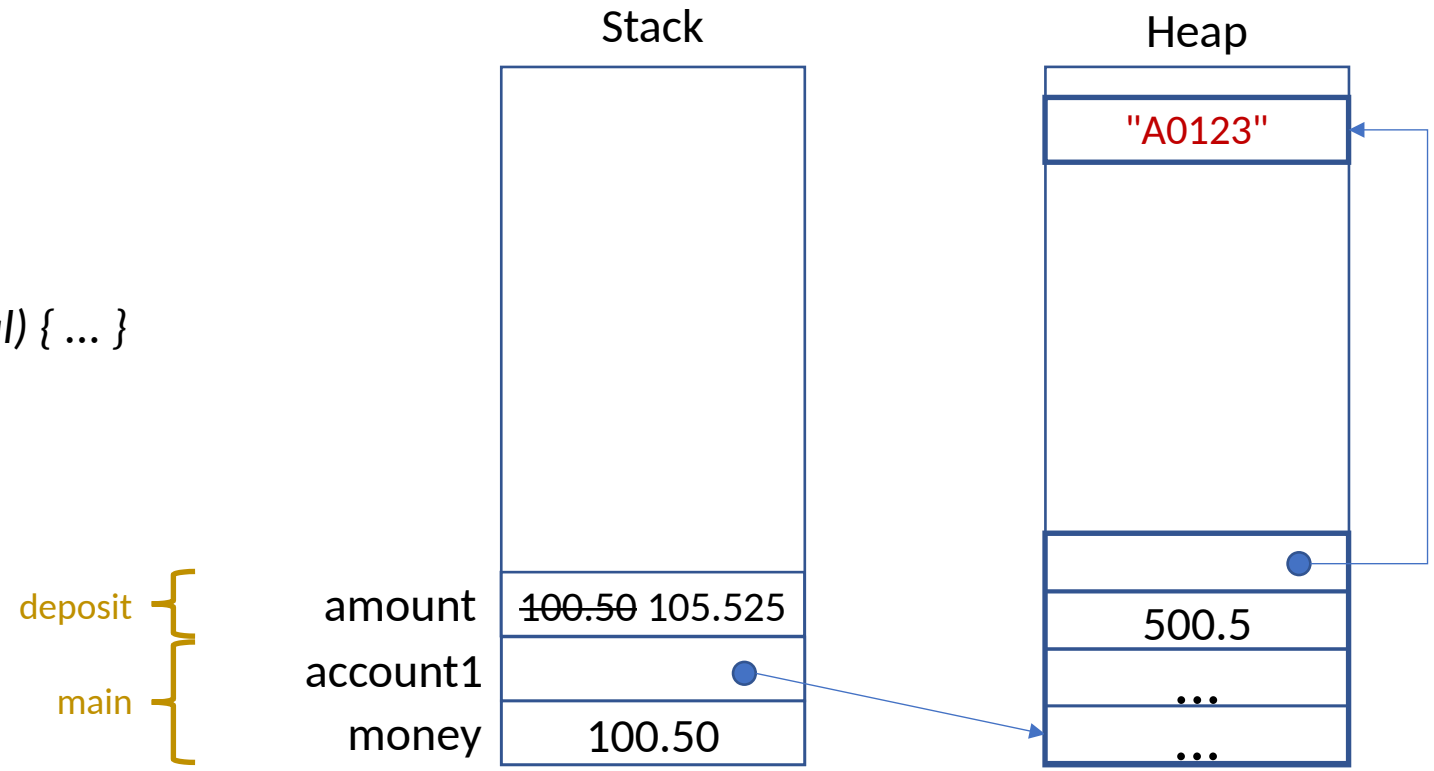
```
class BankAccount
{
    private String number;
    private double balance;

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

    public void deposit(double amount)
    {
        amount *= 1.05;
        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

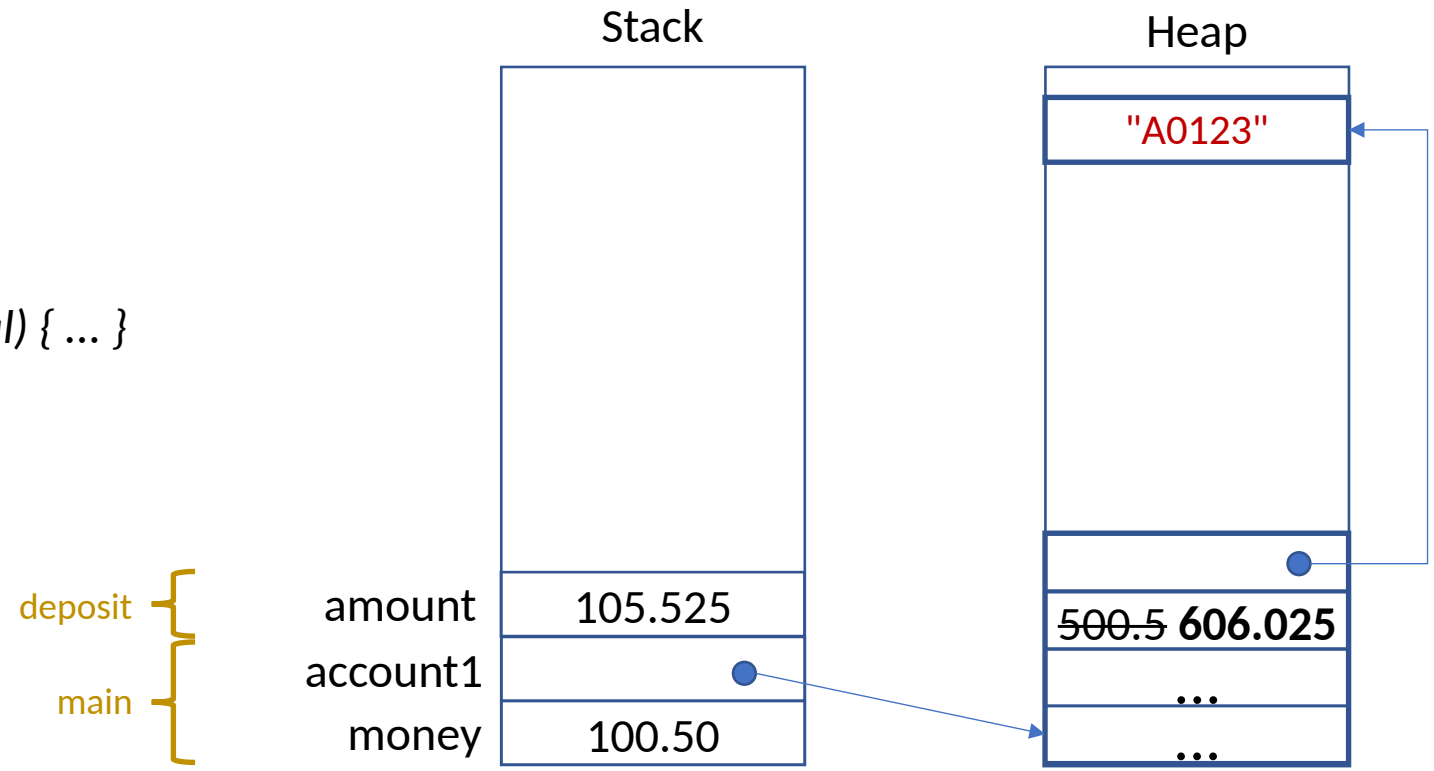
```
class BankAccount
{
    private String number;
    private double balance;

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

    public void deposit(double amount)
    {
        amount *= 1.05;
        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

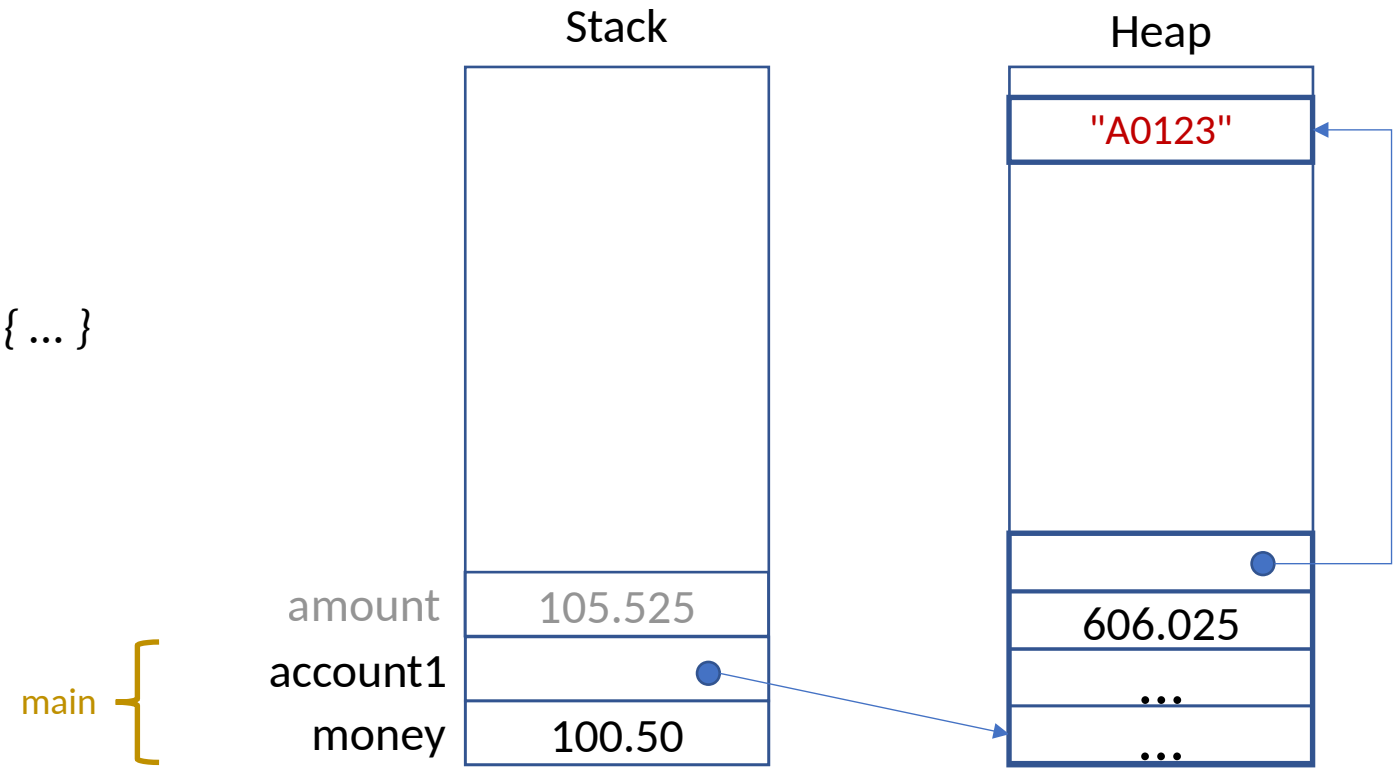
```
class BankAccount
{
    private String number;
    private double balance;

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

    public void deposit(double amount)
    {
        amount *= 1.05;
        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 **all the attributes** of the object *account1* (how explained soon)

Methods parameters: value types

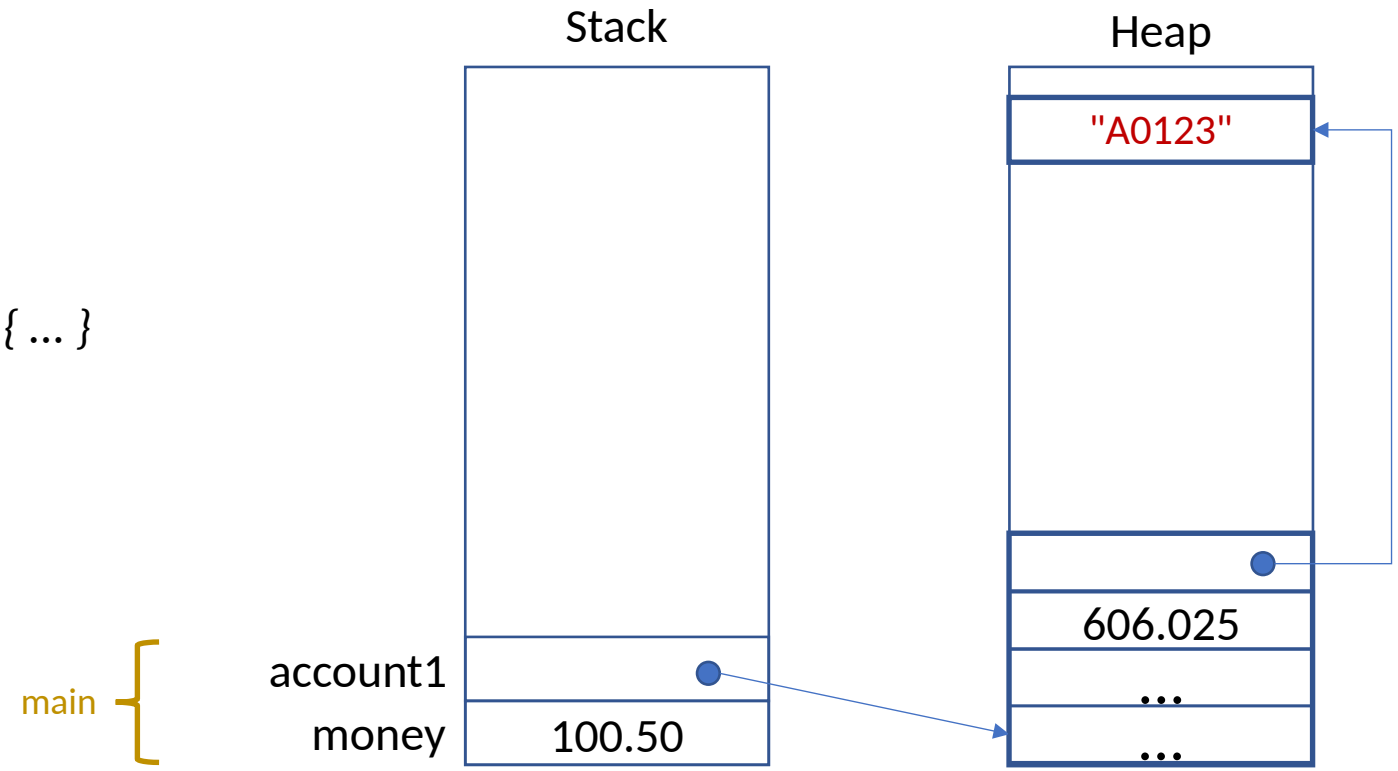
```
class BankAccount
{
    private String number;
    private double balance;

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

    public void deposit(double amount)
    {
        amount *= 1.05;
        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, its stack frame with the *amount* variable is removed from the stack

Methods parameters: value types

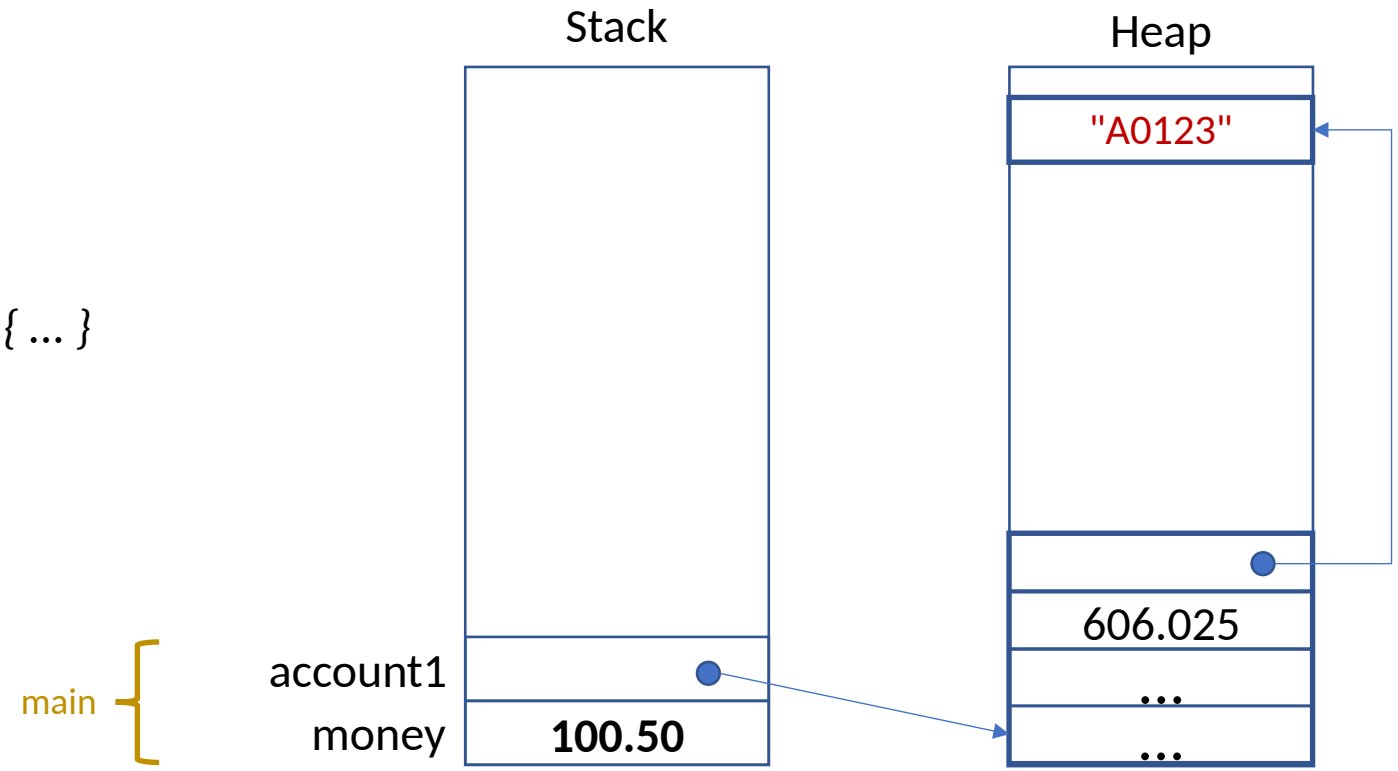
```
class BankAccount
{
    private String number;
    private double balance;

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

    public void deposit(double amount)
    {
        amount *= 1.05;
        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;
        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**

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 method parameters of reference types?

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
        // e.g., amount=500.5 with 6.8% interest => 534.534
    }
}
```

```
class Program
{
    public static void main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 6.8);

        ...
}
```

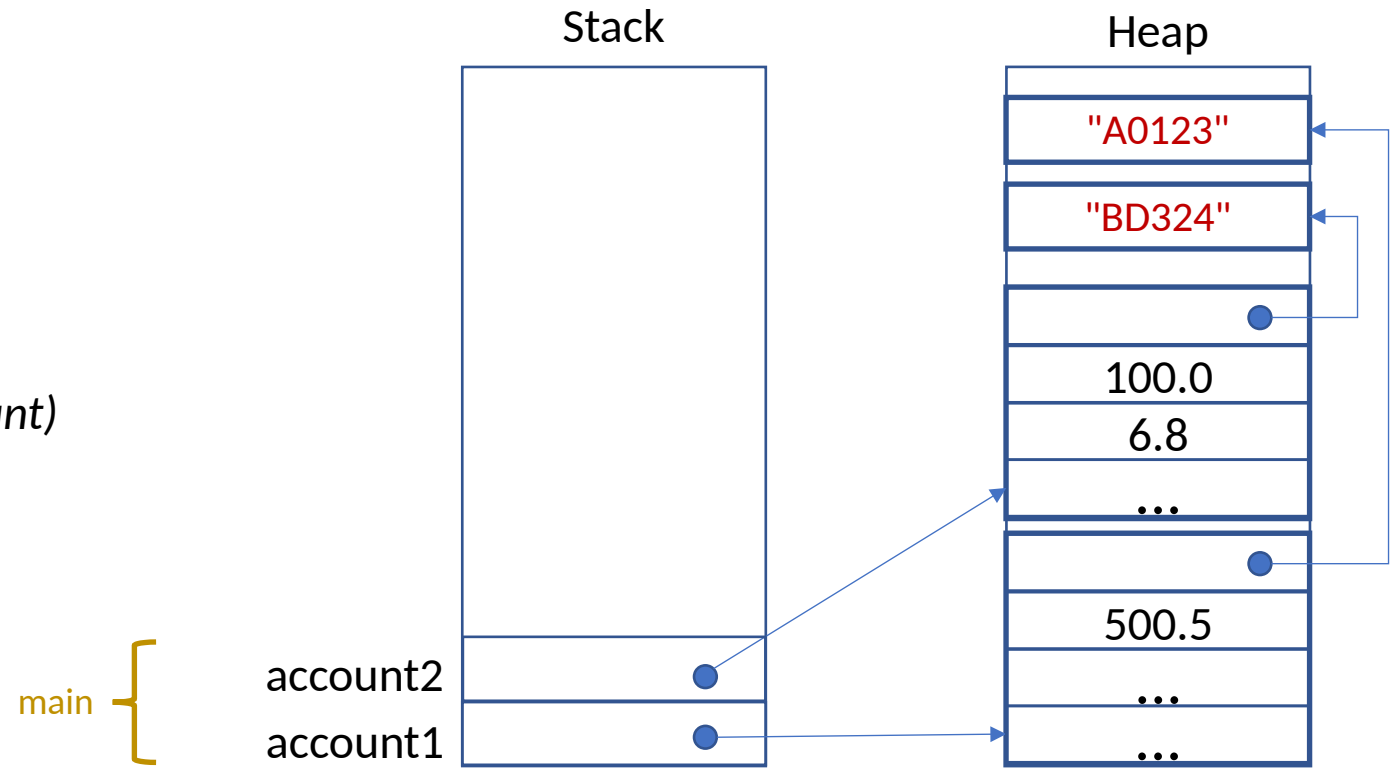
The object *account2* is of the class *SavingAccount*

A new method *moveAccount* in *BankAccount* moves the balance to the *SavingAccount* object passed as the argument and closes the account:

account1.moveAccount(account2) moves all the money from *account1* to *account2*.

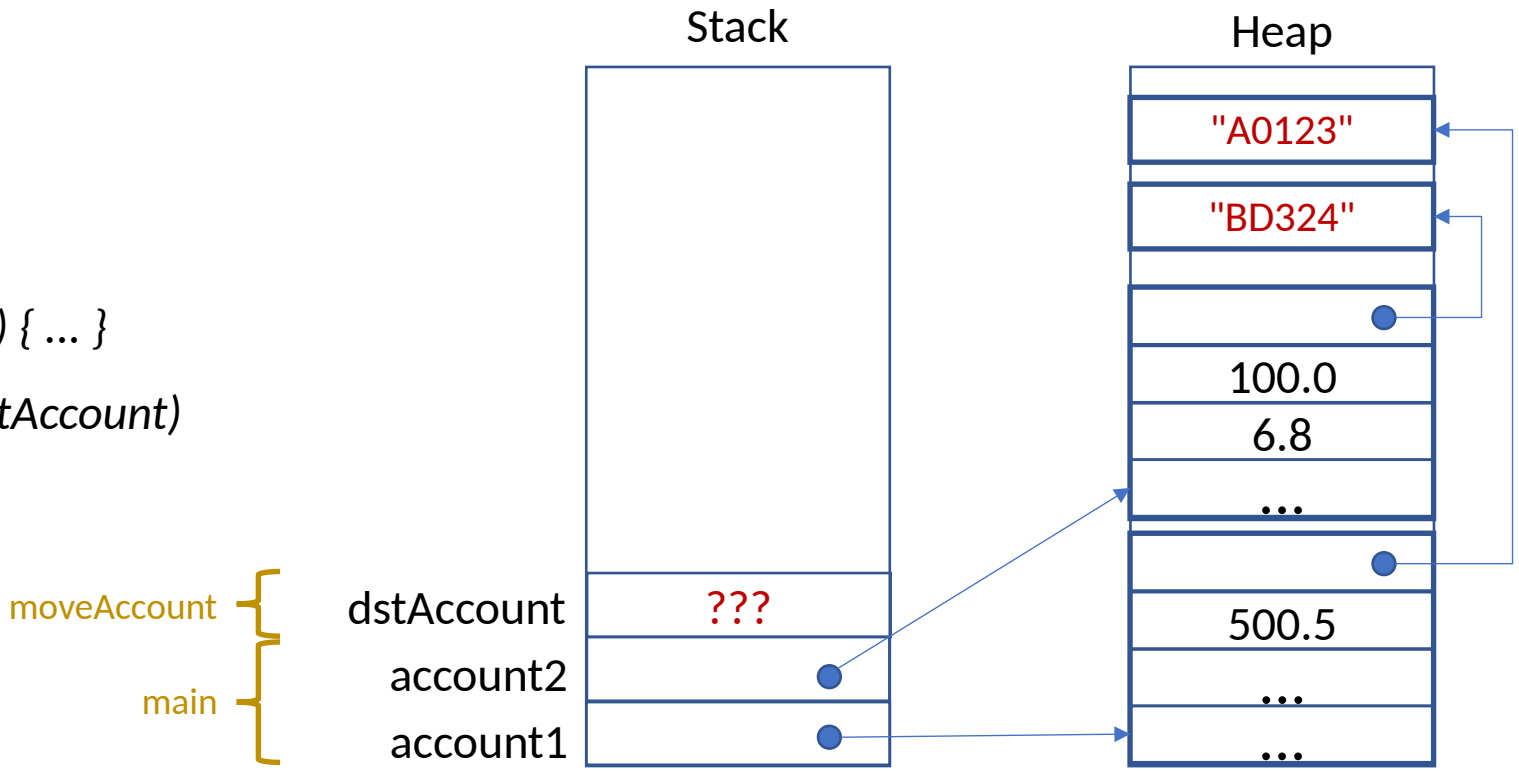
Methods parameters: reference types

```
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, 6.8);

        ...
}
```



Methods parameters: reference types

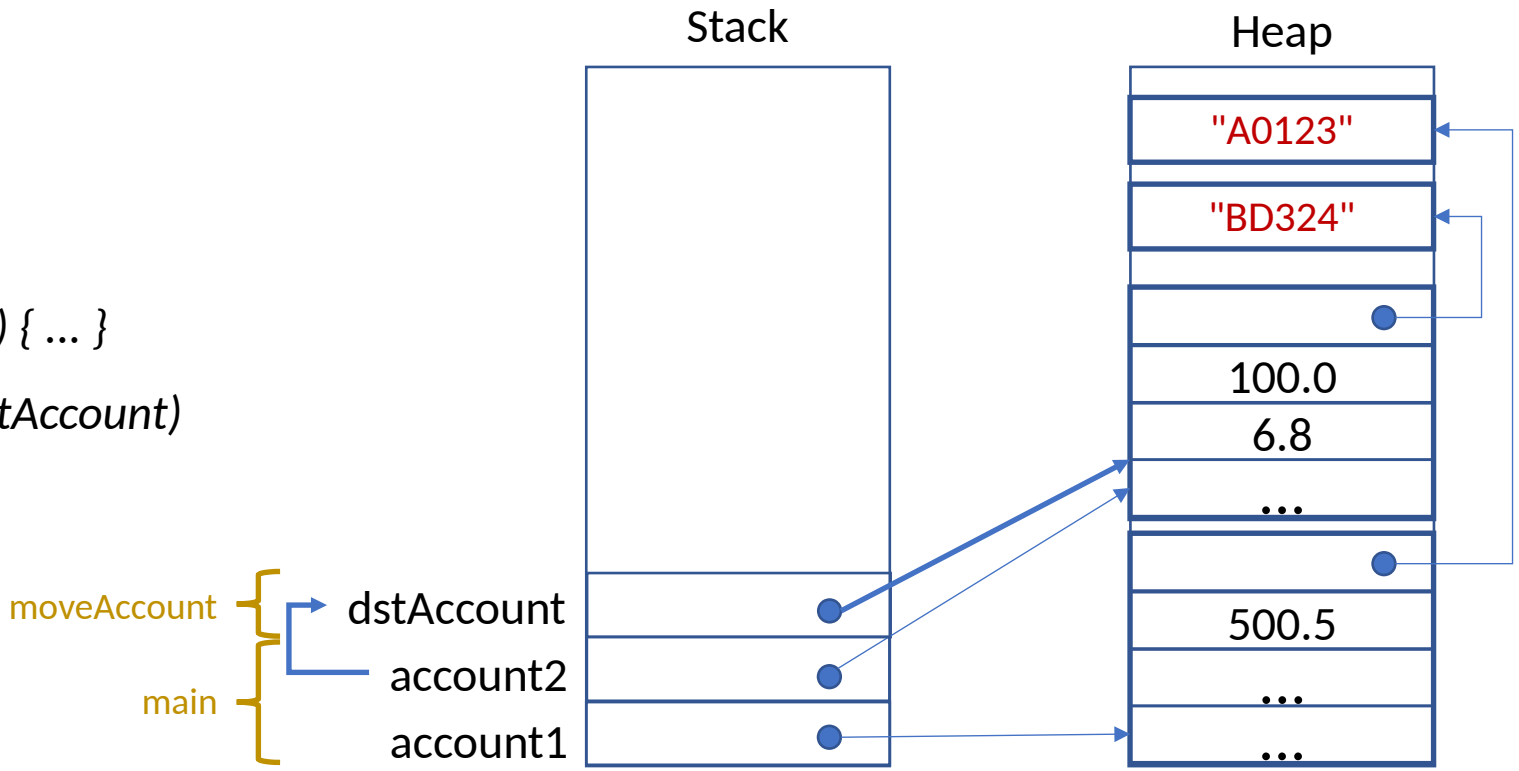
```
class BankAccount
{
    private String number;
    private double balance;

    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, 6.8);
        account1.moveAccount(account2);
        ...
}
```



- What happens on the stack and heap memory when the *moveAccount* method is called on *account1*?
- What will be the status of *account1 and account2* after *moveAccount* terminates?

Methods parameters: reference types

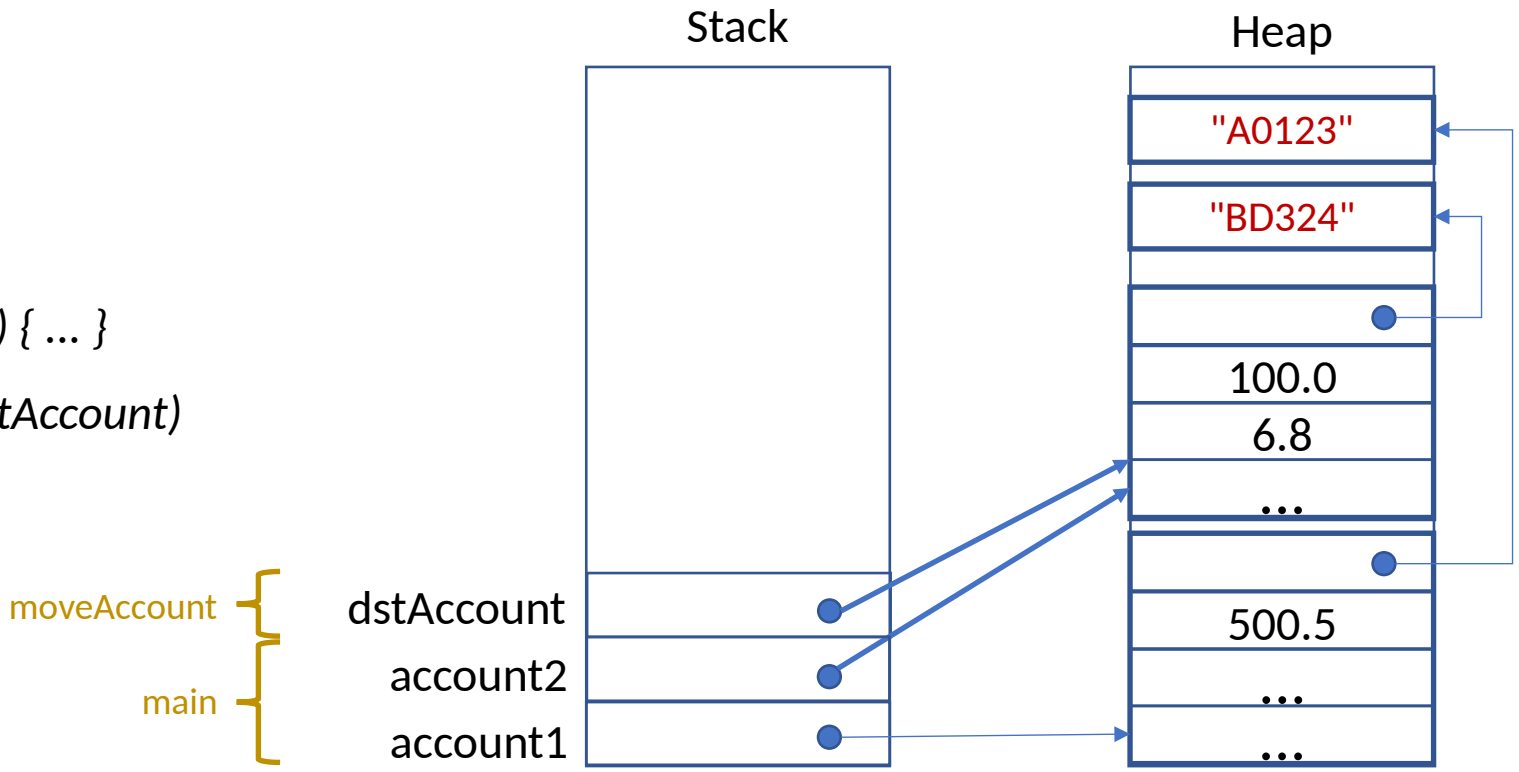
```
class BankAccount
{
    private String number;
    private double balance;

    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, 6.8);
        account1.moveAccount(account2);
        ...
}
```



When moveAccount is invoked, the reference inside *account2* is copied into *dstAccount* – **not** the actual object
this is like
dstAccount = account2;

Methods parameters: reference types

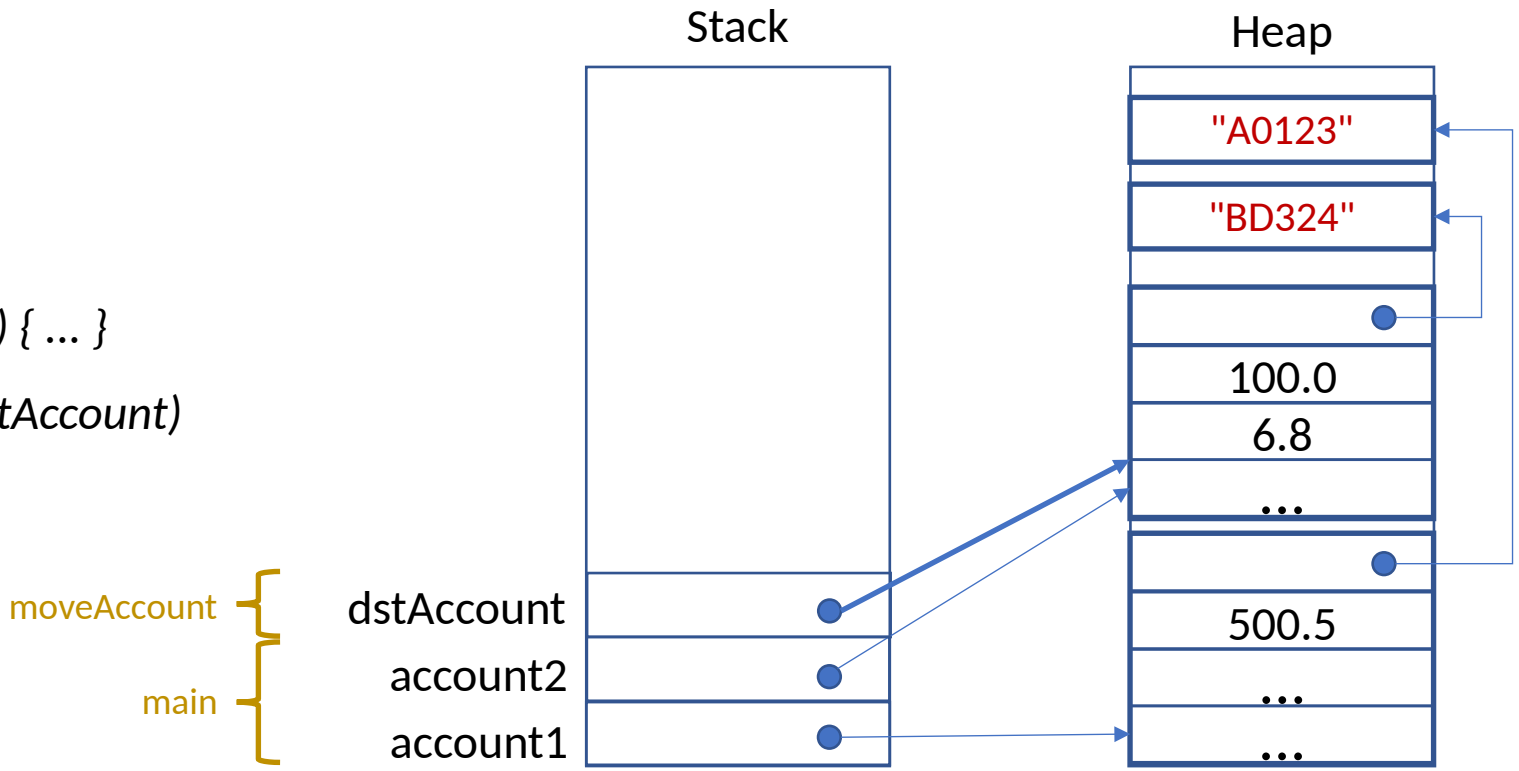
```
class BankAccount
{
    private String number;
    private double balance;

    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, 6.8);
        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, 6.8);
        account1.moveAccount(account2);
        ...
}
```



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

Methods parameters: reference types

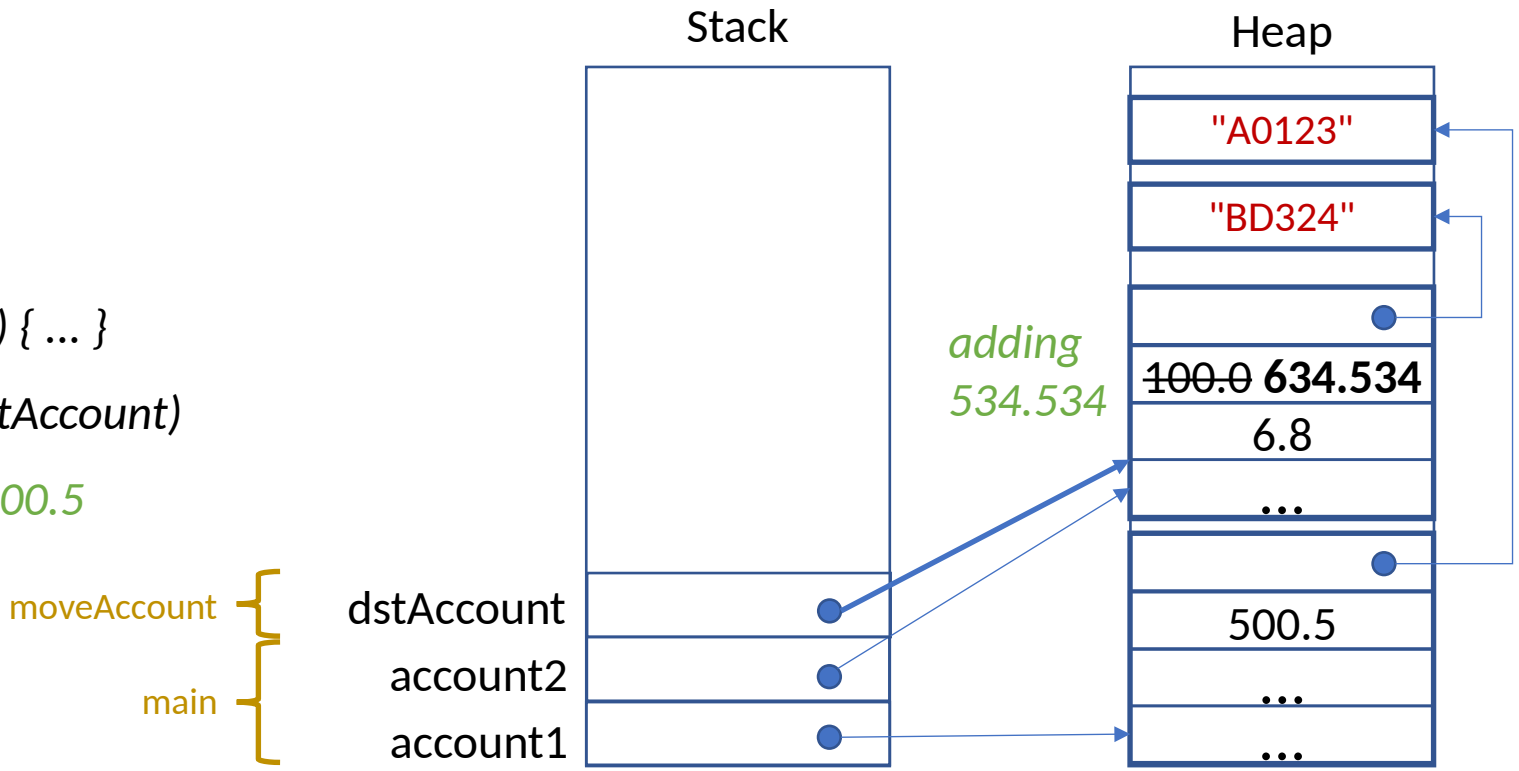
```
class BankAccount
{
    private String number;
    private double balance;

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

    public void moveAccount(SavingAccount dstAccount)
    {
        dstAccount.save(balance);
        close();
    }
}
```

```
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
        // e.g., amount=500.5 with 6.8% interest => 534.534
    }
}
```

```
class Program
{
    public static void main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 6.8);

        ...
}
```

Methods parameters: reference types

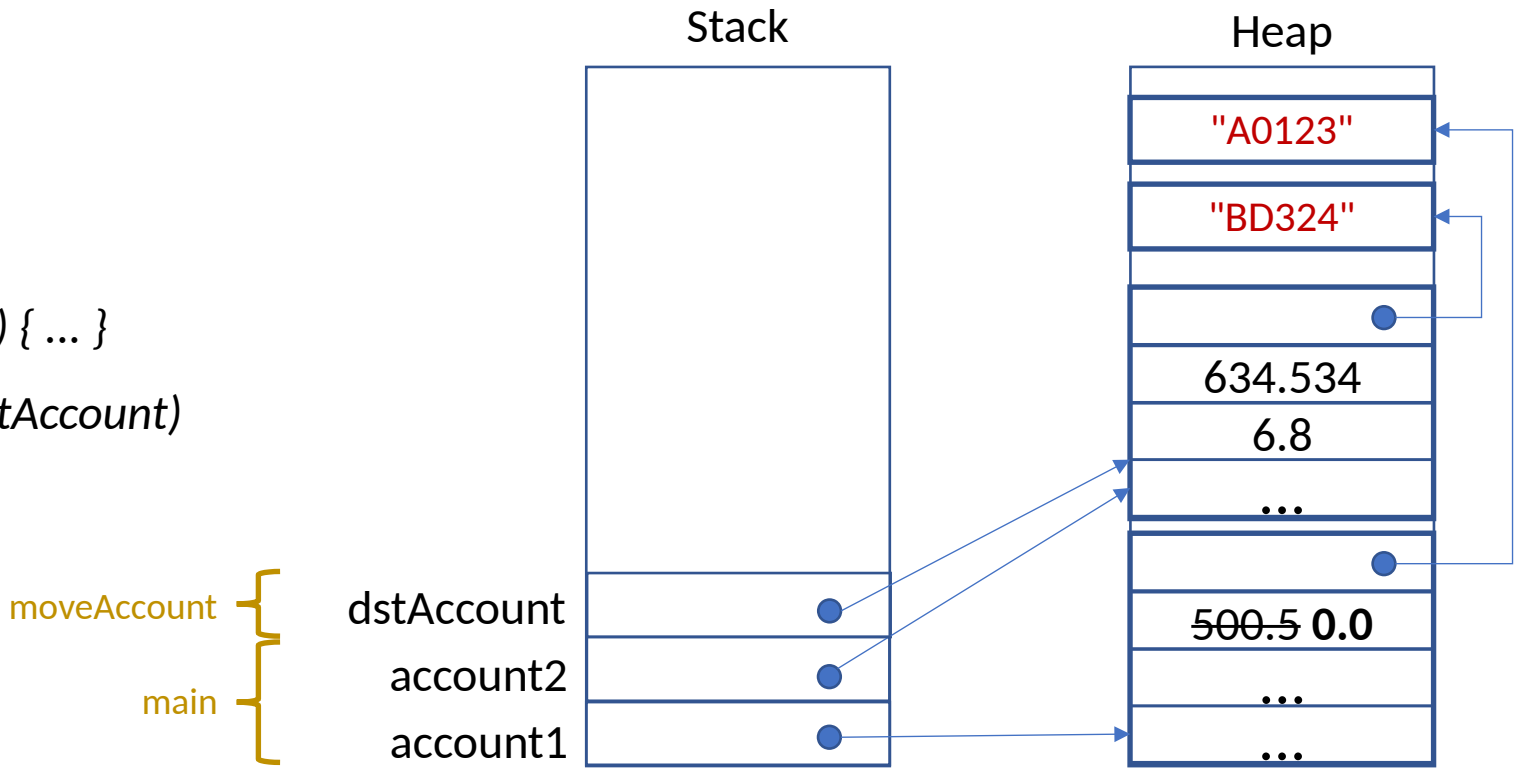
```
class BankAccount
{
    private String number;
    private double balance;

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

    public void moveAccount(SavingAccount dstAccount)
    {
        dstAccount.save(balance); // balance is 500.5
        close();
    }
}

class Program
{
    public static void main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 6.8);
        account1.moveAccount(account2);
        ...
}
```



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

Methods parameters: reference types

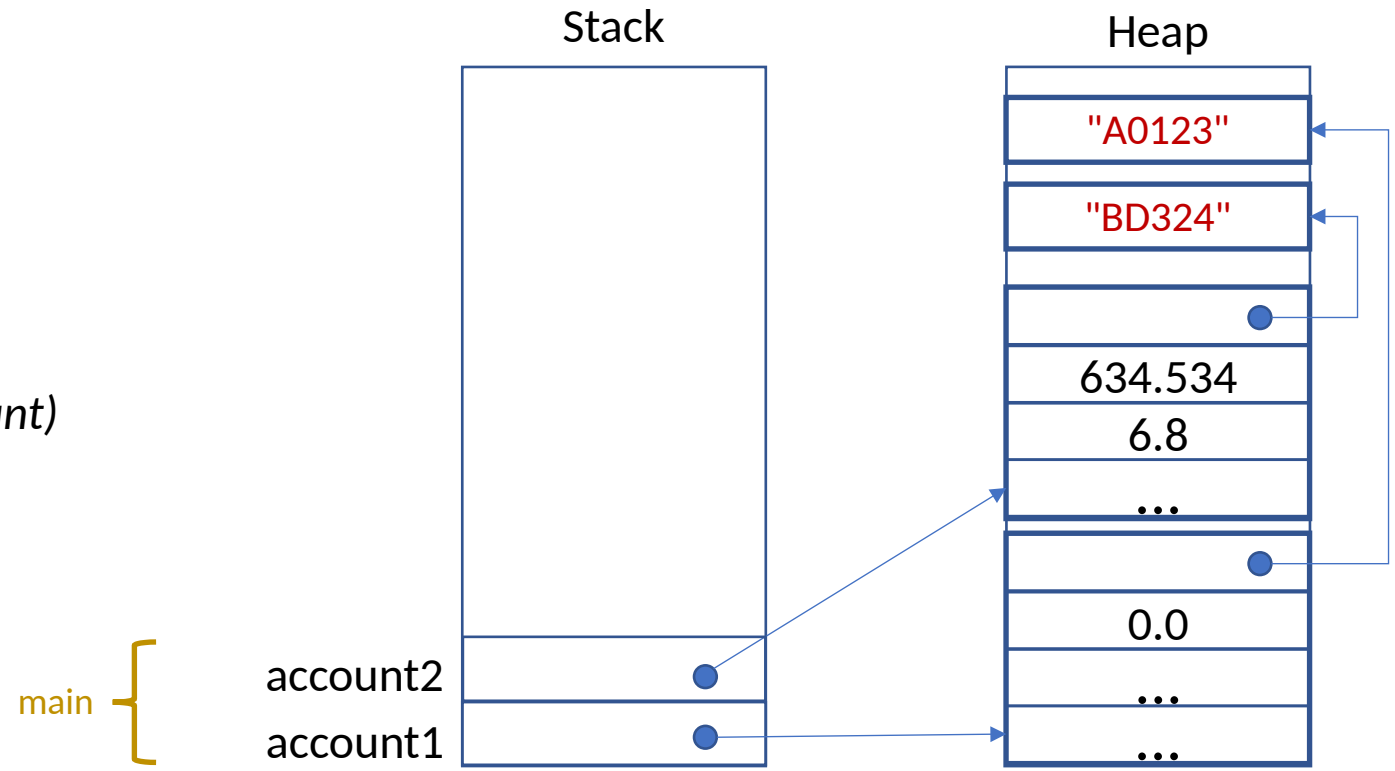
```
class BankAccount
{
    private String number;
    private double balance;

    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, 6.8);
        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, 6.8);

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



Any operations performed via *dstAccount* on the referenced object (*account2*) **persist after** the termination of *moveAccount*

Methods parameters: reference 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 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

Outline

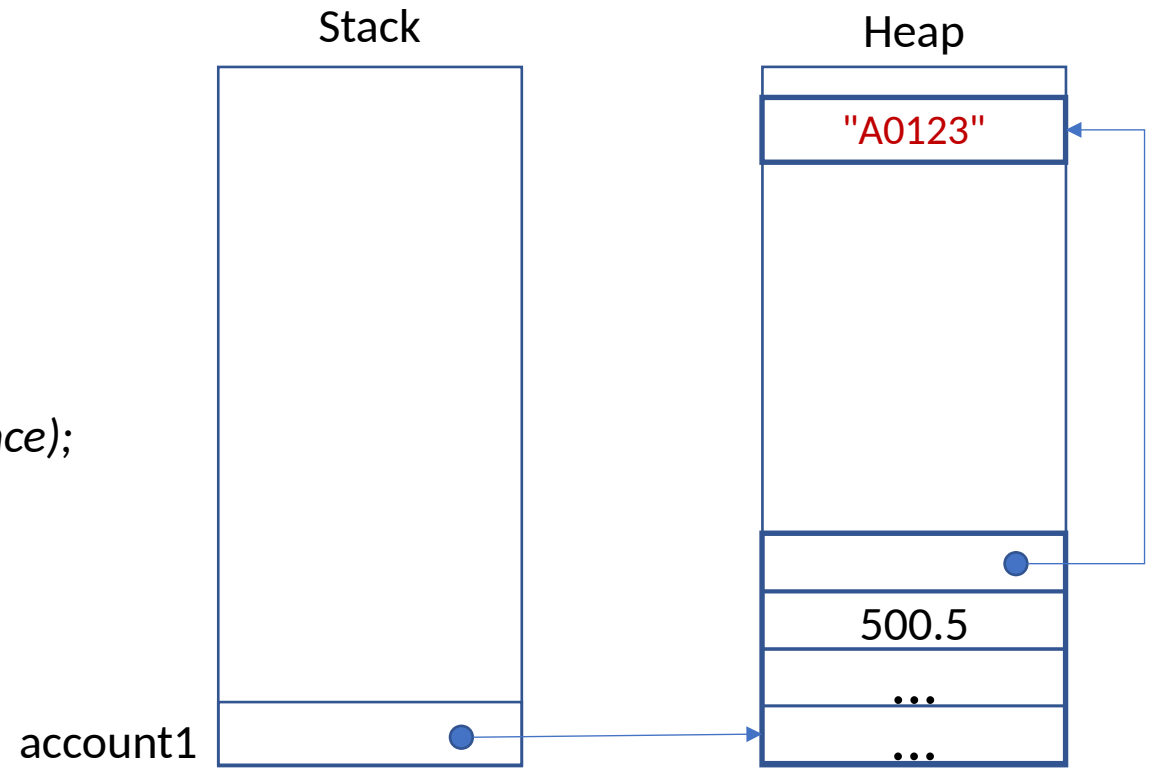
- More on Memory Management
  - Parameter Passing
  - **Objects lifetime**
  - 'this' keyword
- Static
  - Attributes
  - Methods
  - main method

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

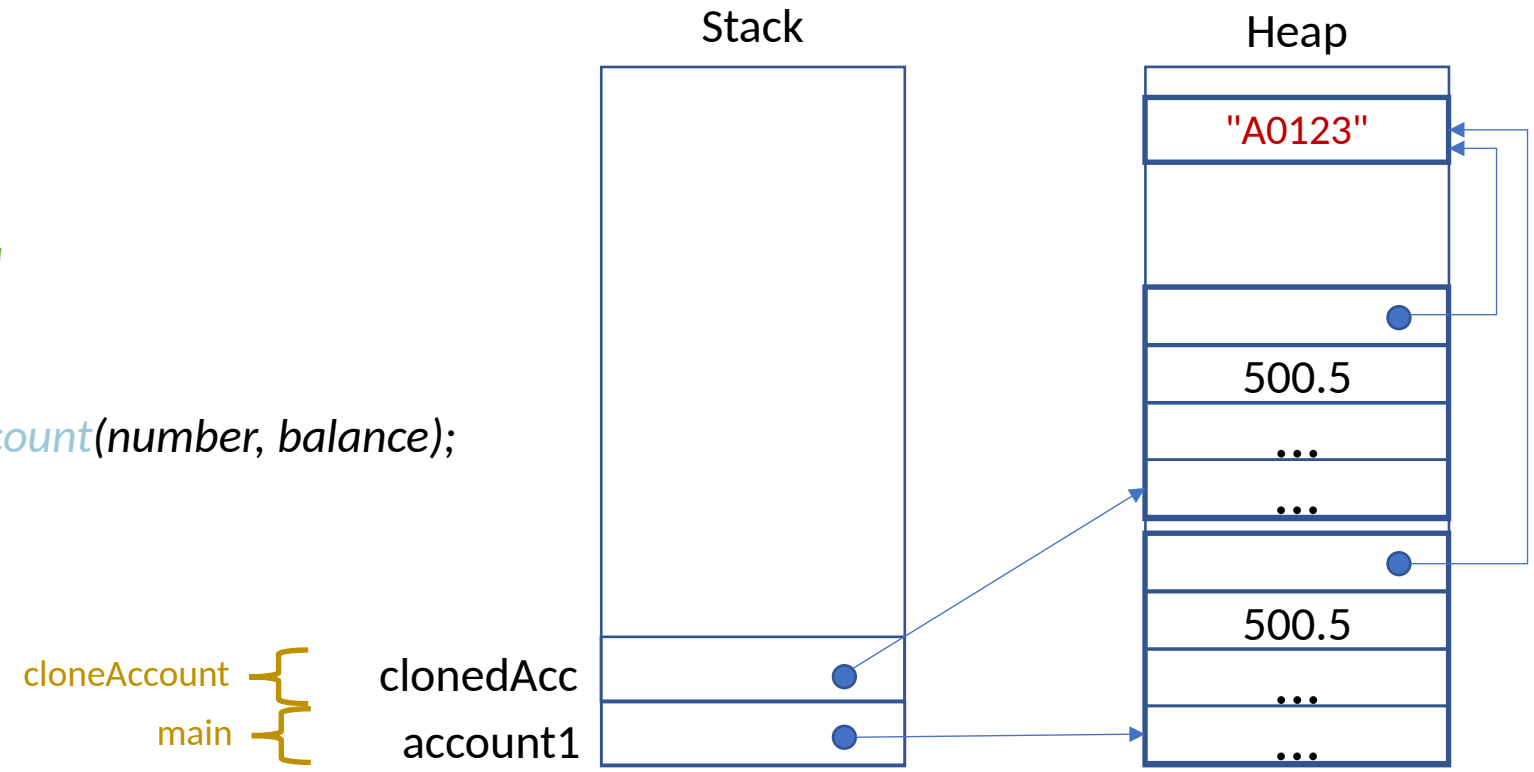


cloneAccount creates a (backup) copy of the account object on which is is invoked

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)

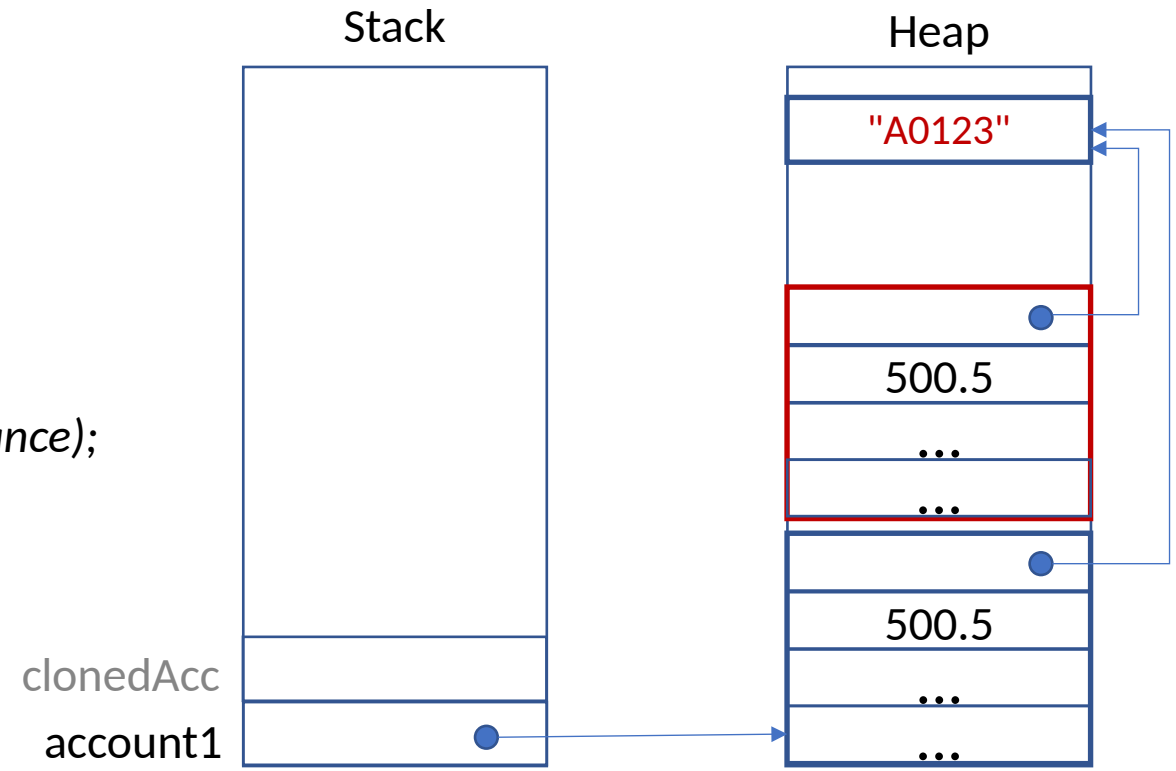
then // more things are performed

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();
        ...
}
```

main



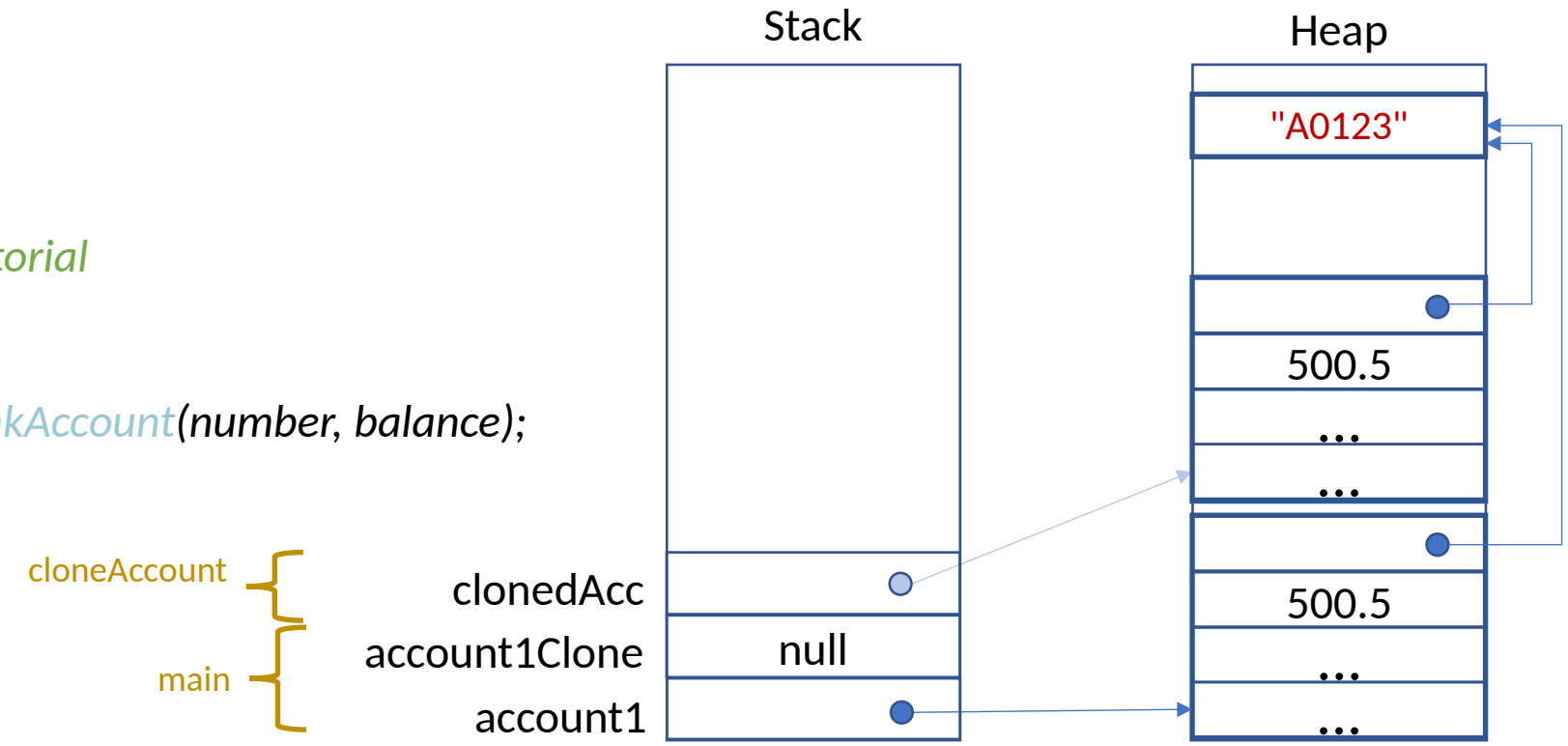
when *cloneAccount* terminates, its stack area is removed, so there will be no variables referring to the cloned object (could be garbage collected)

Question

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

Variables and Objects Lifetime

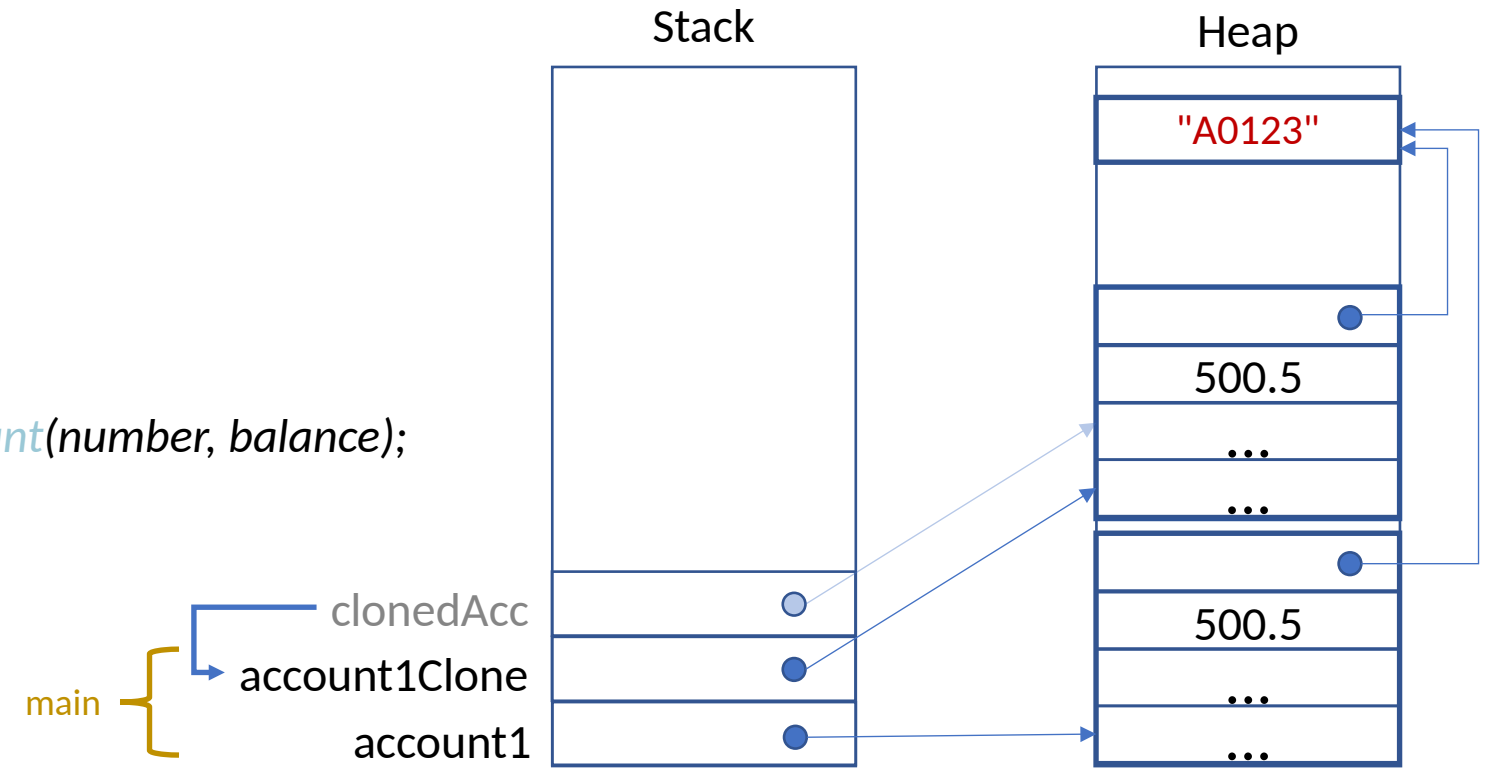
```
class BankAccount
{
    private String number;
    private double balance;

    // ... 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*)

The new object will now **continue to live** on the heap

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

Accessing attributes from a Method

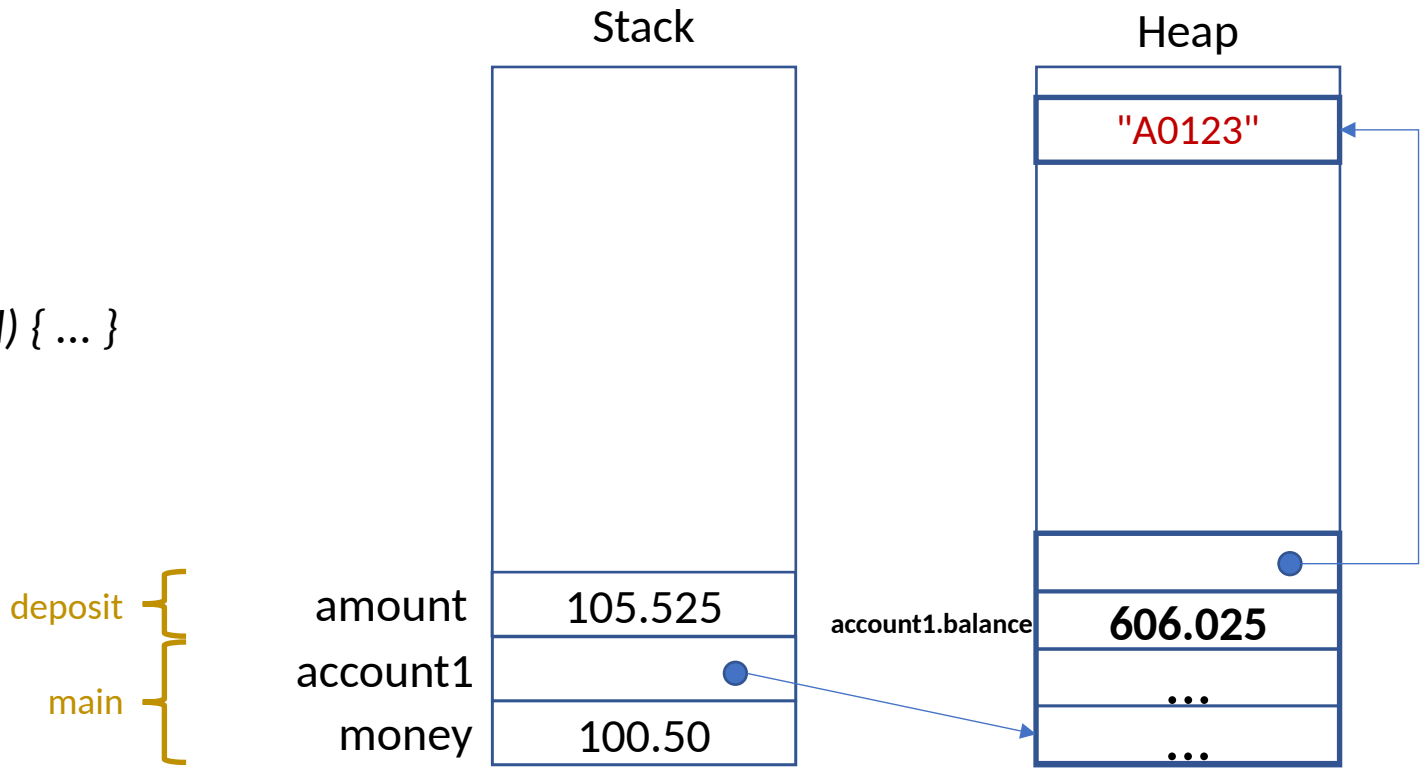
- We know that a method of a class can access the attributes defined in that class
- How is this implemented with *objects* created on the heap?
- How does a method know the *heap memory location* to access?

Outline

- More on Memory Management
  - Parameter Passing
  - Objects lifetime
  - **'this' keyword**
- Static
  - Attributes
  - Methods
  - main method

this keyword

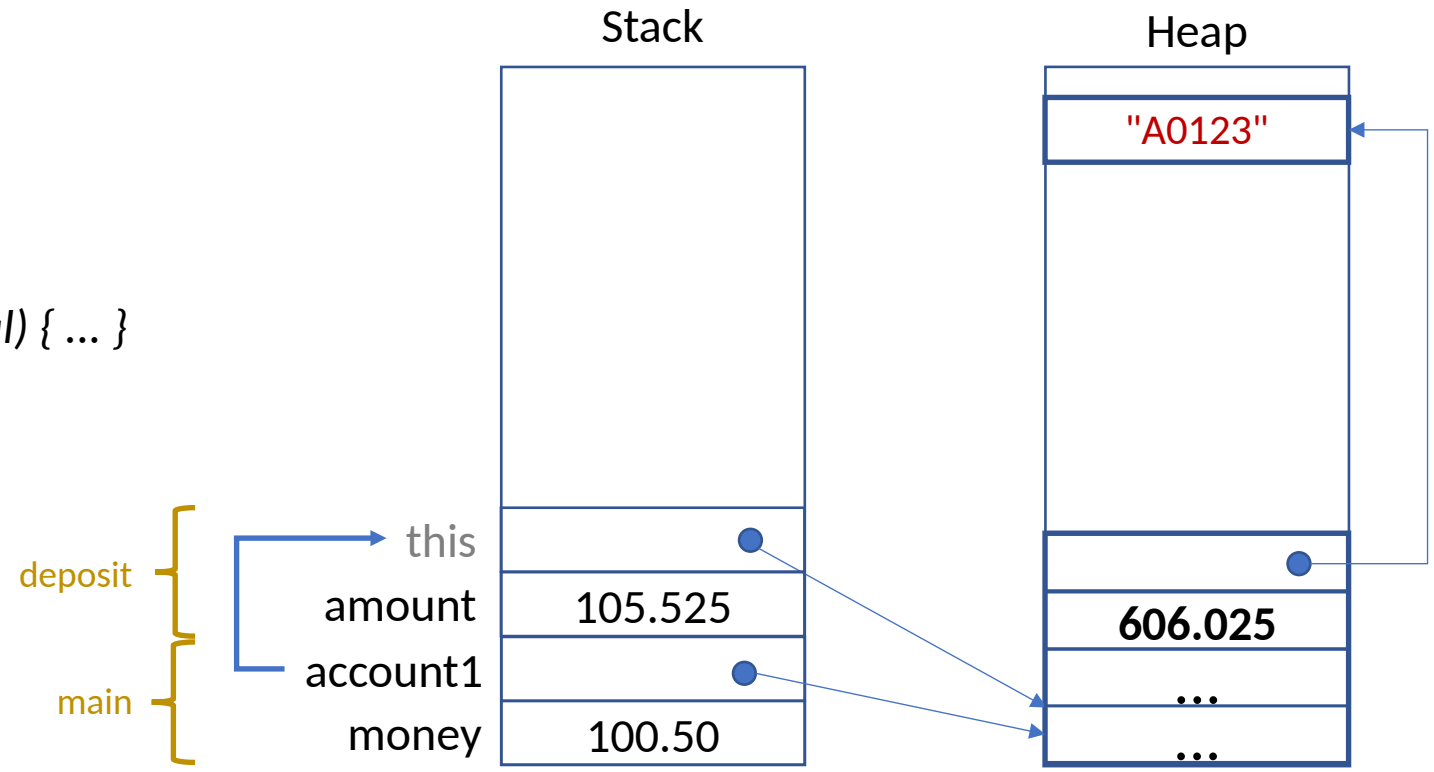
```
class BankAccount
{
    private String number;
    private double balance;

    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

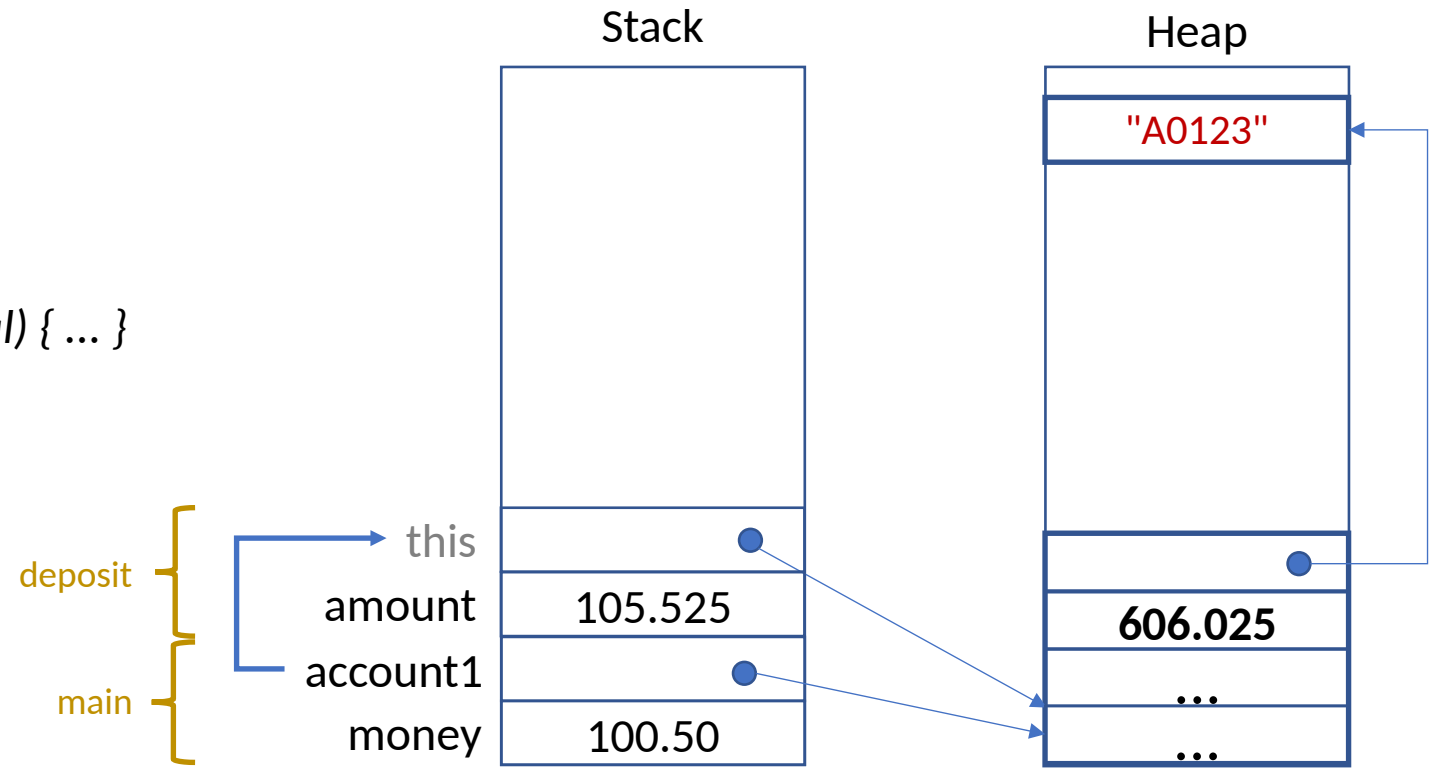
```
class BankAccount
{
    private String number;
    private double balance;

    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?

- Inside a method, to refer to the **object** on which that method is called

this keyword: how can be used?

- Inside a method, to refer to the object on which that method is called
- 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?

- Inside a method, to refer to the object on which that method is called
- To qualify **attributes** hidden by similar names

- 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**

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;
    }
}
```

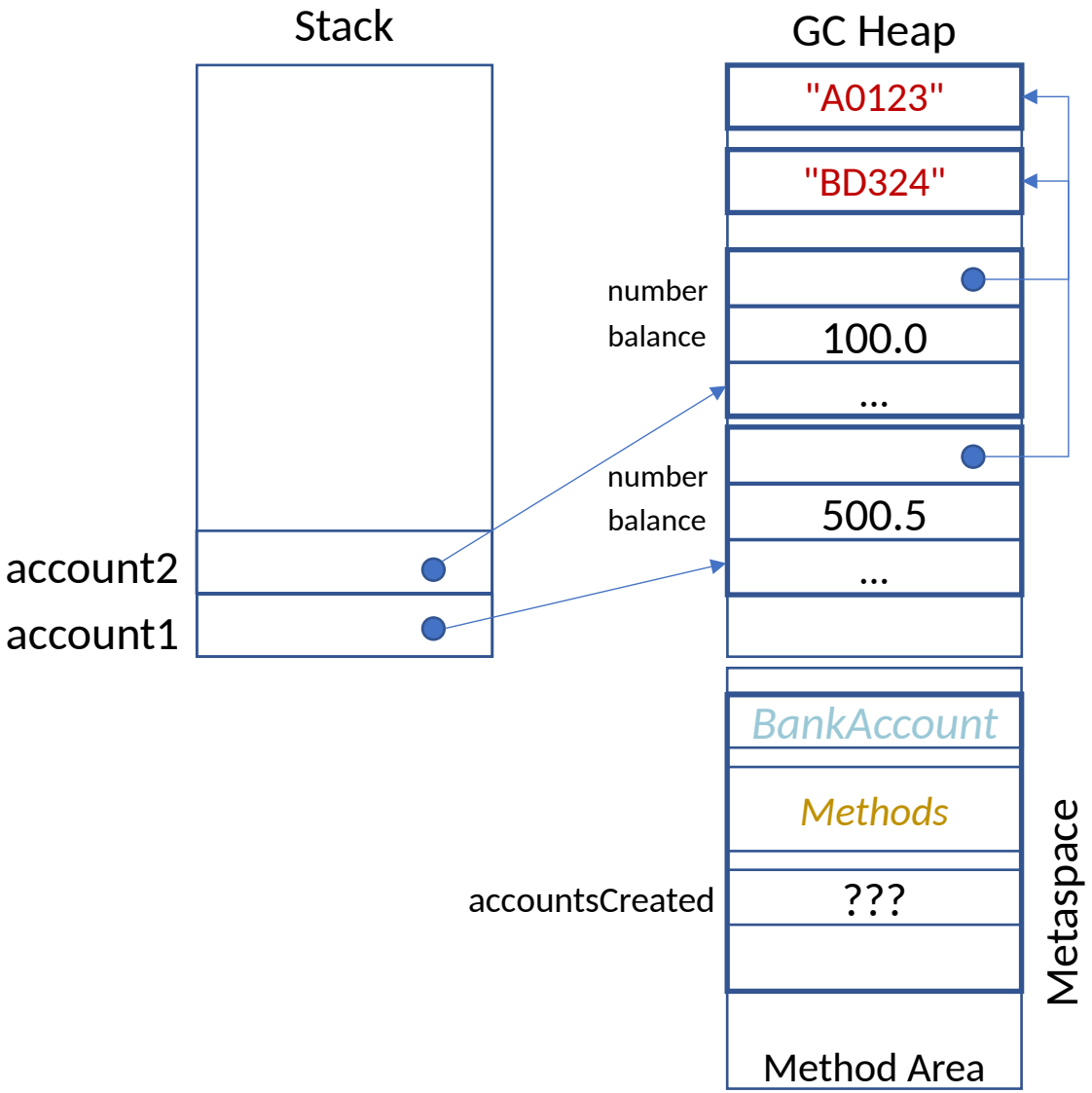
```
class BankAccount
{
    private String number;
    private double balance;

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

    public BankAccount(String num)
    {
        this(num, 0);
    }
}
```

Outline

- More on Memory Management
  - Parameter Passing
  - Objects lifetime
  - 'this' keyword
- Static
  - **Attributes**
  - Methods
  - main method

static attributes

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

    public BankAccount(String num, double bal) {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    public int getAccountsCreated () {
        return accountsCreated;
    }

class Program {
    public static void main() {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        System.out.println(account1.getAccountsCreated());

        account2 = new BankAccount("BD324", 100.0);
        System.out.println(account2.getAccountsCreated());

        System.out.println(account1.getAccountsCreated());
    }
}
```



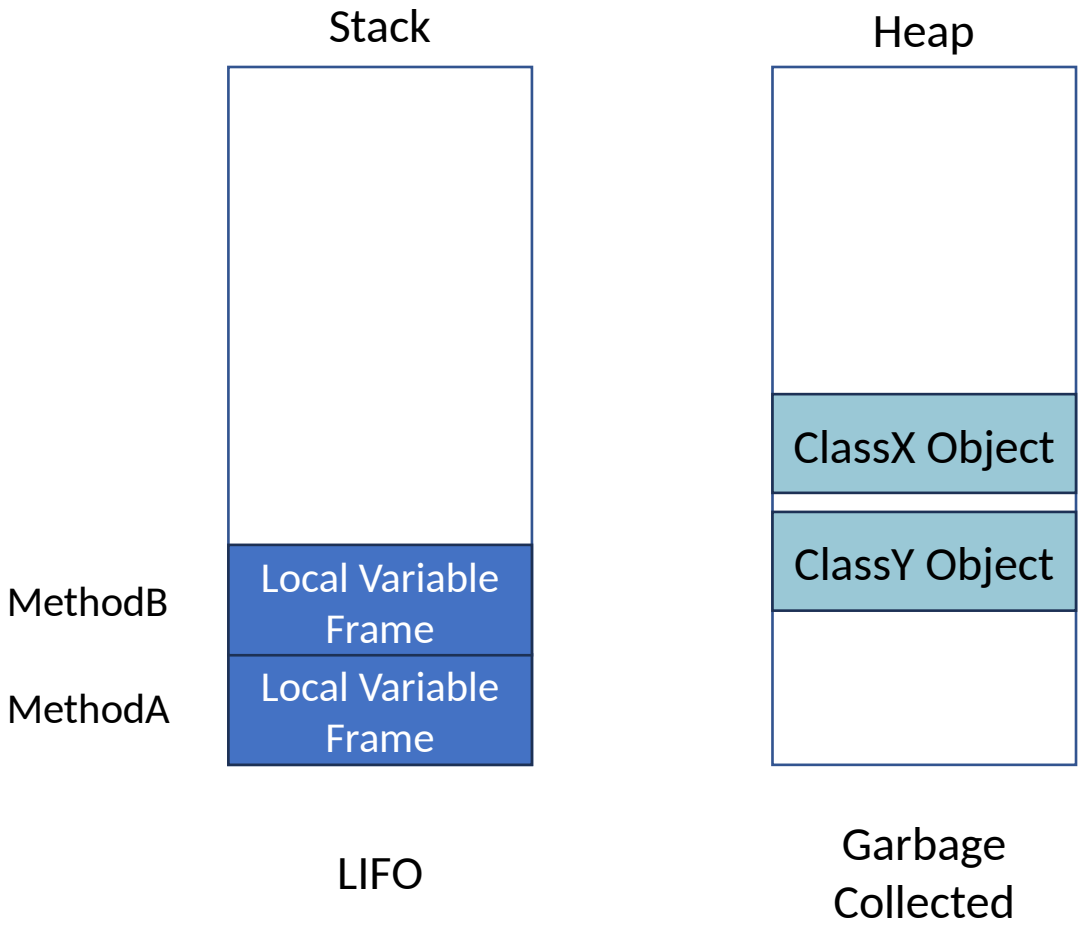
Question

- What is the output of the previous program?

Answer on PollEveryWhere

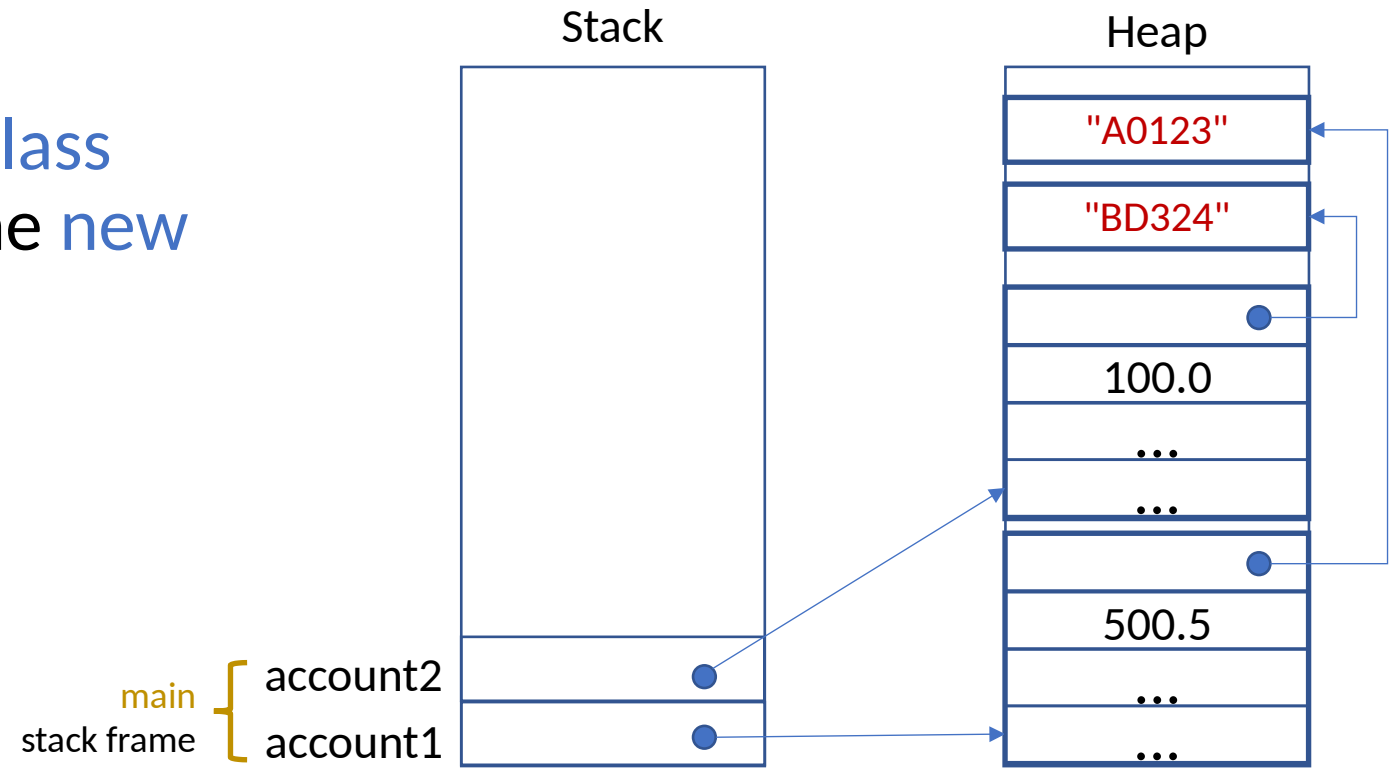
https://pollev.com/francescotusa

Memory Management



Object instances

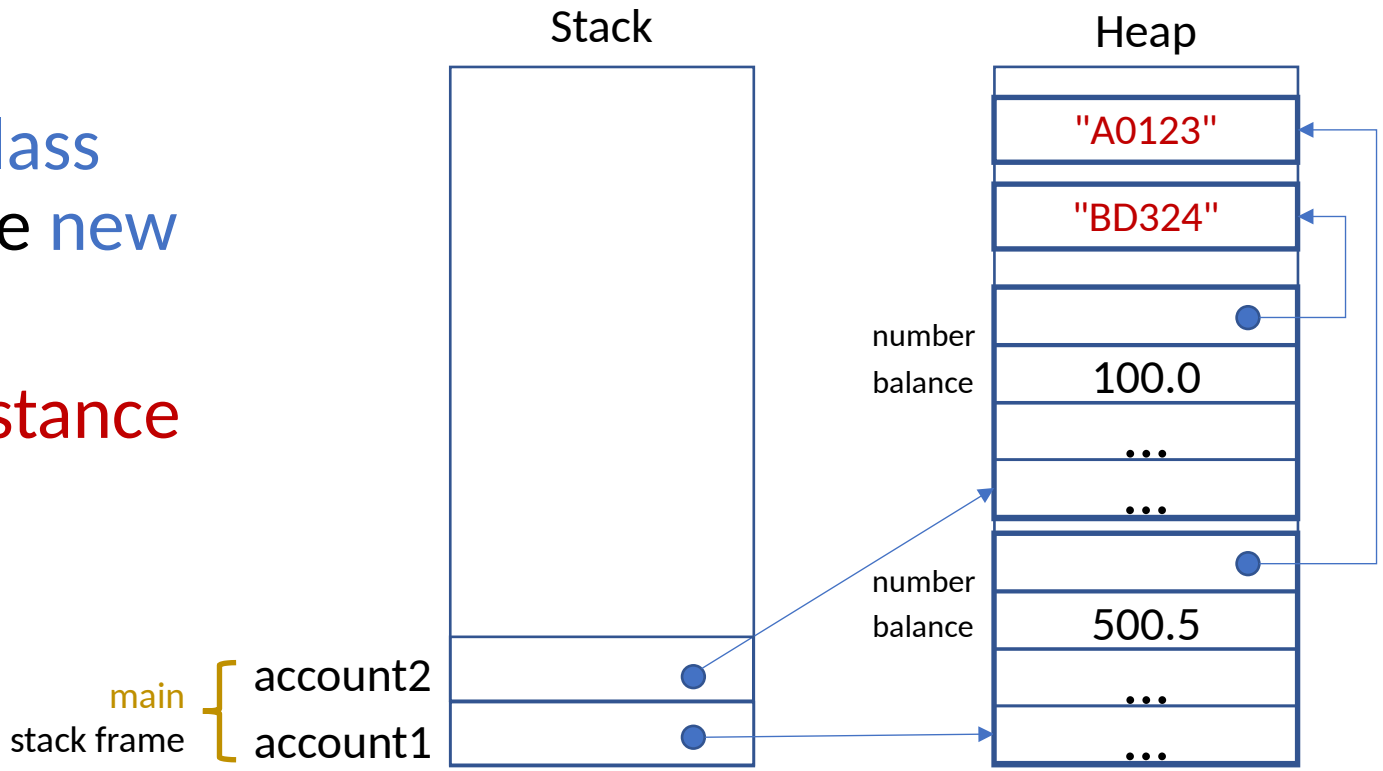
- Object *instances* of a given class are created at *runtime* via the new operator




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

        ...
}
```

Object instances

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




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

        ...
}
```

static keyword

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

static attributes

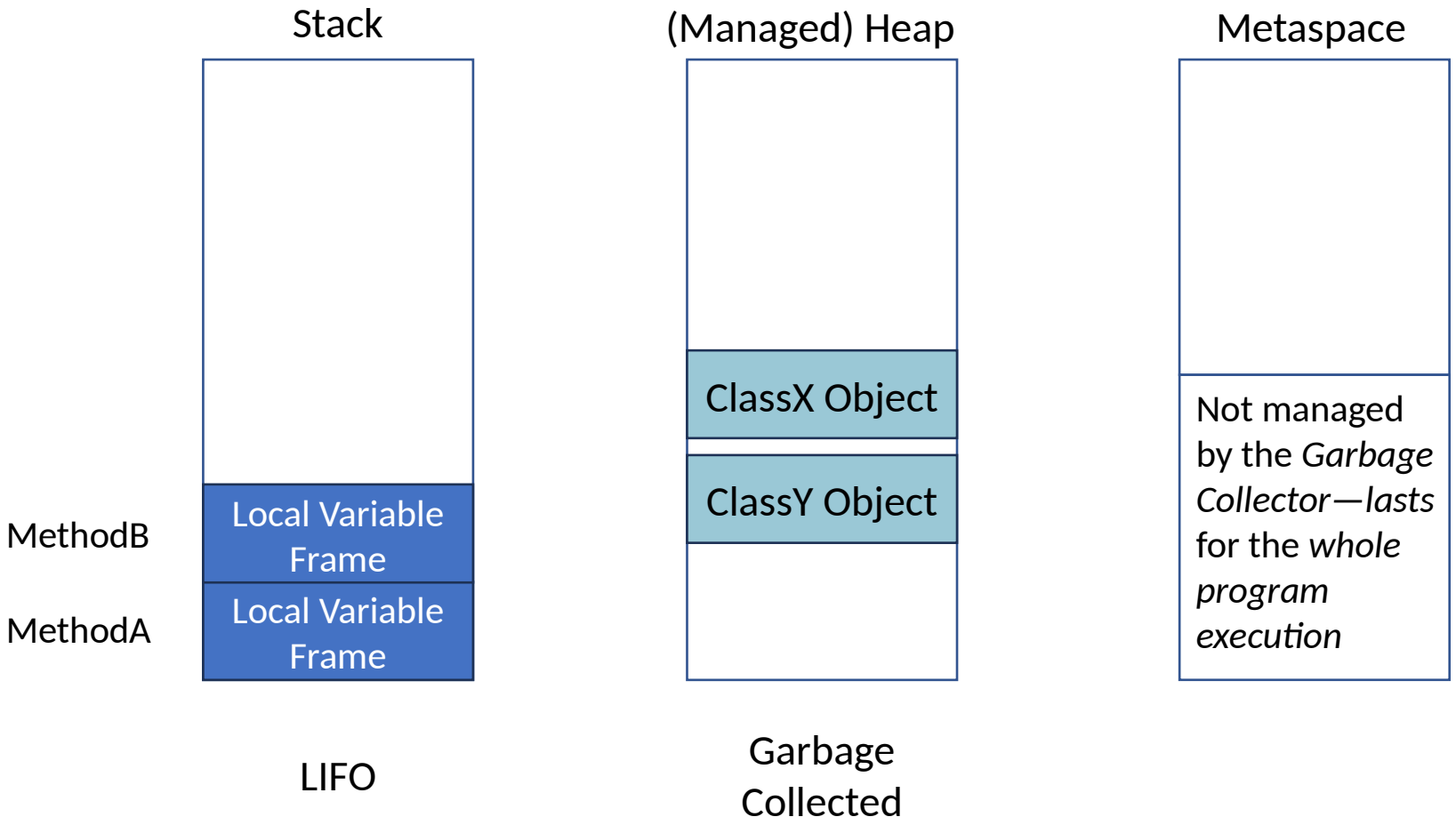
- A static *attribute* of a class is shared by *all the instances* of that class
- It *exists* regardless of objects of that class being instantiated

- Where are static attributes stored?

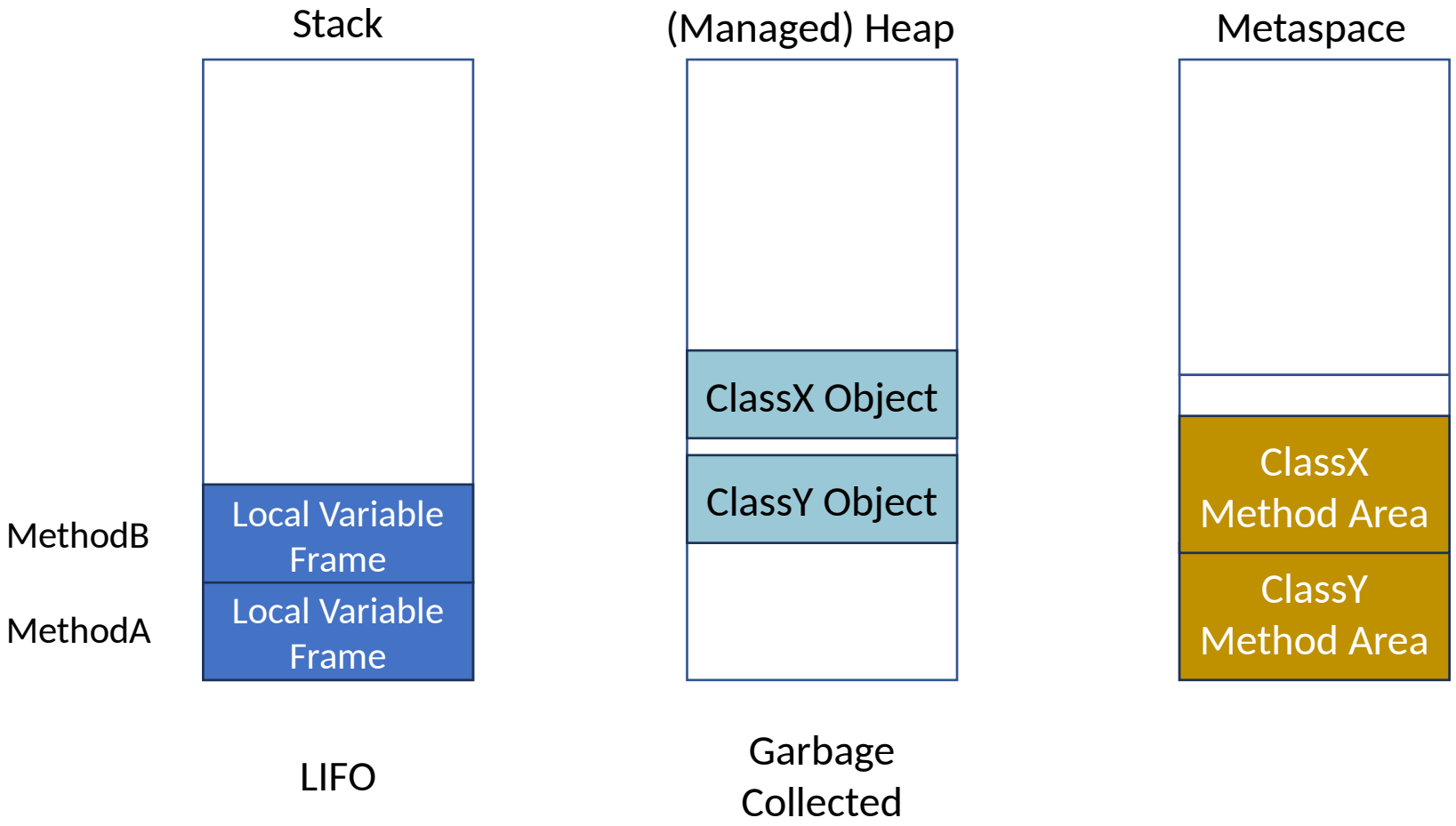
Where are the methods stored?

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

Where are the methods stored?



Where are the methods stored?



The Method Area

- Contains information (metadata) about classes *common* among objects
- Stores the *bytecode* of the classes with the methods definition loaded from the *.class* files

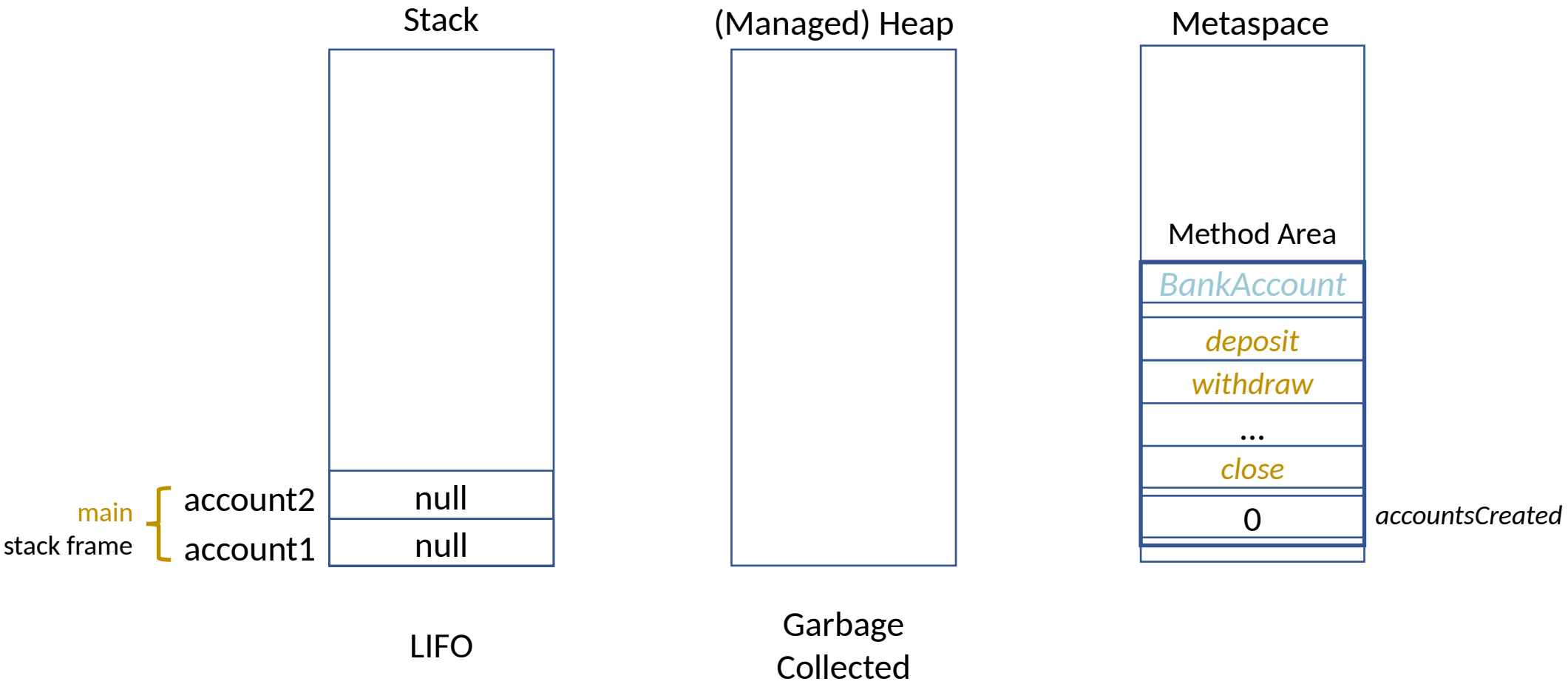
- How is it related to static attributes?

static attributes

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

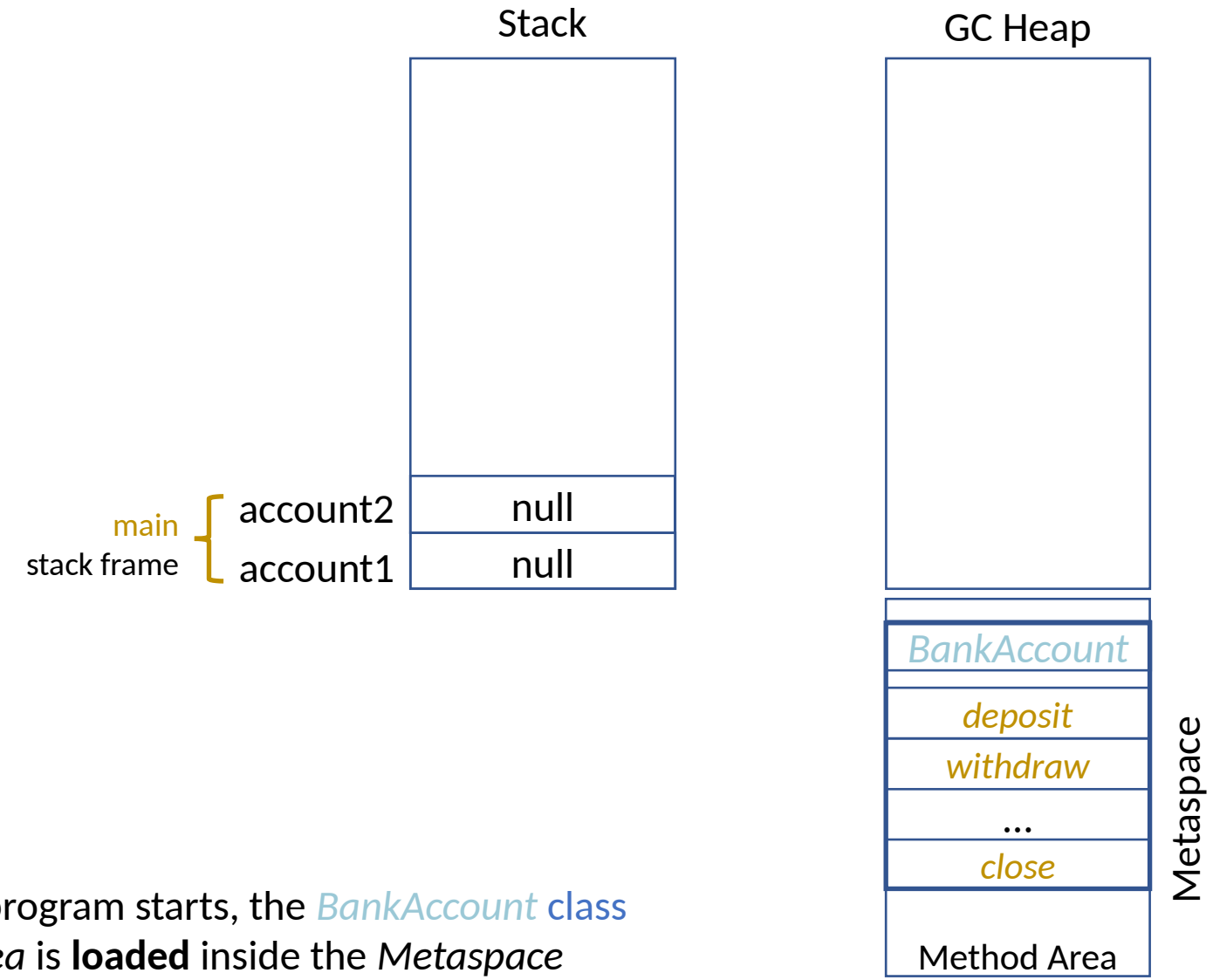
- **Static attributes** are in the *Method Area* of that class in the **Metaspace**

Where are the methods stored?



static attributes

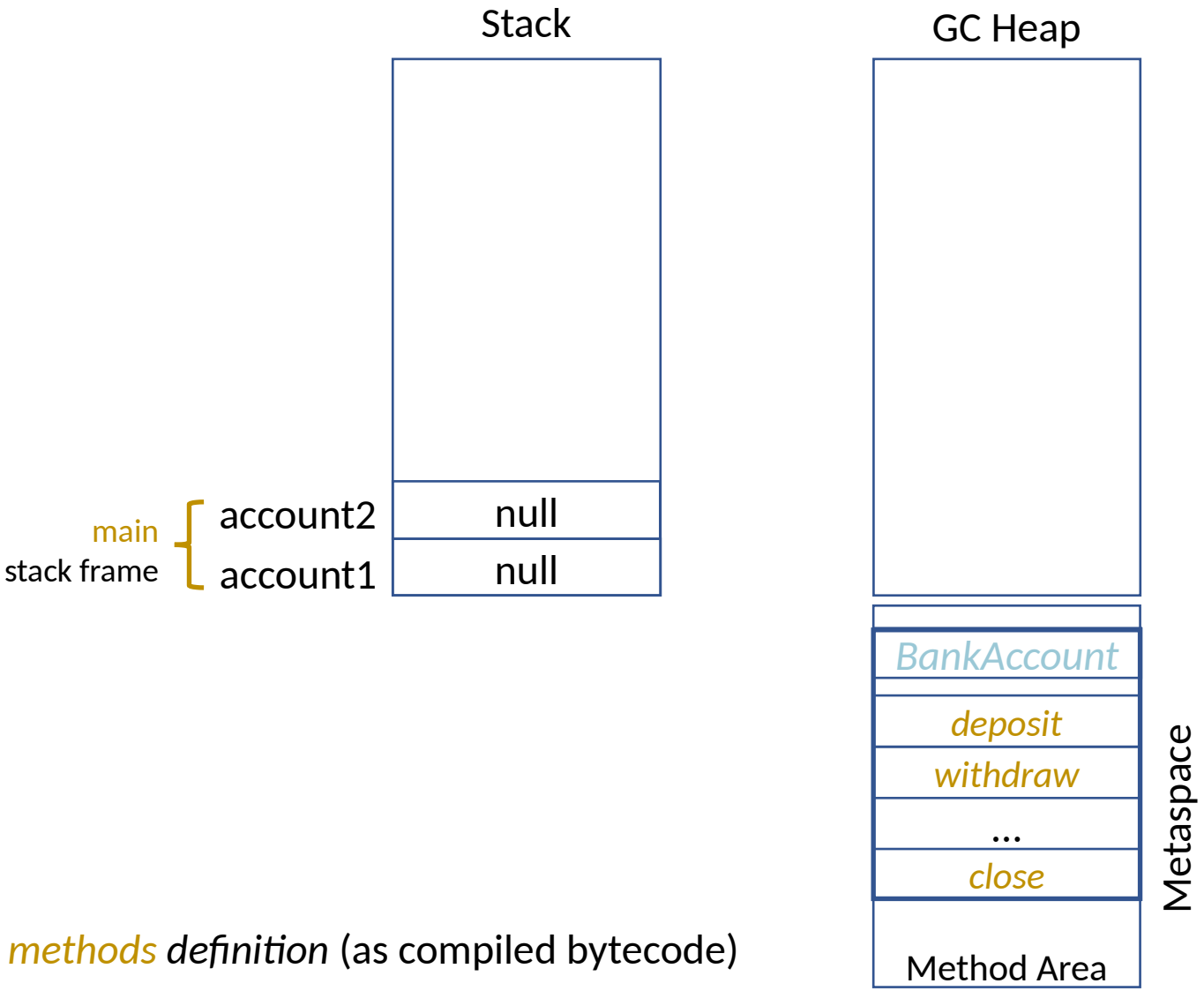
```
class BankAccount
{
    private String number;
    private double balance;
    private static int accountsCreated = 0;

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

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


    }
}
```



When the program starts, the *BankAccount* class *Method Area* is **loaded** inside the *Metaspace*

static attributes

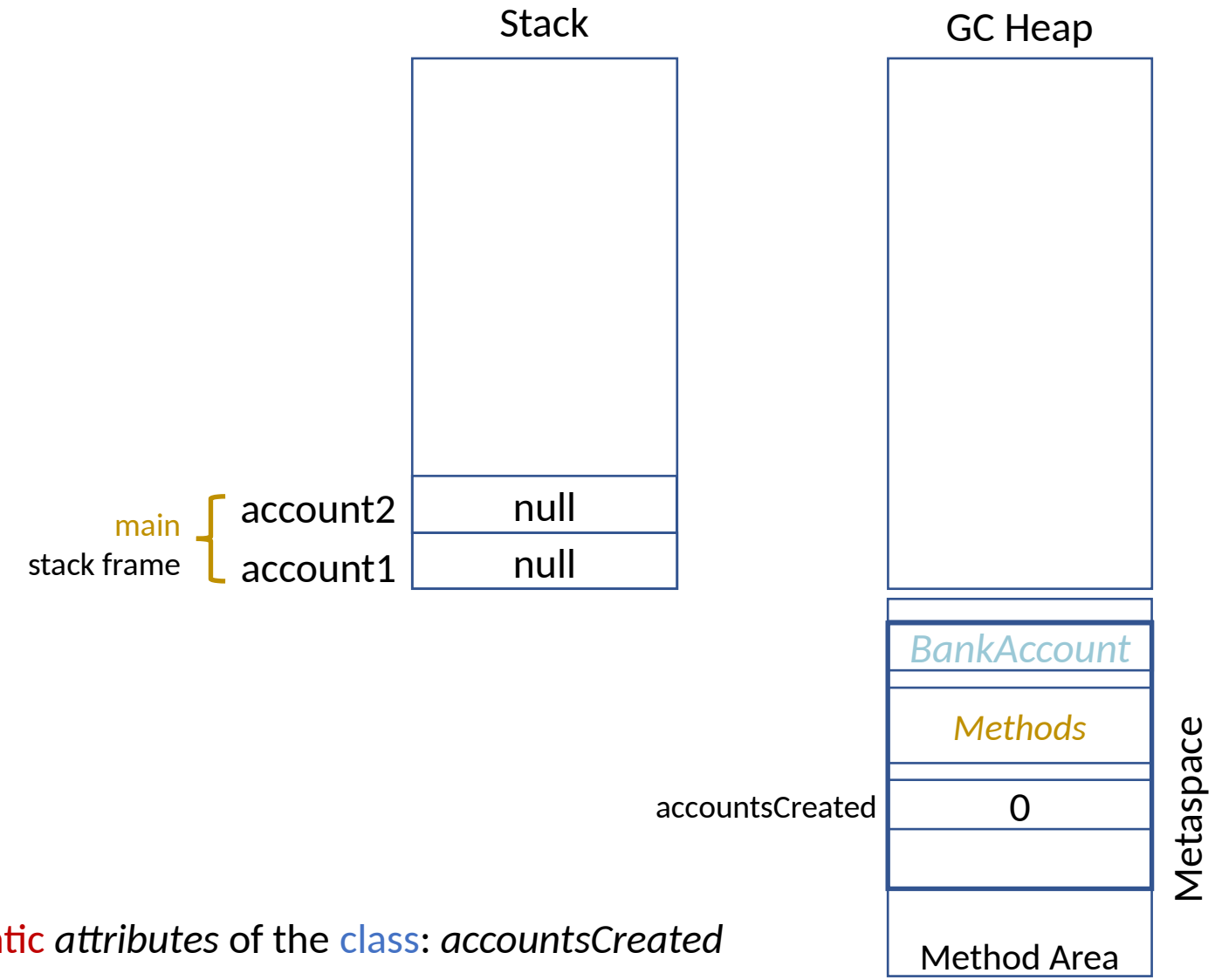
```
class BankAccount
{
    private String number;
    private double balance;
    private static int accountsCreated = 0;

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

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


    }
}
```



This includes *methods definition* (as compiled bytecode)

static attributes

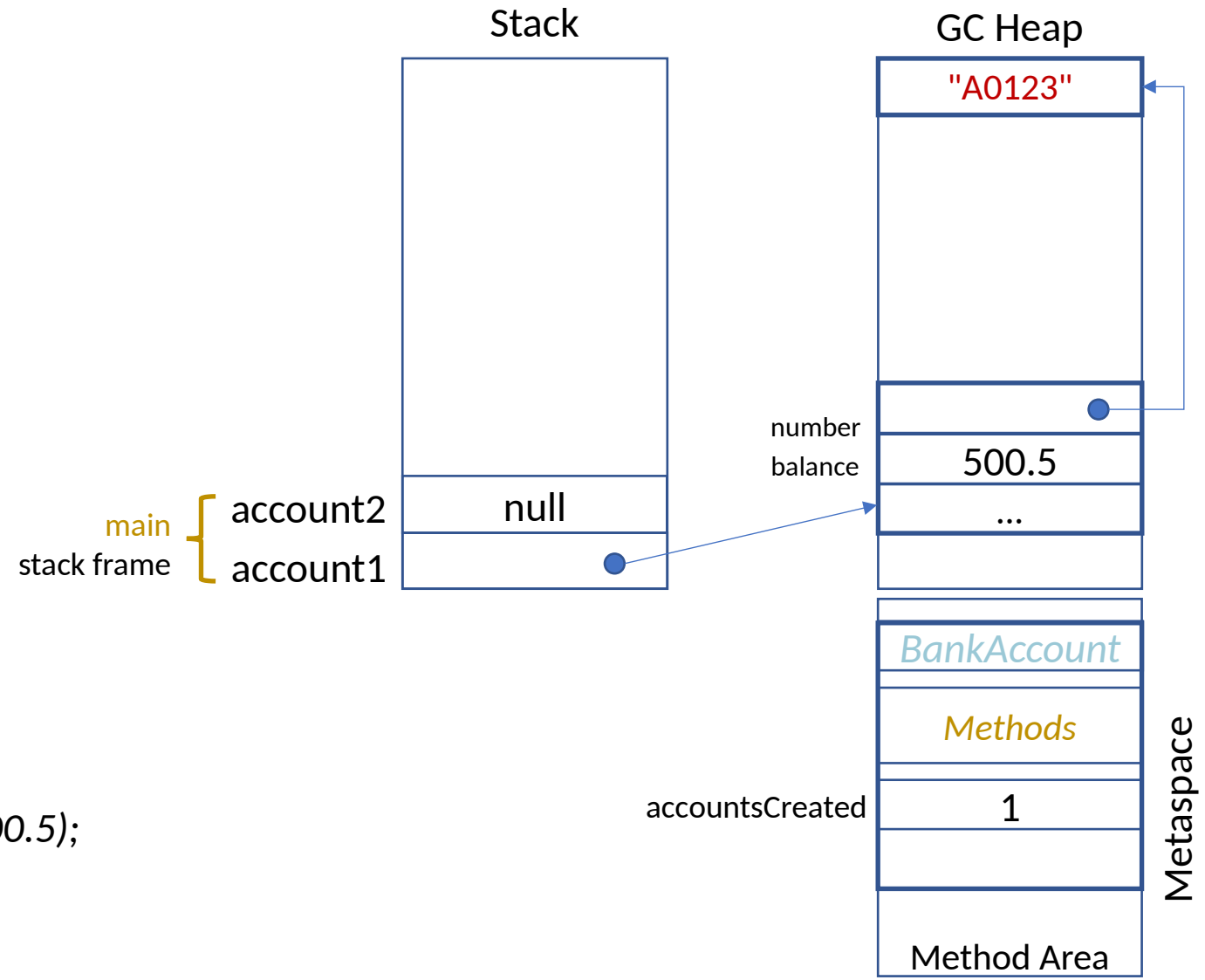
```
class BankAccount
{
    private String number;
    private double balance;
    private static int accountsCreated = 0;

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

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


    }
}
```



And the static *attributes* of the class: *accountsCreated*

static attributes

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

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

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

    }
}
```

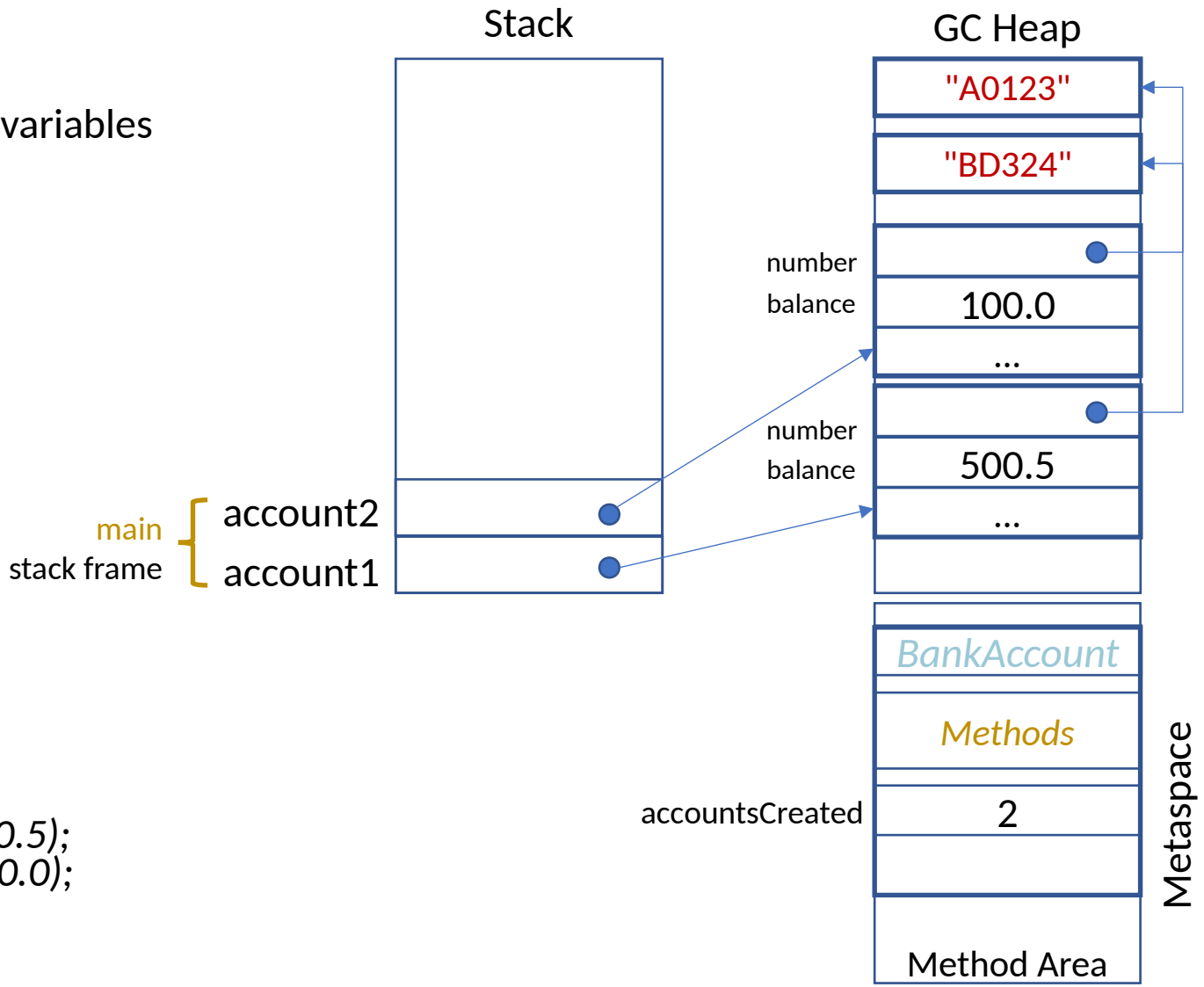


static attributes

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

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

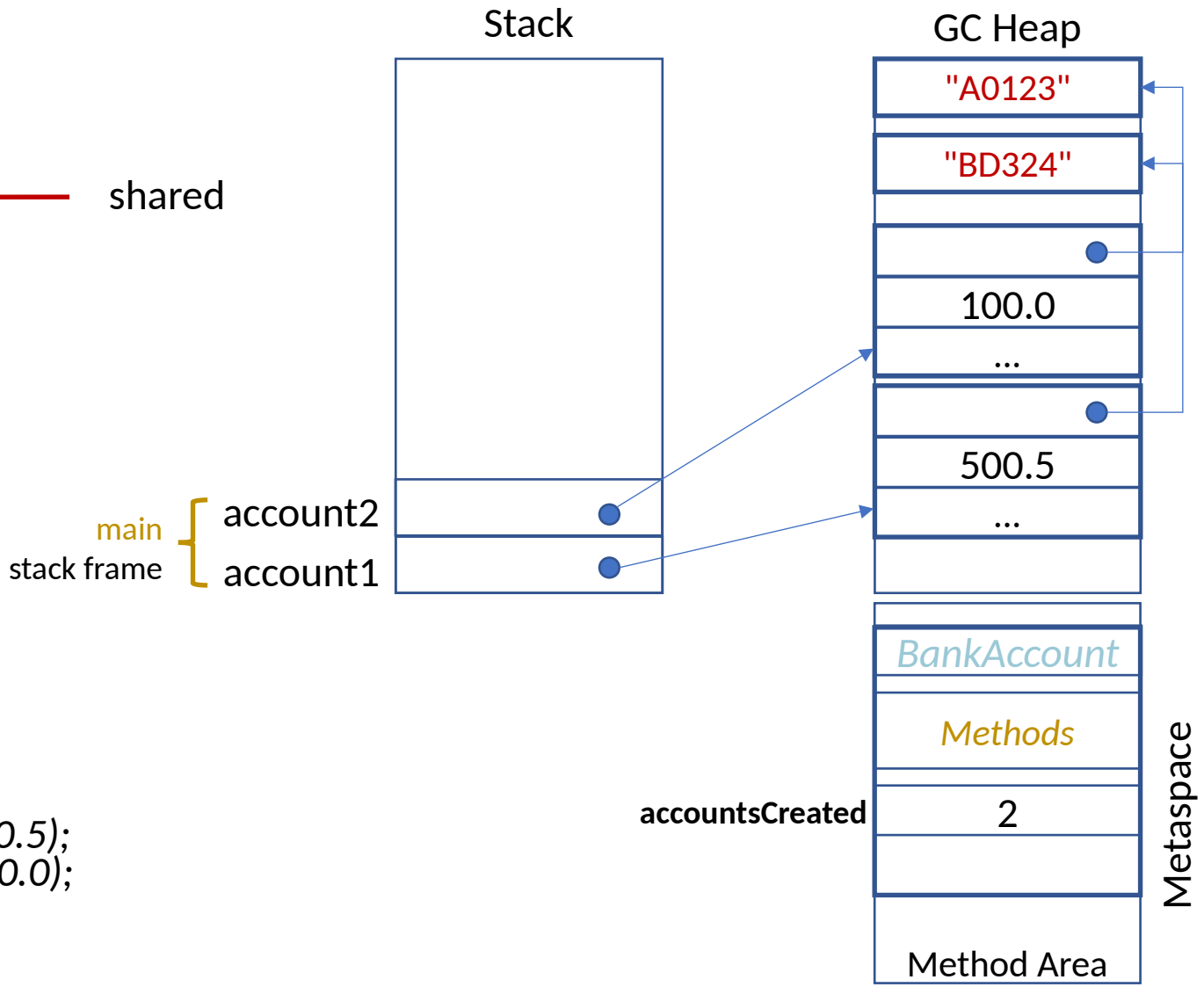
instance variables

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



static attributes

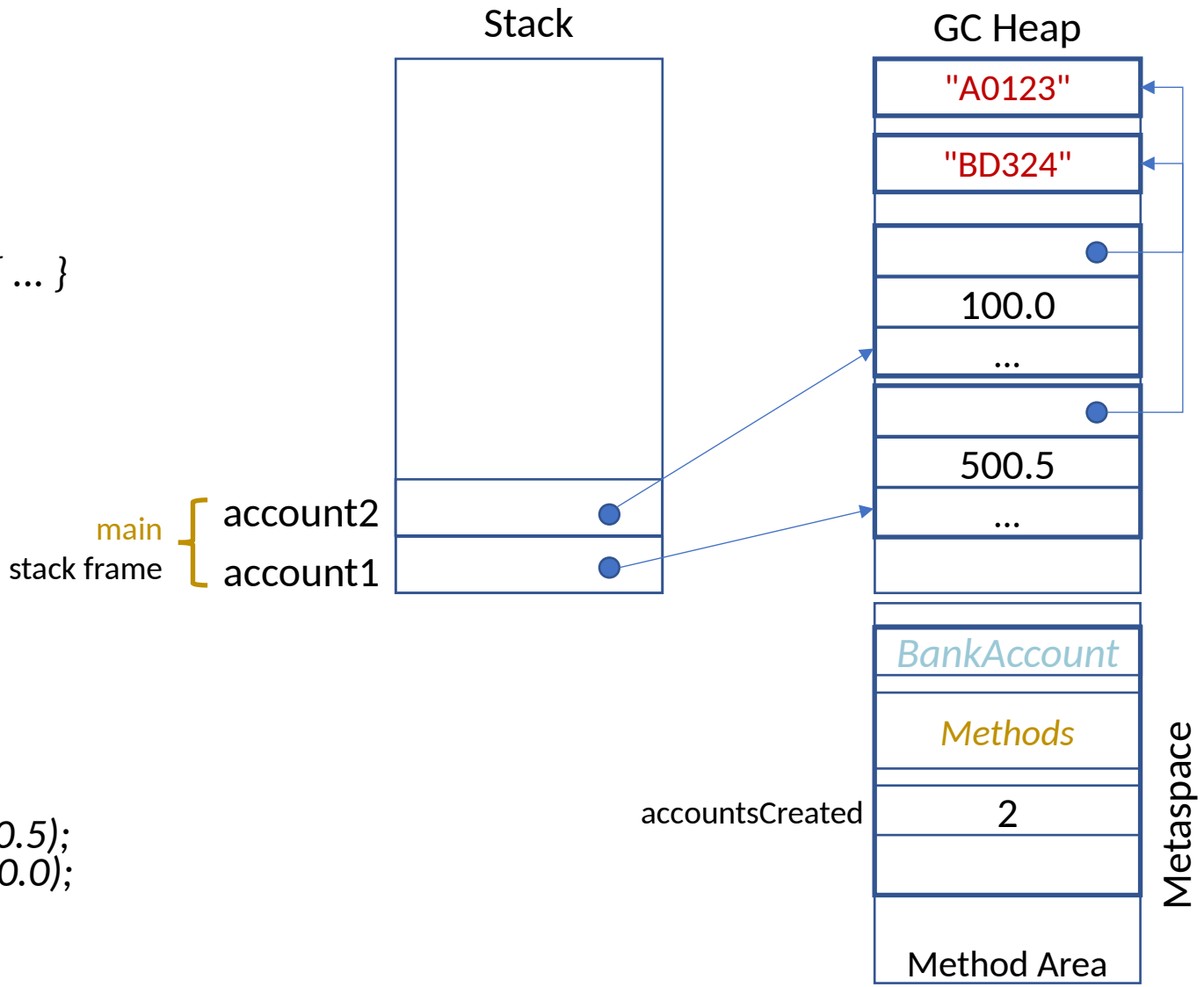
```
class BankAccount
{
    private String number;
    private double balance;
    private static int accountsCreated = 0;

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

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



static attributes

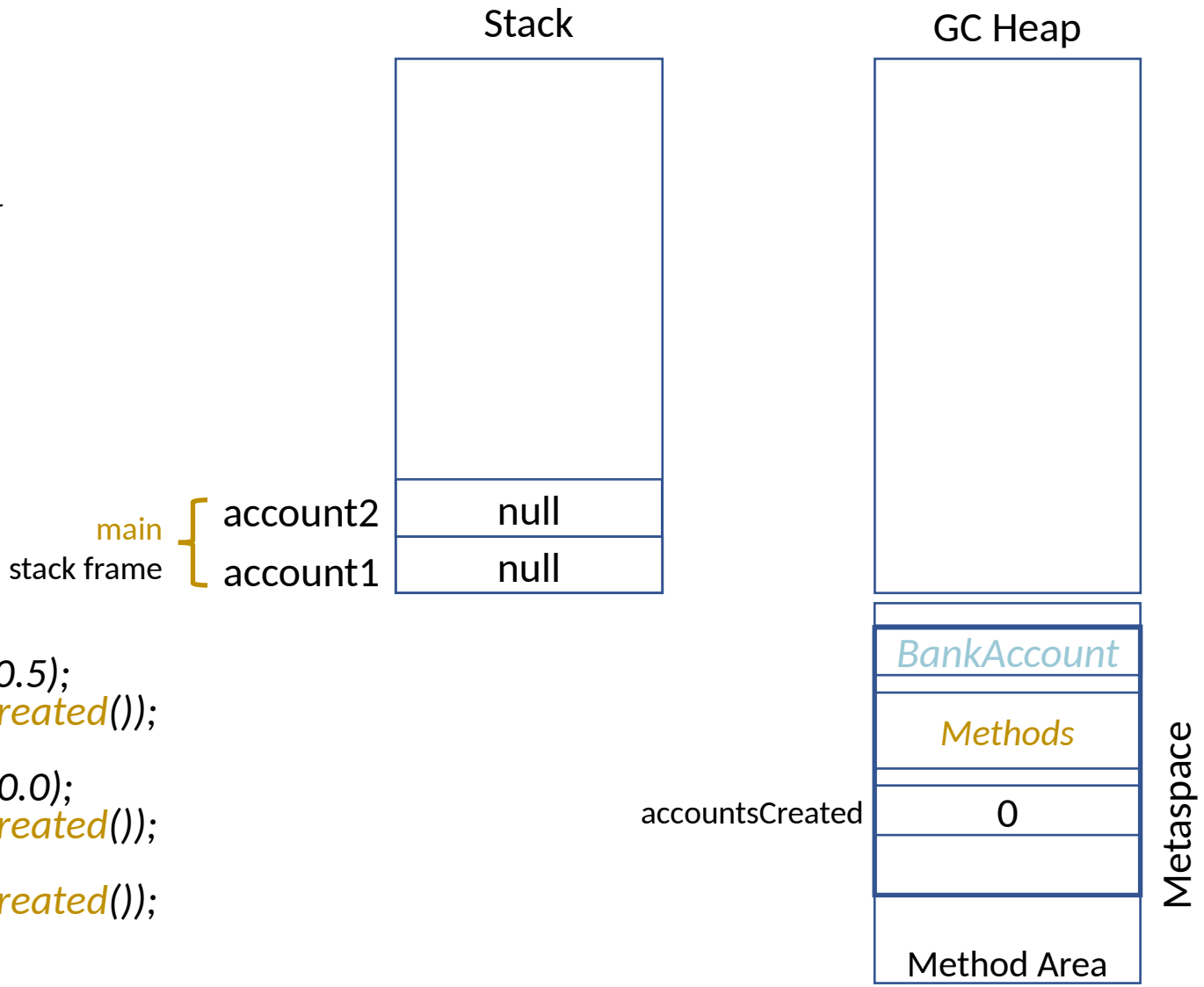
```
class BankAccount
{
    private String number;
    private double balance;
    private static int accountsCreated = 0;

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

    public int getAccountsCreated ()
    {
        return accountsCreated;
    }
}
```

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



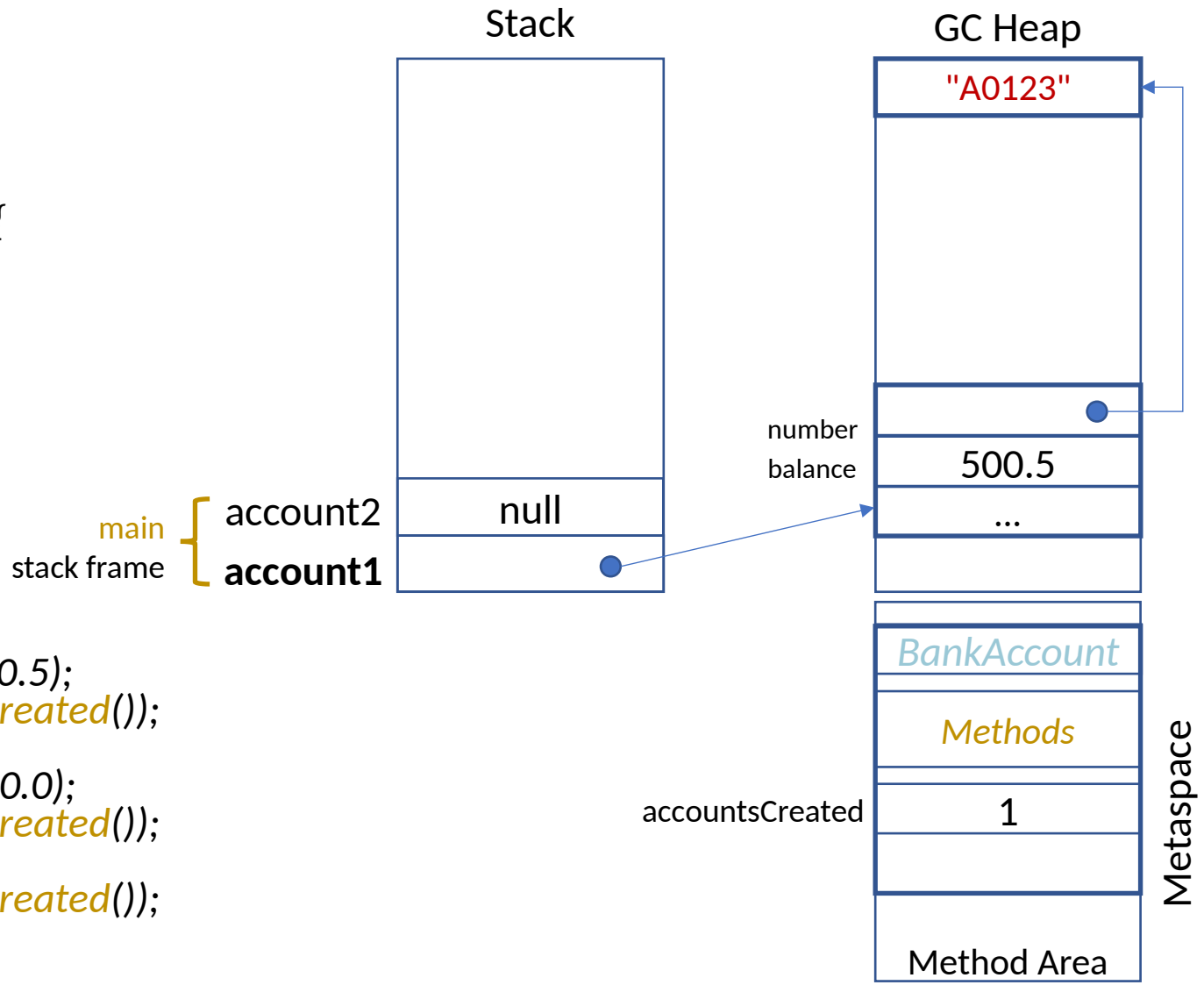
Answer

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

    public BankAccount(String num, double bal) {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    public int getAccountsCreated () {
        return accountsCreated;
    }

class Program {
    public static void main() {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        System.out.println(account1.getAccountsCreated());

        account2 = new BankAccount("BD324", 100.0);
        System.out.println(account2.getAccountsCreated());

        System.out.println(account1.getAccountsCreated());
    }
}
```



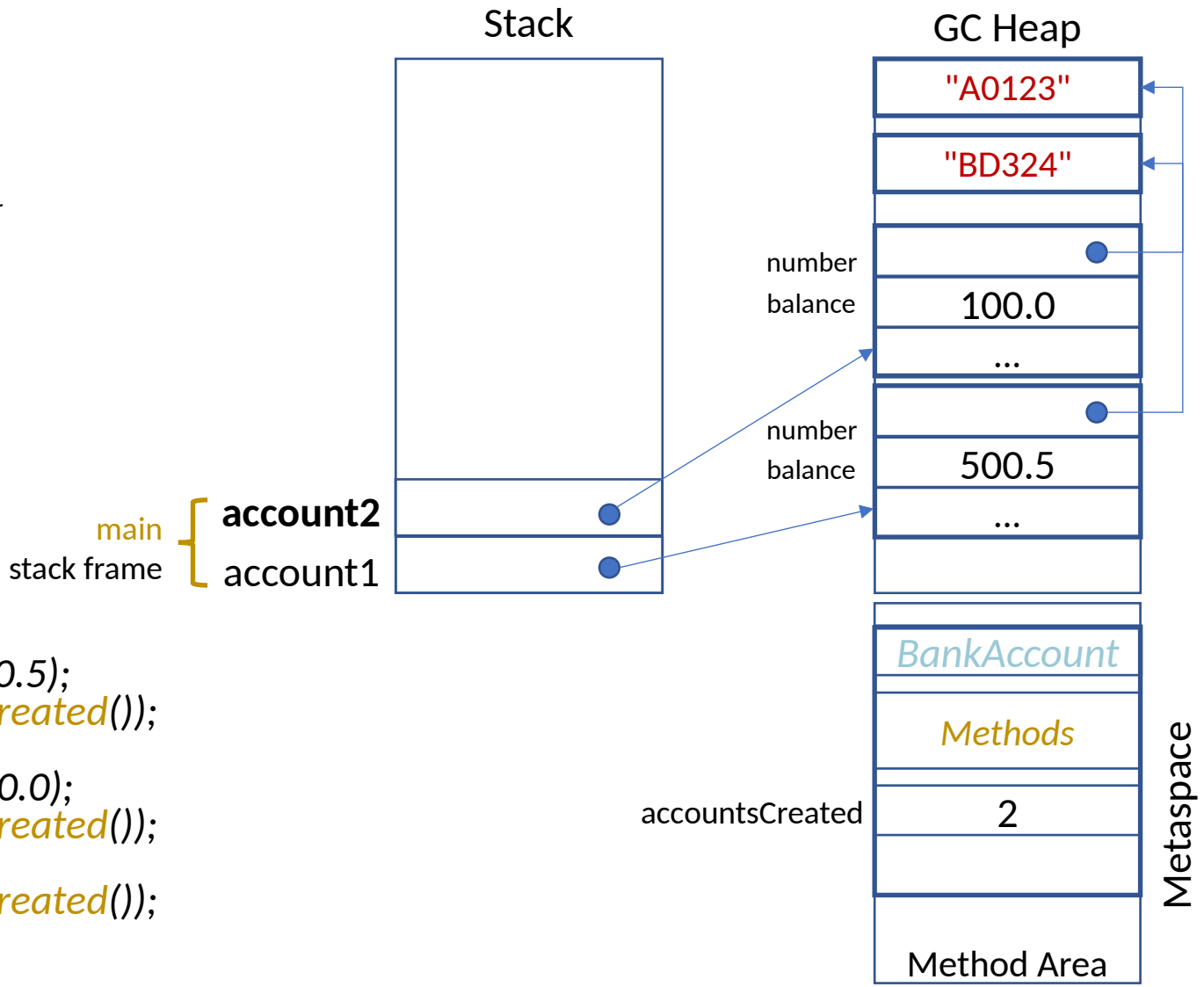
Answer

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

    public BankAccount(String num, double bal) {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    public int getAccountsCreated () {
        return accountsCreated;
    }

class Program {
    public static void main() {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        System.out.println(account1.getAccountsCreated());

        account2 = new BankAccount("BD324", 100.0);
        System.out.println(account2.getAccountsCreated());

        System.out.println(account1.getAccountsCreated());
    }
}
```



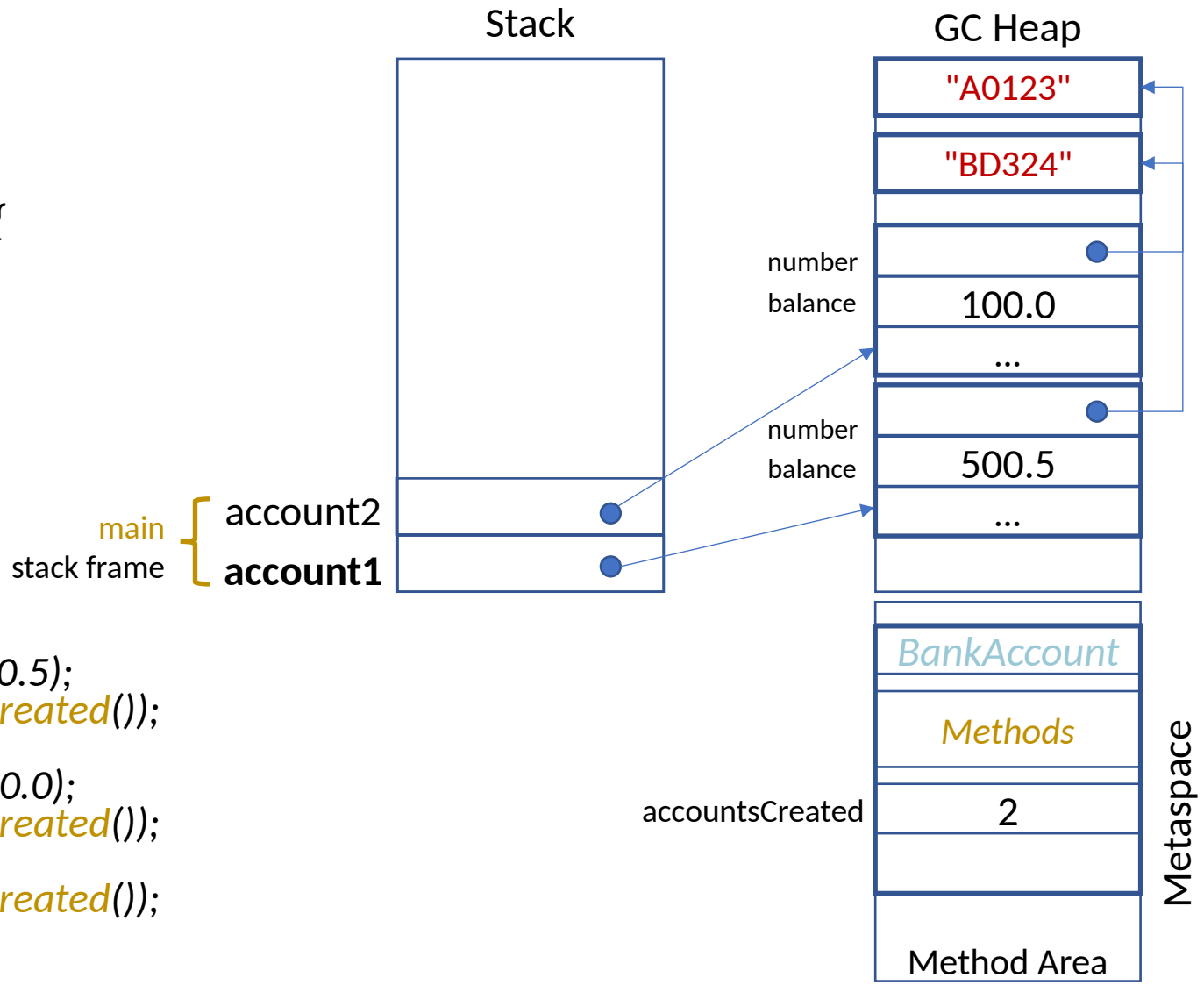
Answer

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

    public BankAccount(String num, double bal) {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    public int getAccountsCreated () {
        return accountsCreated;
    }

class Program {
    public static void main() {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        System.out.println(account1.getAccountsCreated());

        account2 = new BankAccount("BD324", 100.0);
        System.out.println(account2.getAccountsCreated());

        System.out.println(account1.getAccountsCreated());
    }
}
```



Answer

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

    public BankAccount(String num, double bal) {
        number = num;
        balance = bal;
        accountsCreated++;
    }
    public int getAccountsCreated () {
        return accountsCreated;
    }

class Program {
    public static void main() {
        BankAccount  account1, account2;
        account1 = new BankAccount("A0123", 500.5);
        System.out.println(account1.getAccountsCreated());

        account2 = new BankAccount("BD324", 100.0);
        System.out.println(account2.getAccountsCreated());

        System.out.println(account1.getAccountsCreated());
    }
}
```



static attributes: summary

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

- It is *located* on the *Metaspace* inside the *Method Area* of that class
- Its content is *not specific to an object*

Outline

- More on Memory Management
  - Parameter Passing
  - Objects lifetime
  - 'this' keyword
- Static
  - Attributes
  - **Methods**
  - main method

static methods

- Question: implement code with methods to calculate *math* functions
  - *sqrt*(…)
  - *pow* (…)
  - *log* (…)

- How would you organise the code?
- Would you create objects to do these calculations?
- Would the behaviour of these objects depend on any instance variables?

static methods

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

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

static methods

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

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

static methods

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

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

   public double add(double[] x)
   {
      double sum = 0.0;
      for (double e : x)
         sum = sum + e;
      return sum;
   }
}
```

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

static methods

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

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

    public double add(double[] x)
    {
       double sum = 0.0;
       for (double e : x)
          sum = sum + e;
       return sum;
    }
}
```

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

static methods

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

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

    public double add(double[] x)
    {
       double sum = 0.0;
       for (double e : x)
          sum = sum + e;
       return sum;
    }
}
```

The methods will *not depend on the attributes*

static methods

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

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

   public double add(double[] x)
   {
      double sum = 0.0;
      for (double e : x)
         sum = sum + e;
      return sum;
   }
}
```

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

static methods

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

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

    public static double add(double[] x)
    {
       double sum = 0.0;
       for (double e : x)
          sum = sum + e;
       return sum;
    }
}
```

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

static methods

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

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

   public static double add(double[] x)
   {
      double sum = 0.0;
      for (double e : x)
         sum = sum + e;
      return sum;
   }
}
```

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

System.out.println(CalcManager.isEven(number));
      System.out.println(CalcManager.add(values));
   }
}
```

no need to instantiate a CalcManager object

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

static methods

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

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

   public static double add(double[] x) // if isEven
   {
      double sum = 0.0;
      for (double e : x)
         if (isEven(e))
            sum = sum + e;
      return sum;
   }
```

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

static methods

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

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

Question

- Should *getAccountsCreated* be static?

Question

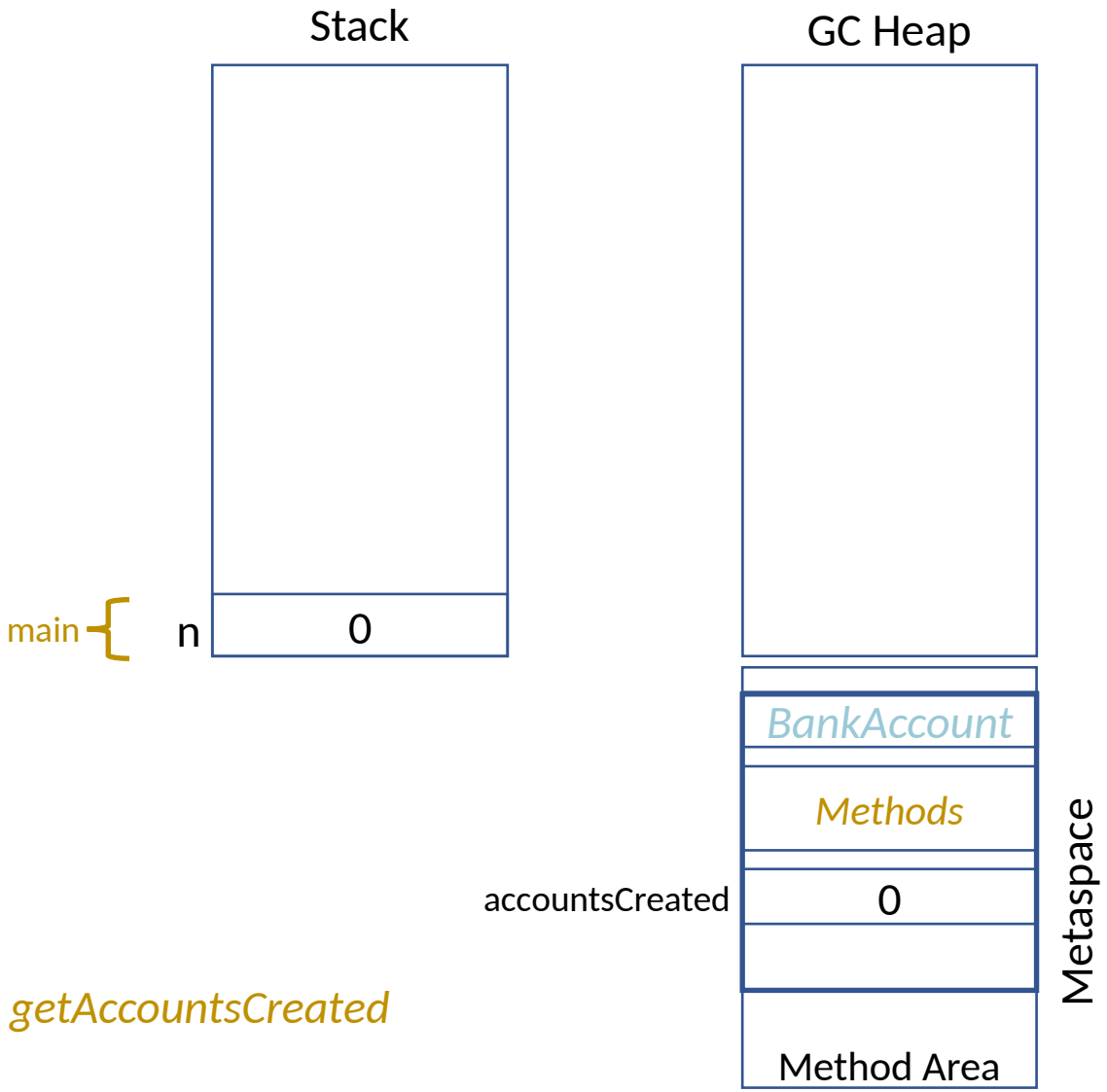
- Should *getAccountsCreated* be static?
- Yes, it **does not depend on an object state** but on the class-level static attribute *accountsCreated*

static method invocation

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

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

    public static int getAccountsCreated()
    {
        return accountsCreated;
    }

    // other methods of BankAccount
}
```

```
class Program
{
    public static void main()
    {
        int n = BankAccount.getAccountsCreated();

    }
}
```

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

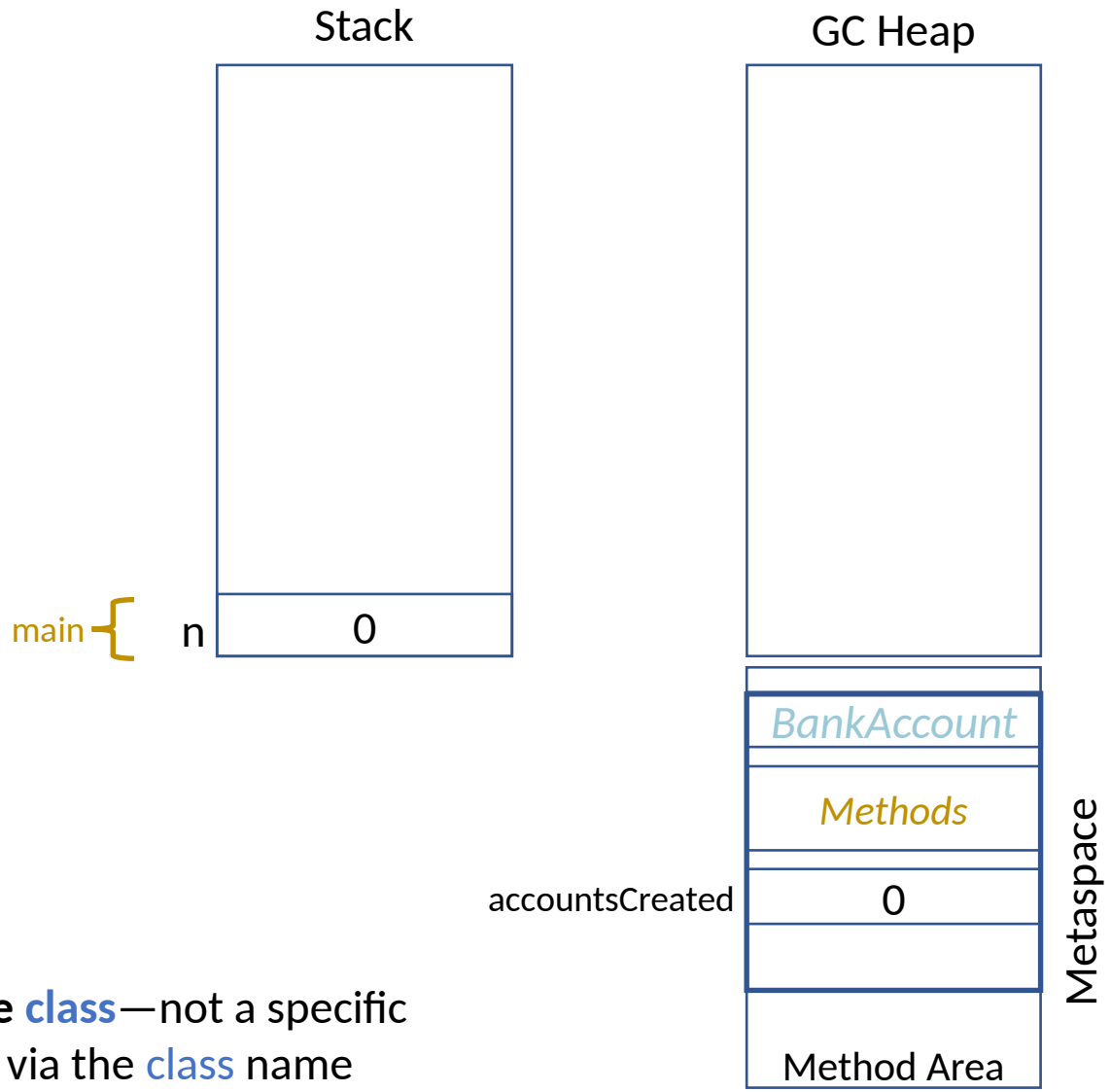


static method invocation

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

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

    public static int getAccountsCreated()
    {
        return accountsCreated;
    }

    // other methods of BankAccount
}
```

```
class Program
{
    public static void main()
    {
        int n = BankAccount.getAccountsCreated();

    }
}
```

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

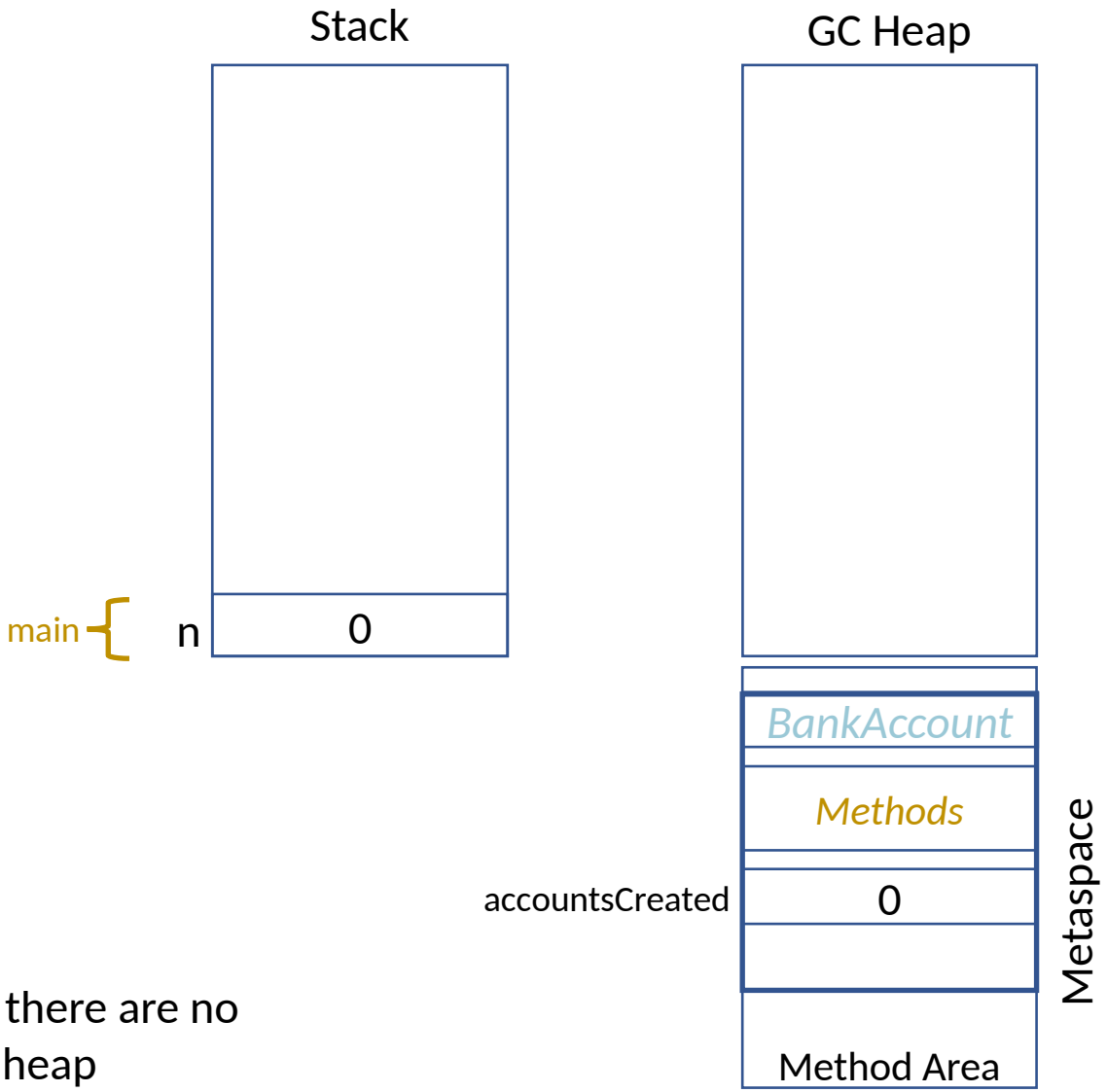


static method invocation

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

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

    public static int getAccountsCreated()
    {
        return accountsCreated;
    }

    // other methods of BankAccount
}
```

```
class Program
{
    public static void main()
    {
        int n = BankAccount.getAccountsCreated();

    }
}
```

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



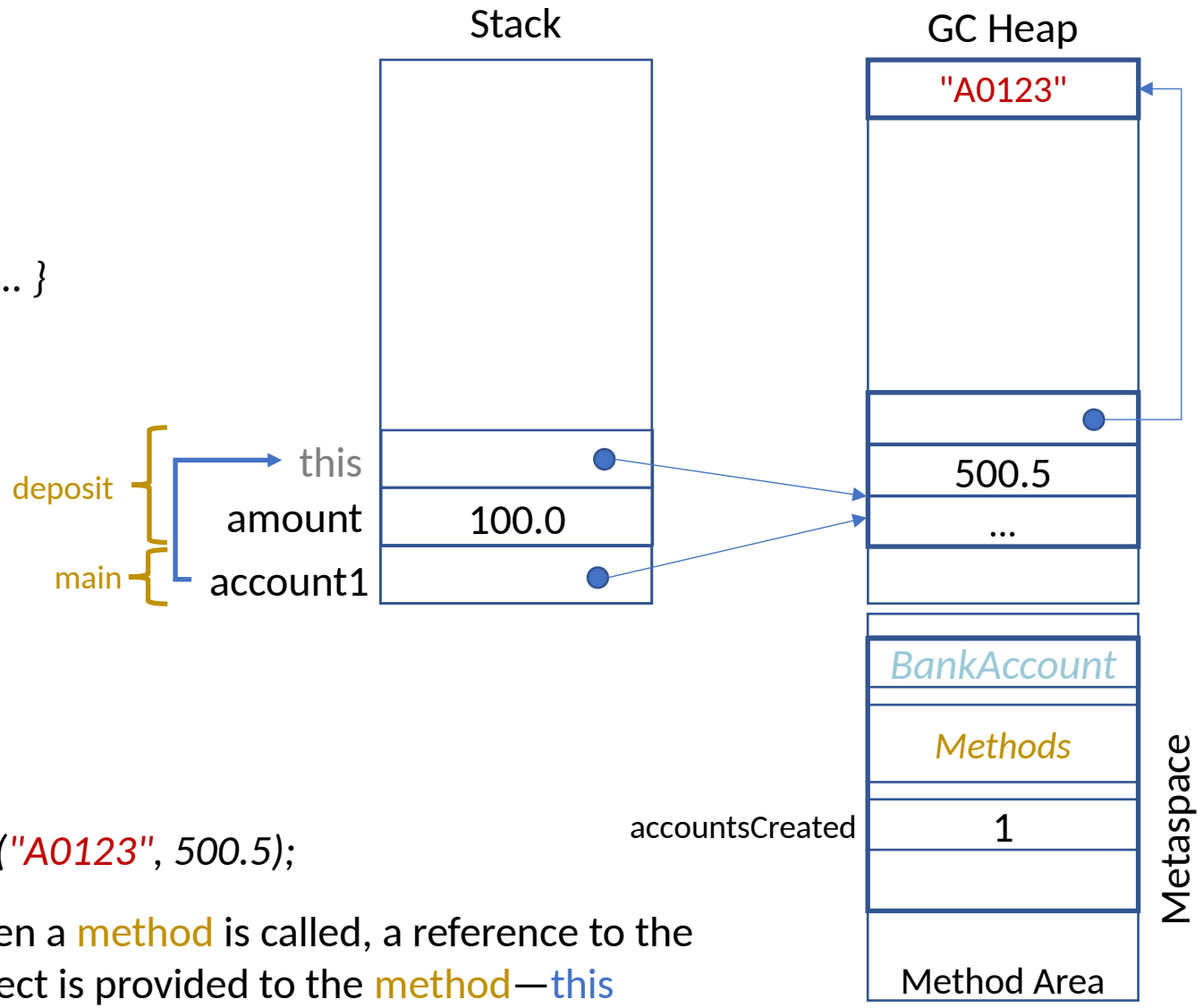
static method invocation: Question

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

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

   public static int getAccountsCreated()
   {
      System.out.println(balance);
      return accountsCreated;
   }

   // other methods of BankAccount
}

class Program
{
   public static void main()
   {
      int n = BankAccount.getAccountsCreated();

   }
}
```

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

Instance method invocation

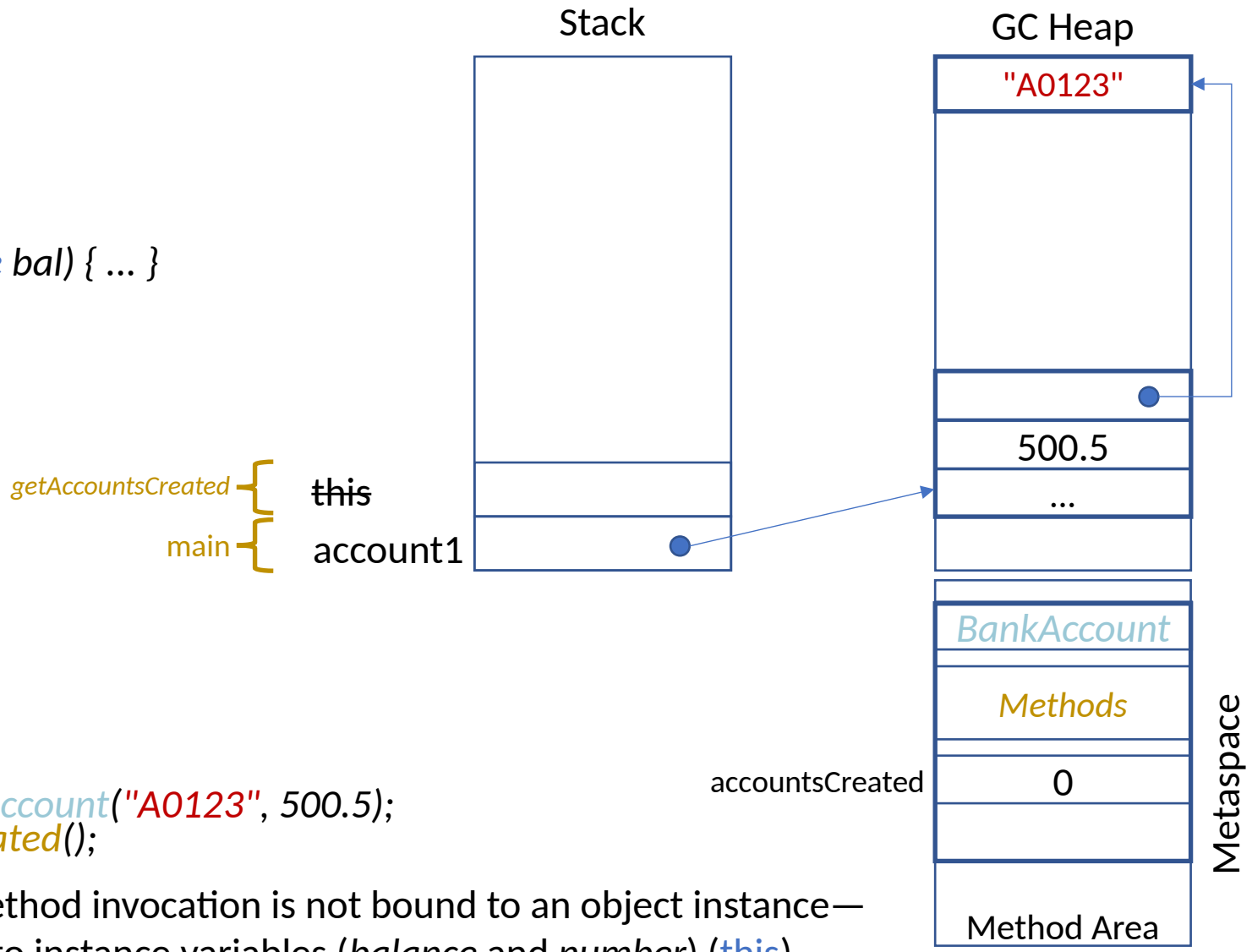
```
class BankAccount
{
    private String number;
    private double balance;
    private static int accountsCreated = 0;

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

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

    // other methods of BankAccount
}

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



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

static method invocation

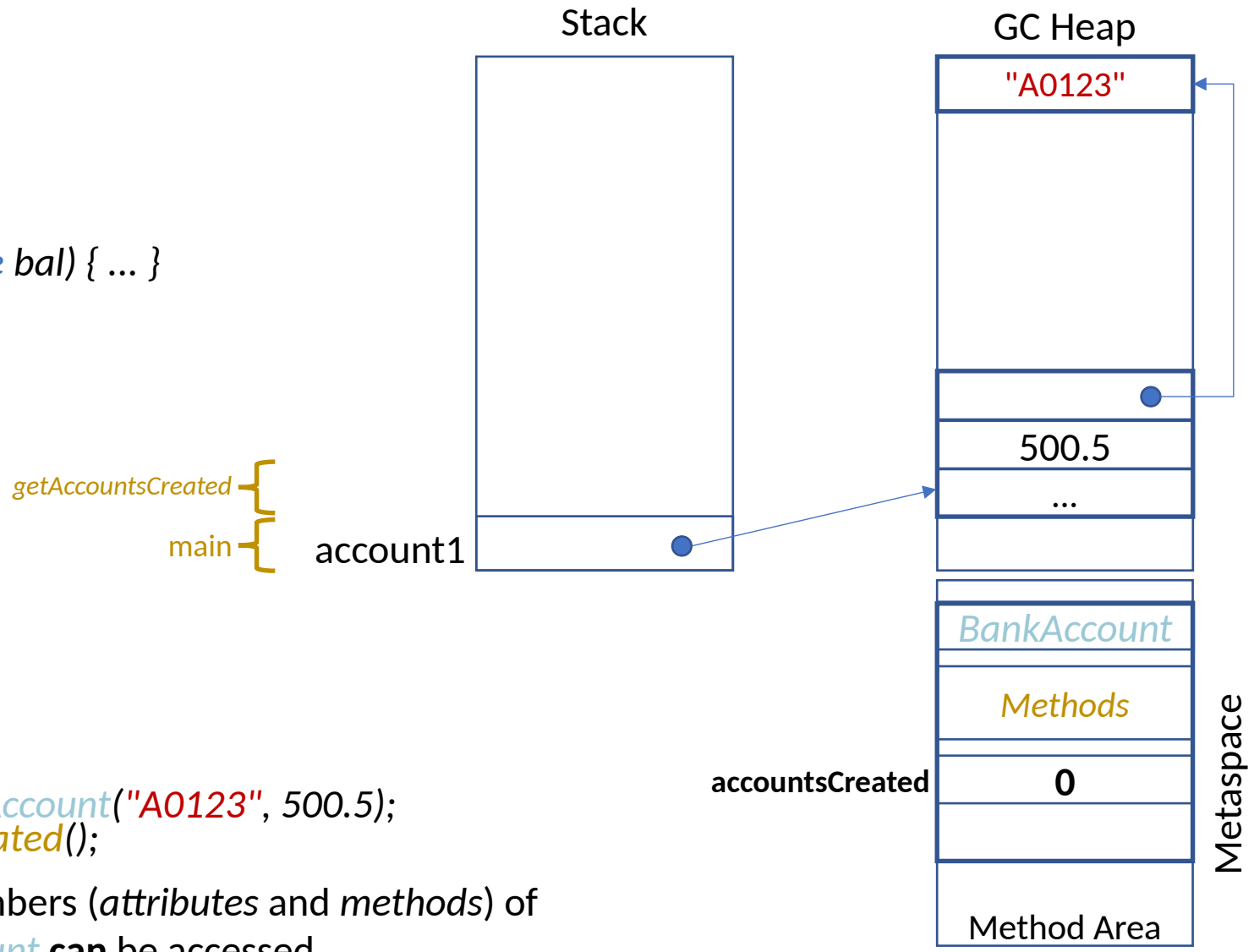
```
class BankAccount
{
    private String number;
    private double balance;
    private static int accountsCreated = 0;

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

    public static int getAccountsCreated()
    {
        System.out.println(balance);
        return accountsCreated;
    }

    // other methods of BankAccount
}

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



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

static method invocation

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

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

    public static int getAccountsCreated()
    {
        System.out.println(balance);
        return accountsCreated;
    }

    // other methods of BankAccount
}

class Program
{
    public static void main(String[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        int n = BankAccount.getAccountsCreated();

    }
}
```



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

static method invocation: Answer

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

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

    public static int getAccountsCreated()
    {
        System.out.println(balance);
        return accountsCreated;
    }

    // other methods of BankAccount
}

class Program
{
    public static void main(String[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        int n = BankAccount.getAccountsCreated();

    }
}
```

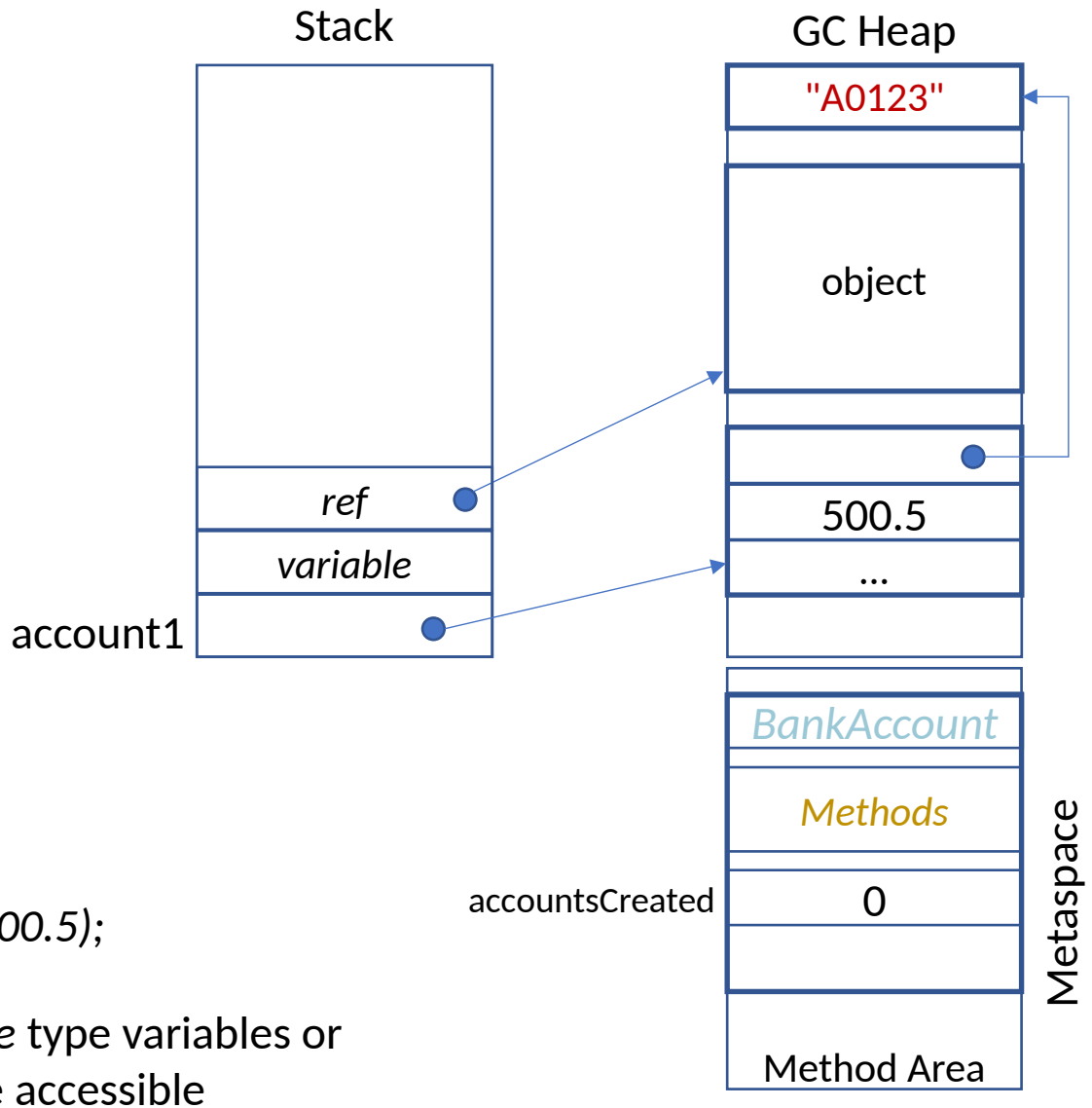
no, it would generate a compiler error!

static method invocation

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

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

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

    // other methods of BankAccount
}

class Program
{
    public static void main(String[] args)
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        int n = BankAccount.getAccountsCreated();

    }
}
```

getAccountsCreated

main



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

Outline

- More on Memory Management
  - Parameter Passing
  - Objects lifetime
  - 'this' keyword
- Static
  - Attributes
  - Methods
  - **main method**

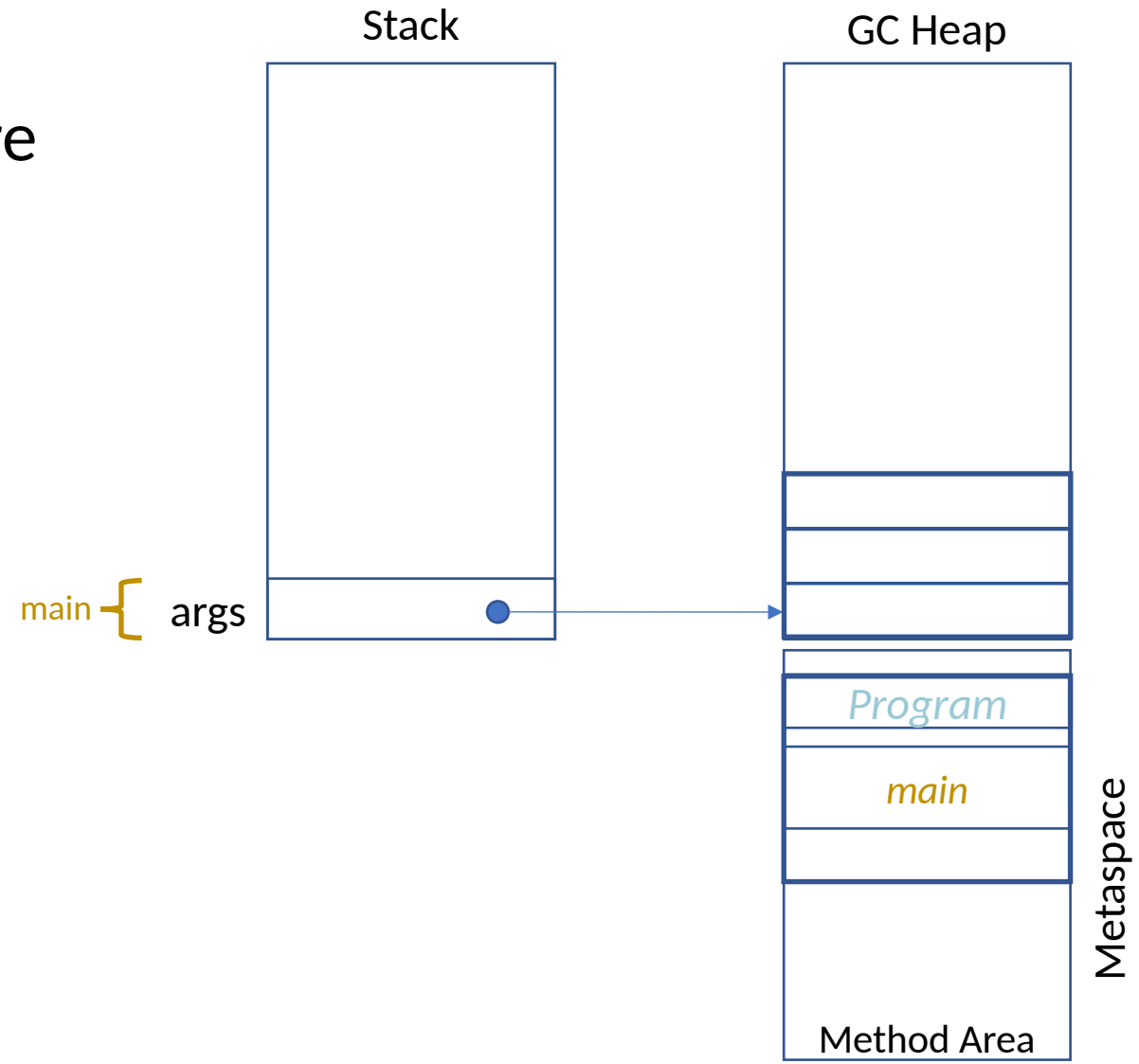
static context: main

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

static context: main

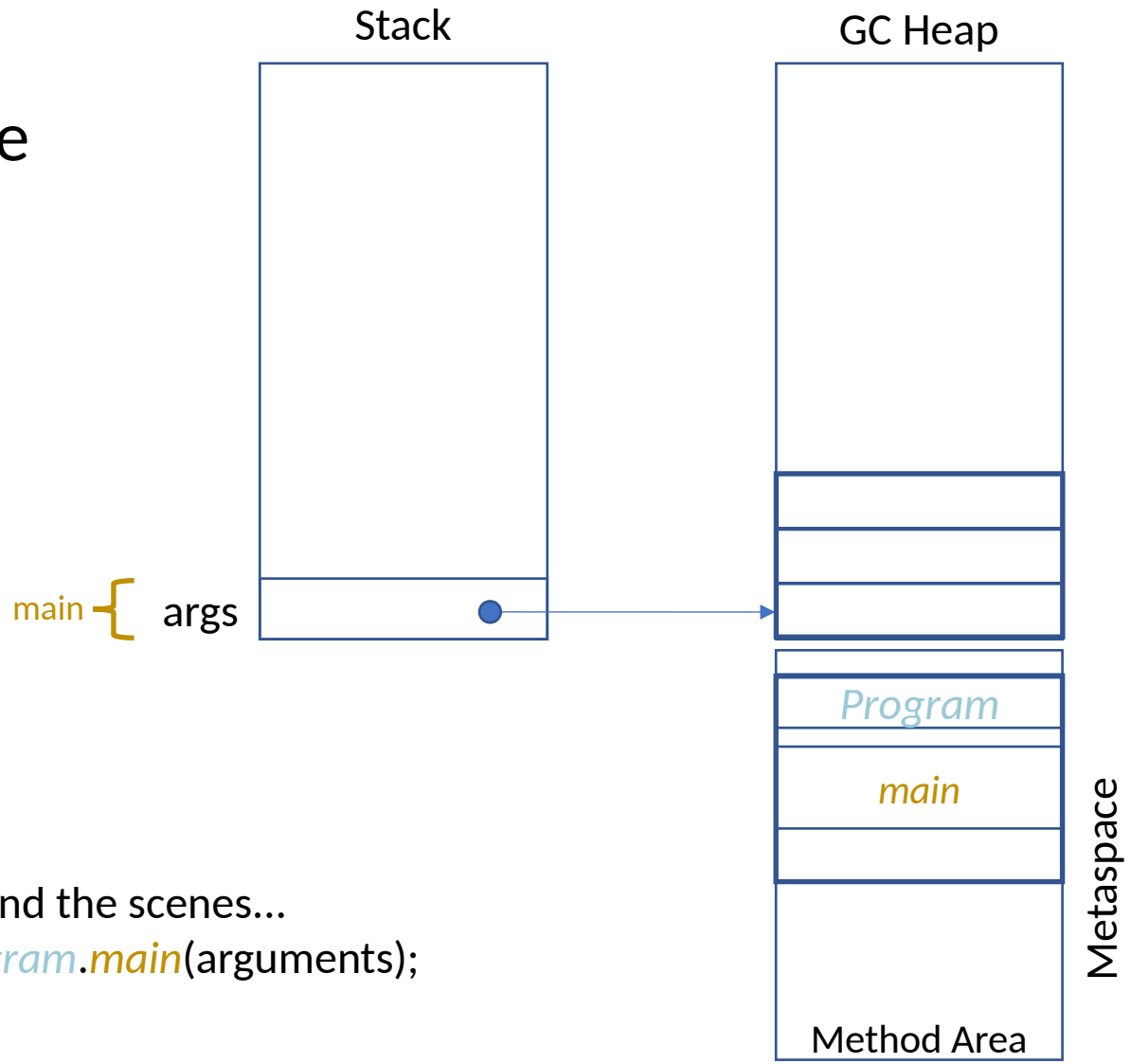
- When the program starts, there are no objects of the class where the main is defined
- The JVM needs to invoke the main without referring to an object instance of that class

```
class Program
{
    public static void main(String[] args)
    {

    }
}
```



static context: main

- When the program starts, there are no objects of the class where the main is defined
- The JVM needs to invoke the main without referring to an object instance of that class




```
class Program
{
   public static void main(String[] args)
   {

   }
}
```

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

Questions