7SENG011W
Object Oriented Programming

Java Memory Management: Value Types and Reference Types, Stack and Heap

Dr Francesco Tusa

Readings

Books

- Head First Java
  - Chapter 3

Online

- Java Language Specification – Chapter 4. Types, Values, and Variables

Outline

- Value Types and Reference Types
  - Definition
- Stack and Heap Memory
  - Concepts and Introduction
  - Usage During Method Invocation
- Value Types and Reference Types
  - Memory Allocation Examples
  - Assignment and Equality Check
  - Parameter Passing During Method Invocation

Question

On Java Memory Management

Answer on PollEveryWhere

https://pollev.com/francescotusa

Value Types and Reference Types

- **Primitive** types are basic Java types:
  - int, double, boolean, char, etc.

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

Value Types and Reference Types

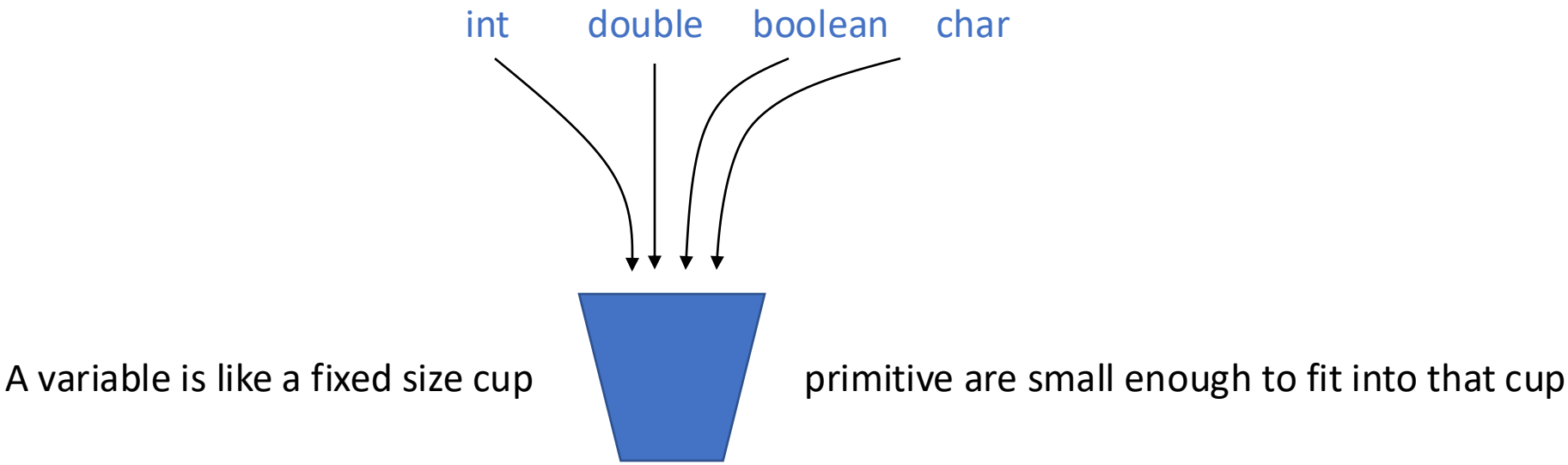
- **Primitive** types are basic Java types:
  - int, double, boolean, char, etc.
  - the actual data **values are stored inside** the variable: **value type**
- **Reference** types are *arrays* and *objects*:
  - String, int[], double[], *Point*, *Circle*, *BankAccount*, etc.

Value Types and Reference Types

- **Primitive** types are basic Java types:
  - int, double, boolean, 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)

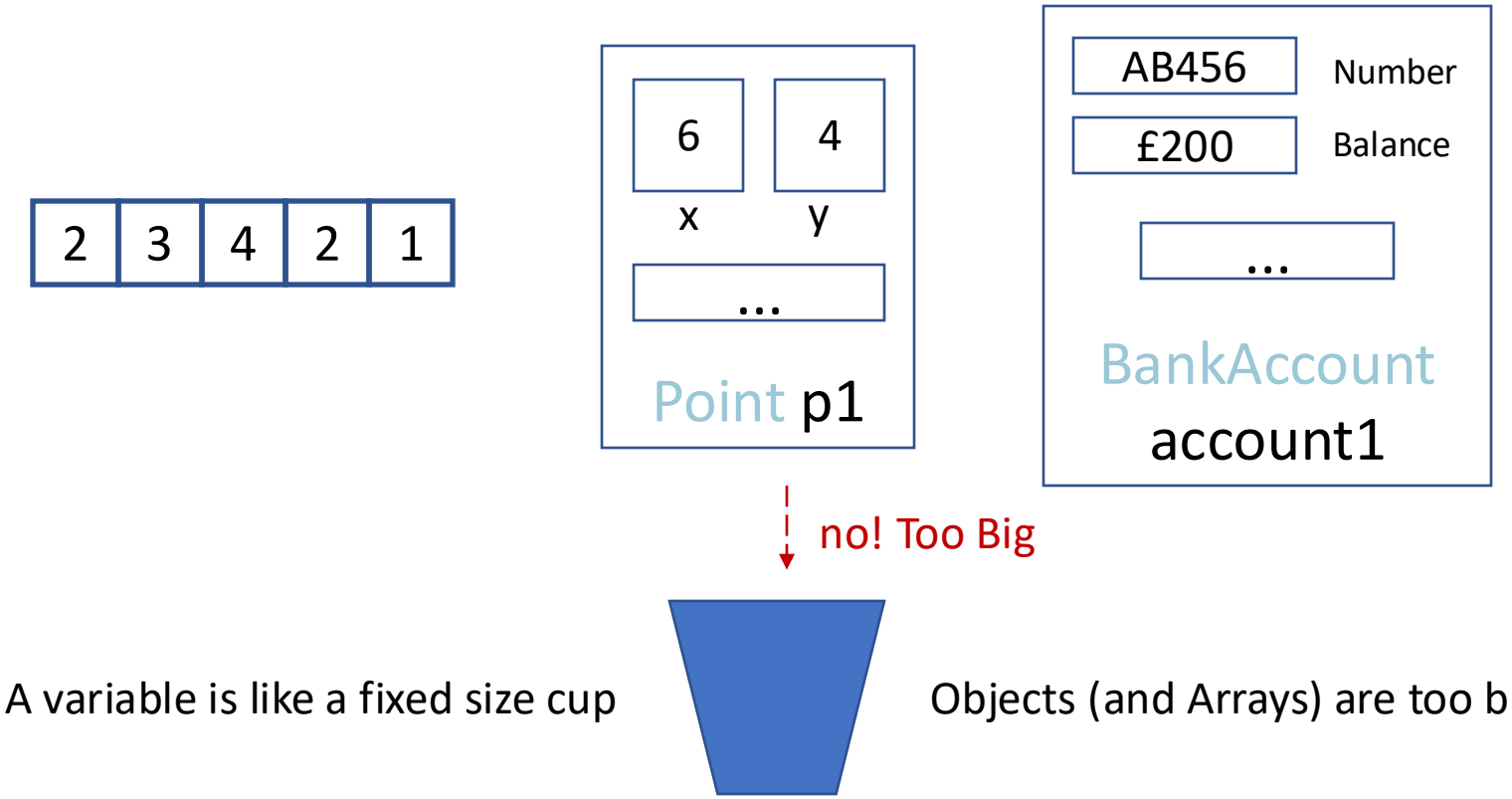
How primitive types are stored

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



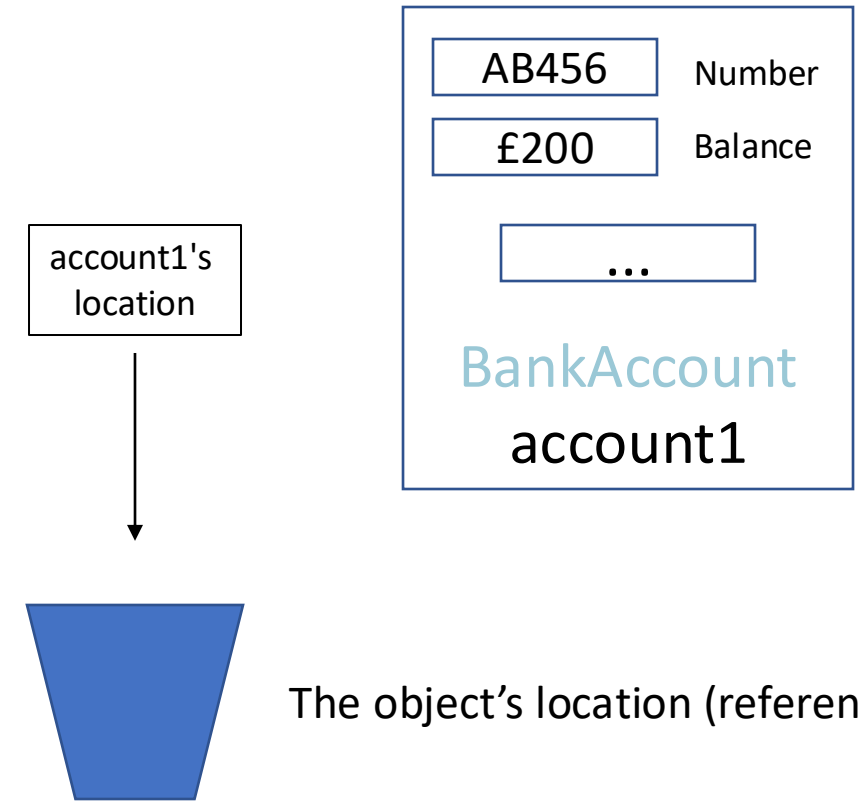
How reference types are stored

- **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**



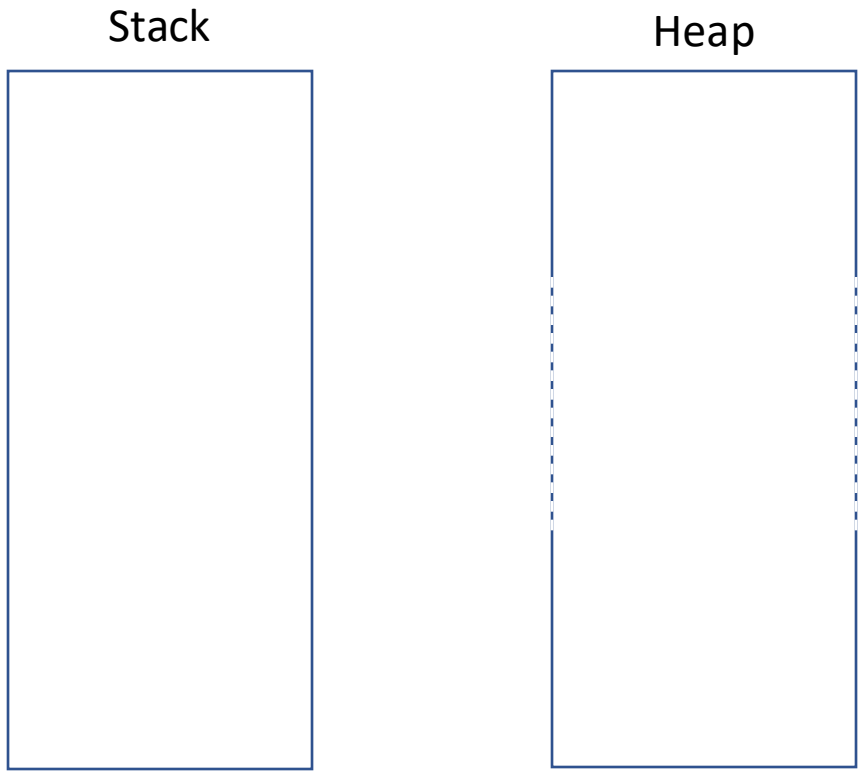
Outline

- Value Types and Reference Types
  - Definition
- Stack and Heap Memory
  - **Concepts and Introduction**
  - Usage During Method Invocation
- Value Types and Reference Types
  - Memory Allocation Examples
  - Assignment and Equality Check
  - Parameter Passing During Method Invocation

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***

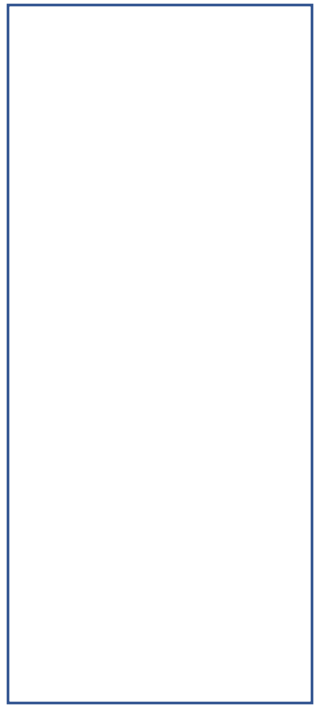
Memory stack and heap



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

Memory **stack** and heap

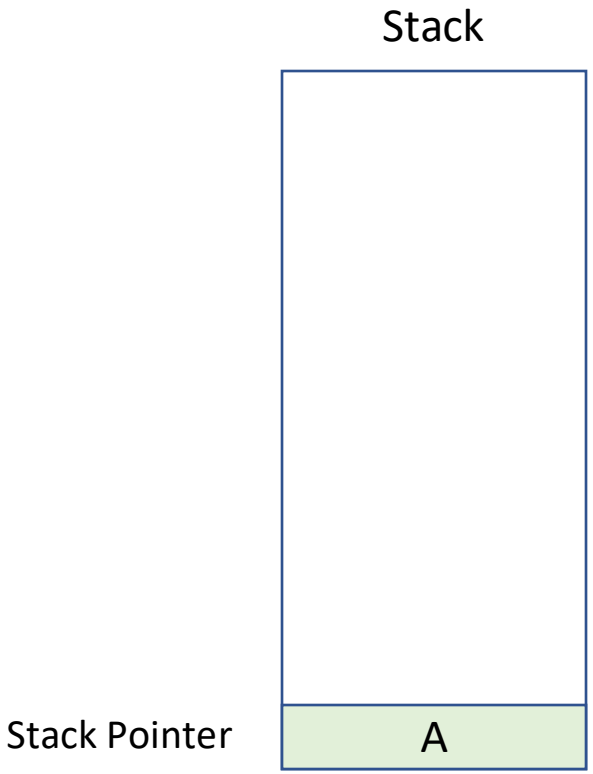
Stack



Adding element: **Push** A

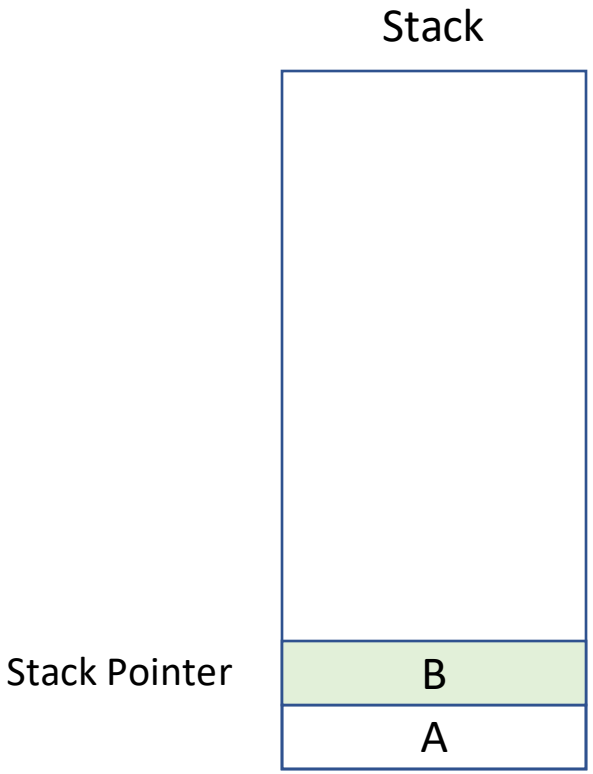
Stack memory allocation pattern: LIFO

Memory **stack** and heap



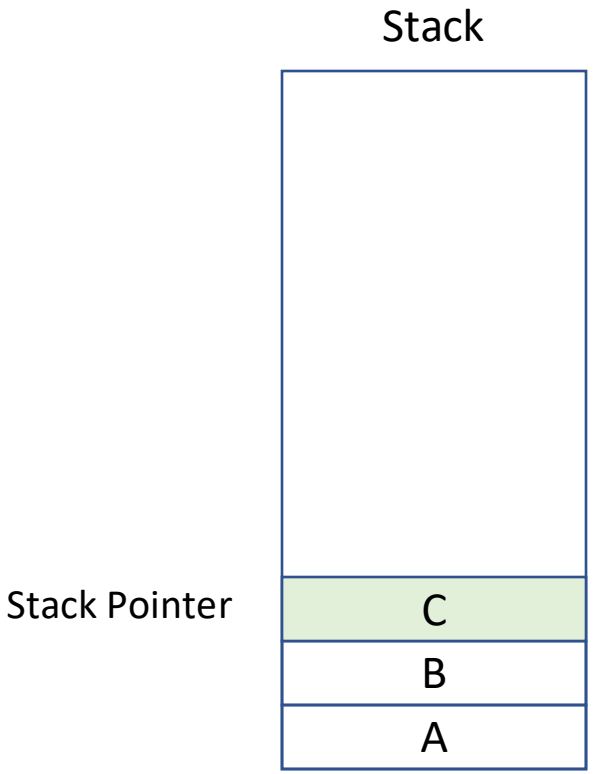
Added element: A

Memory **stack** and heap



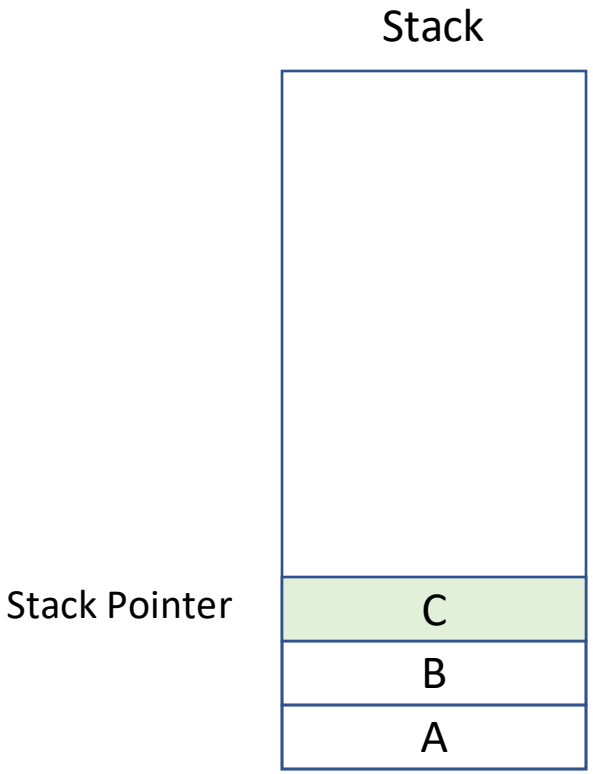
Added element: B

Memory **stack** and heap



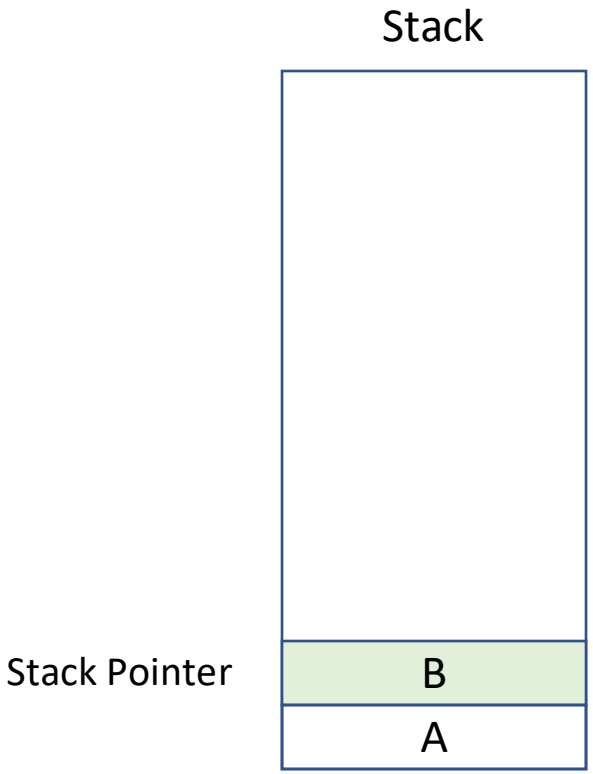
Added element: C

Memory **stack** and heap



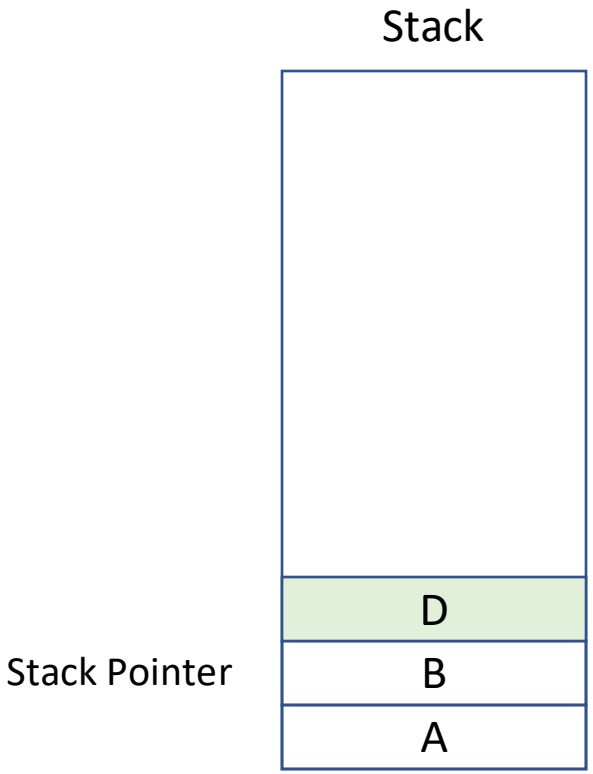
Removing element: **Pop** C

Memory **stack** and heap



Removed element: C

Memory **stack** and heap

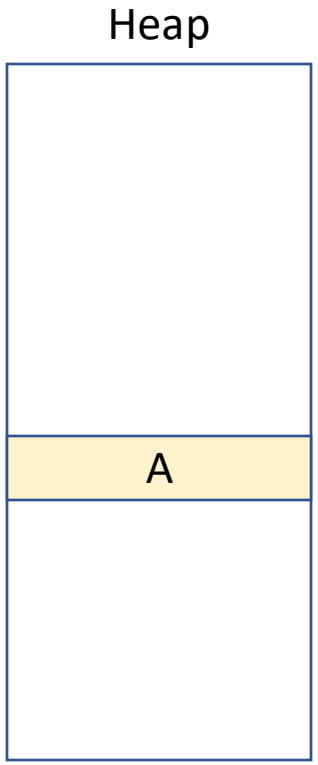


Added element: D

Stack memory allocation pattern: LIFO

Memory stack and **heap**

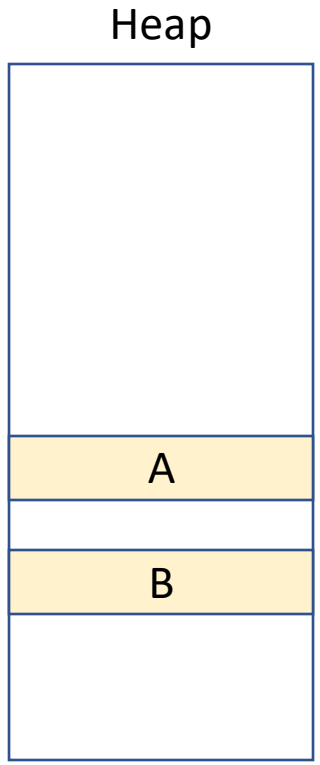
Added element: A



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

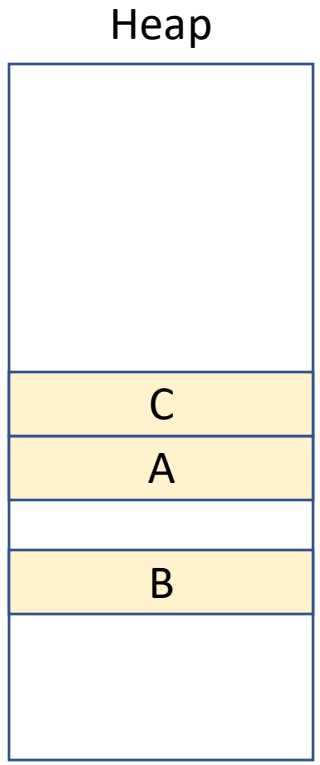
Memory stack and **heap**

Added element: B



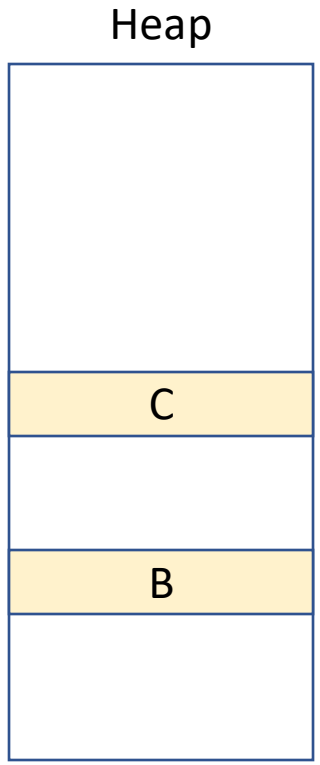
Memory stack and **heap**

Added element: C



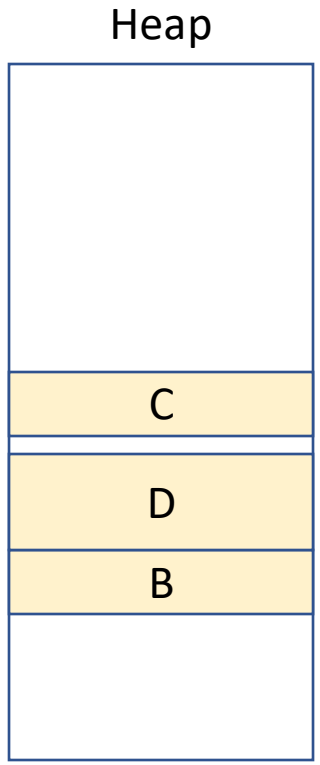
Memory stack and **heap**

Removed element: A



Memory stack and **heap**

Added element: D



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

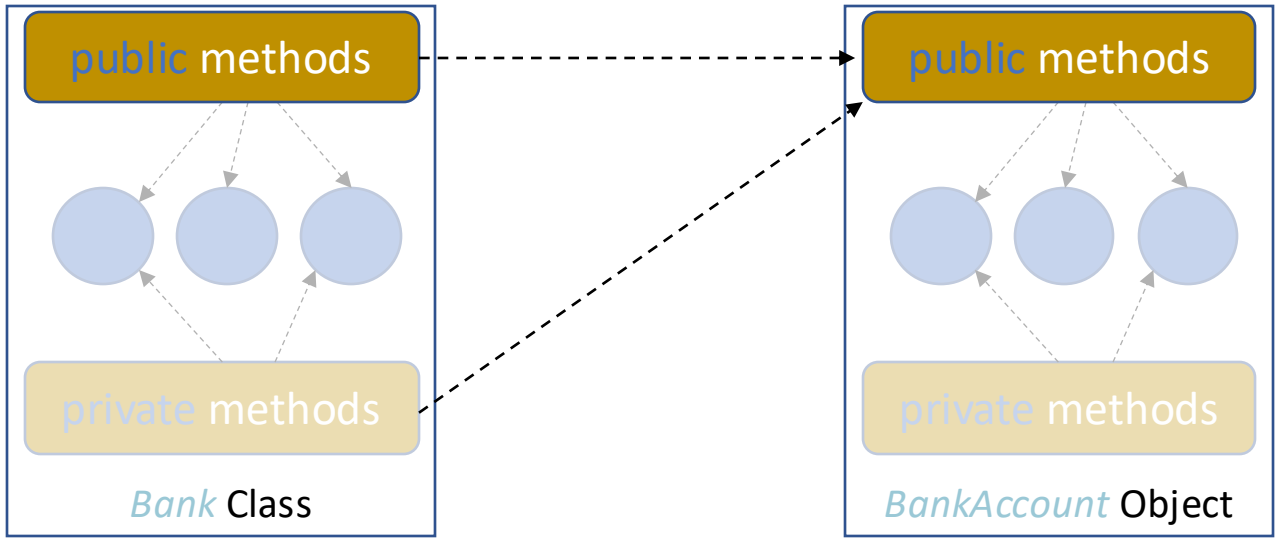
Outline

- Value Types and Reference Types
  - Definition
- Stack and Heap Memory
  - Concepts and Introduction
  - **Usage During Method Invocation**
- Value Types and Reference Types
  - Memory Allocation Examples
  - Assignment and Equality Check
  - Parameter Passing During Method Invocation

Object interface example: calling methods

public static void main(String[] args)

...

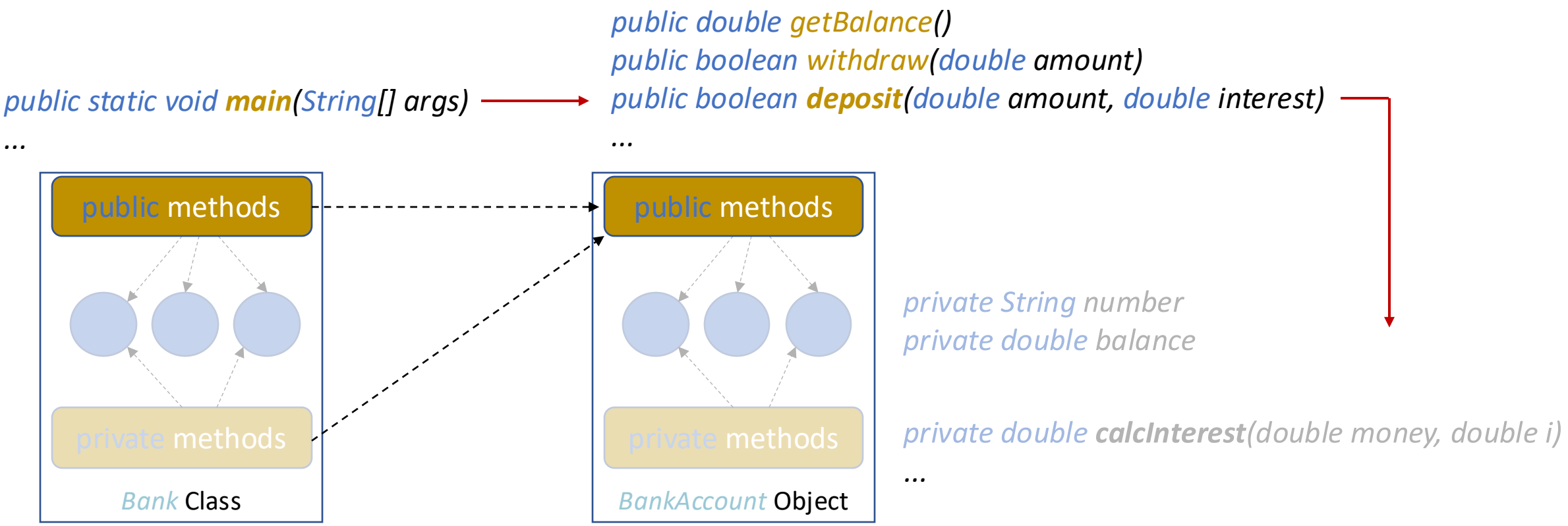


public double getBalance()
public boolean withdraw(double amount)
public boolean deposit(double amount, double interest)
...

private String number
private double balance

private double calcInterest(double money, double i)
...

Object interface example: calling methods



Object interface example: calling methods

```
class Bank {
    public static void main(String[] args) {
        BankAccount acc1 = new BankAccount("AB123", 100.0);
        double cash = 50.0;
        double fixInt = 0.03;
        acc1.deposit(cash, fixInt);
    }
}
```

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

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

    public boolean deposit(double amount, double interest) {
        if (amount < 0 || interest < 0)
            return false;

        double sum = calcInterest(amount, interest)

        balance += sum;
        return true;
    }

    private double calcInterest(double money, double i) {
        // complicated calculation
        return money * (1 + i);
    }
}
```

Object interface example: calling methods

```
class Bank {
   public static void main(String[] args) {
      BankAccount acc1 = new BankAccount("AB123", 100.0);
      double cash = 50.0;
      double fixInt = 0.03;
      acc1.deposit(cash, fixInt);
   }
}
```

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

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

   public boolean deposit(double amount, double interest) {
      if (amount < 0 || interest < 0)
         return false;

      double sum = calcInterest(amount, interest)

      balance += sum;
      return true;
   }

   private double calcInterest(double money, double i) {
      // complicated calculation
      return money * (1 + i);
   }
}
```

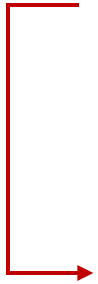
What happens inside the memory when the *main* calls *deposit* and *deposit* calls *calcInterest*?

Object interface example: calling methods

```
class Bank {
   public static void main(String[] args) {
      BankAccount acc1 = new BankAccount("AB123", 100.0);
      double cash = 50.0;
      double fixInt = 0.03;
      acc1.deposit(cash, fixInt);
   }
}
```



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

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

   public boolean deposit(double amount, double interest) {
      if (amount < 0 || interest < 0)
         return false;

      double sum = calcInterest(amount, interest)

      balance += sum;
      return true;
   }

   private double calcInterest(double money, double i) {
      // complicated calculation
      return money * (1 + i);
   }
}
```



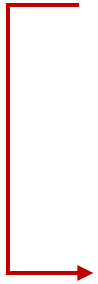
During each method call, the associated method's **parameters** and **local variables** need to be allocated in memory

Object interface example: calling methods

```
class Bank {
    public static void main(String[] args) {
        BankAccount acc1 = new BankAccount("AB123", 100.0);
        double cash = 50.0;
        double fixInt = 0.03;
        acc1.deposit(cash, fixInt);
    }
}
```



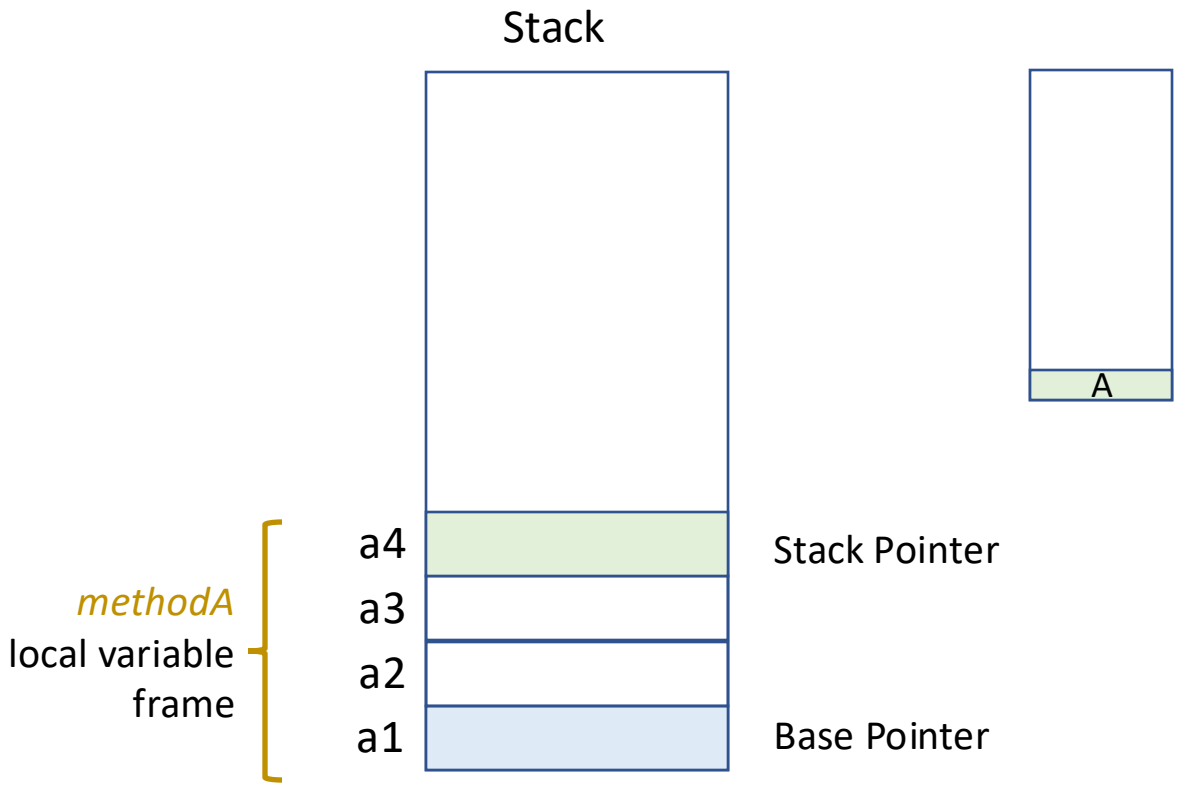
```
class BankAccount {
    private String number;
    private double balance;

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

    public boolean deposit(double amount, double interest) {
        if (amount < 0 || interest < 0)
            return false;

        double sum = calcInterest(amount, interest)

        balance += sum;
        return true;
    }

    private double calcInterest(double money, double i) {
        // complicated calculation
        return money * (1 + i);
    }
}
```



We will now go through the details of how the **stack** and **heap** memory are involved in the process

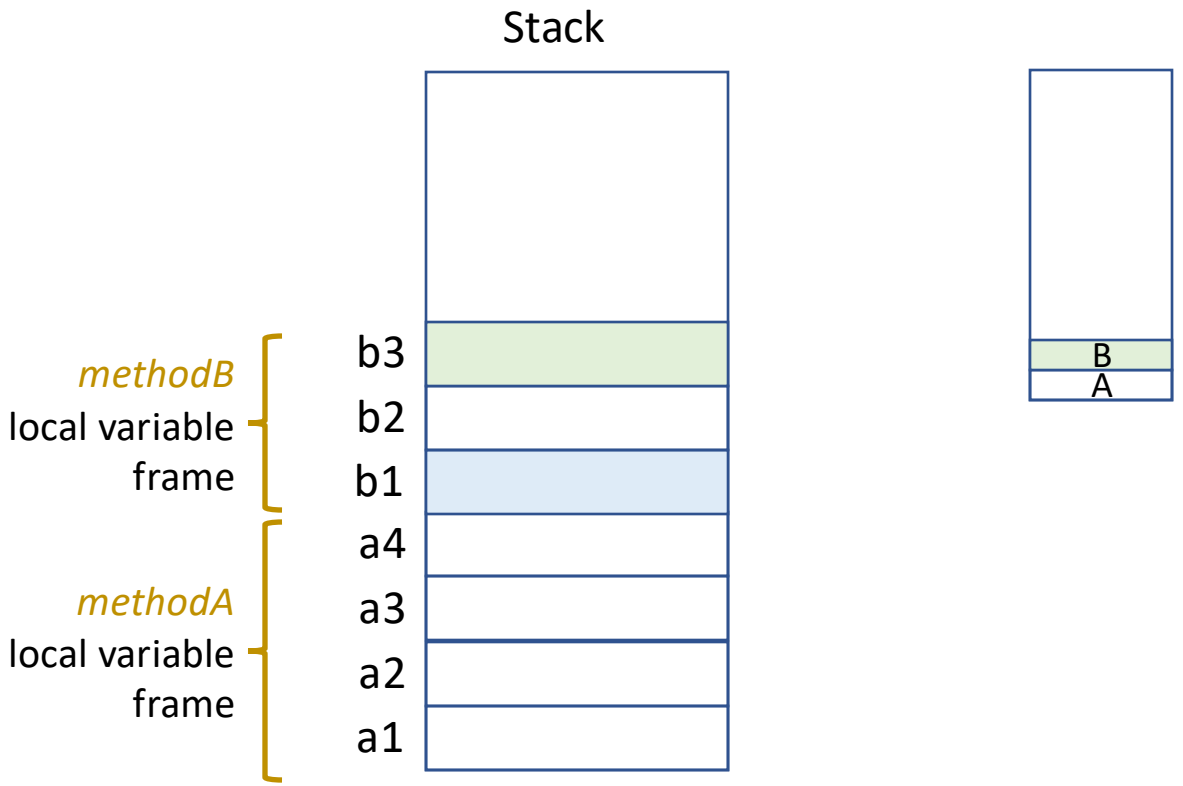
Method Invocation

```
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

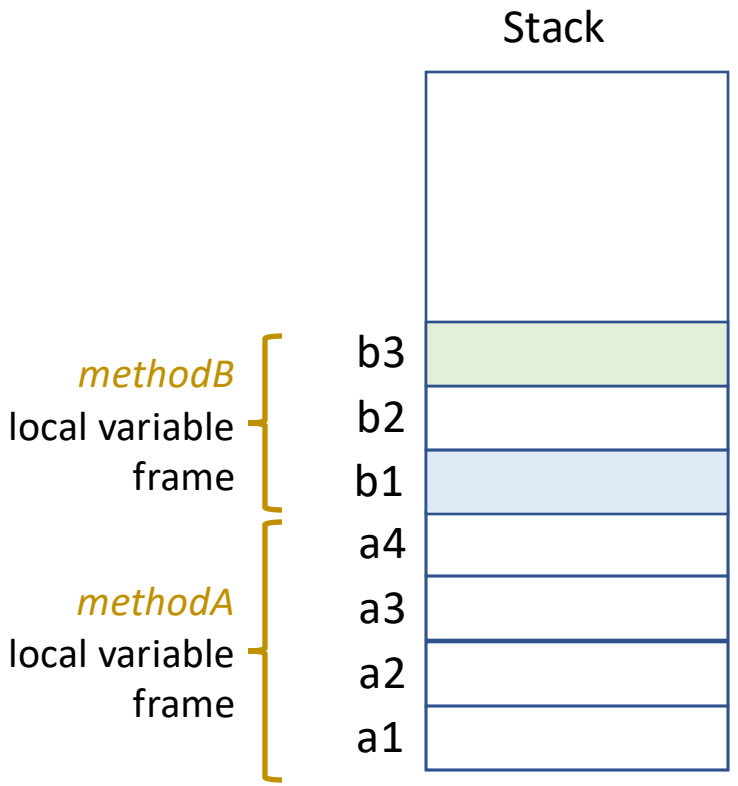
```
void methodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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*?

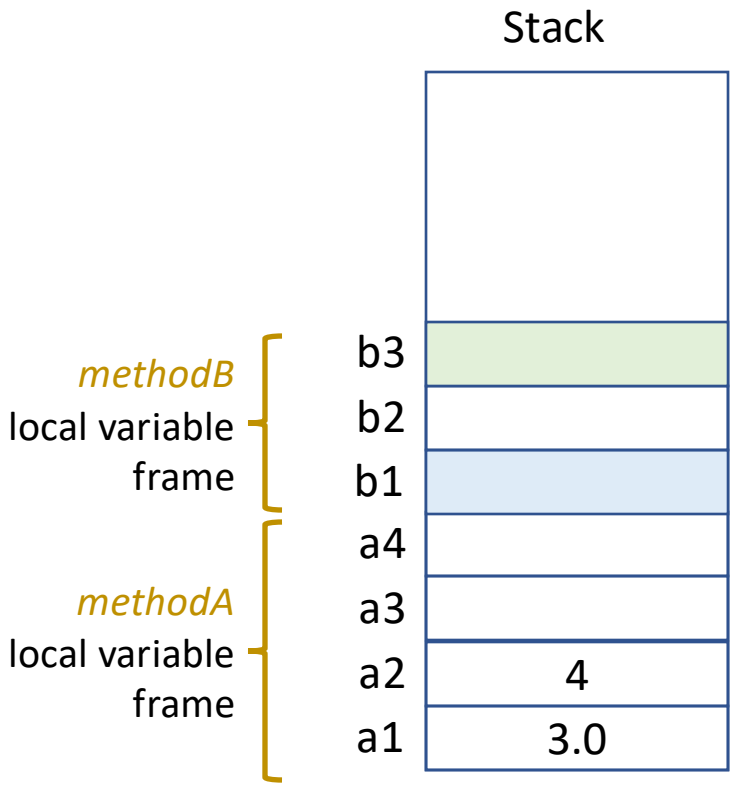
```
void methodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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*?

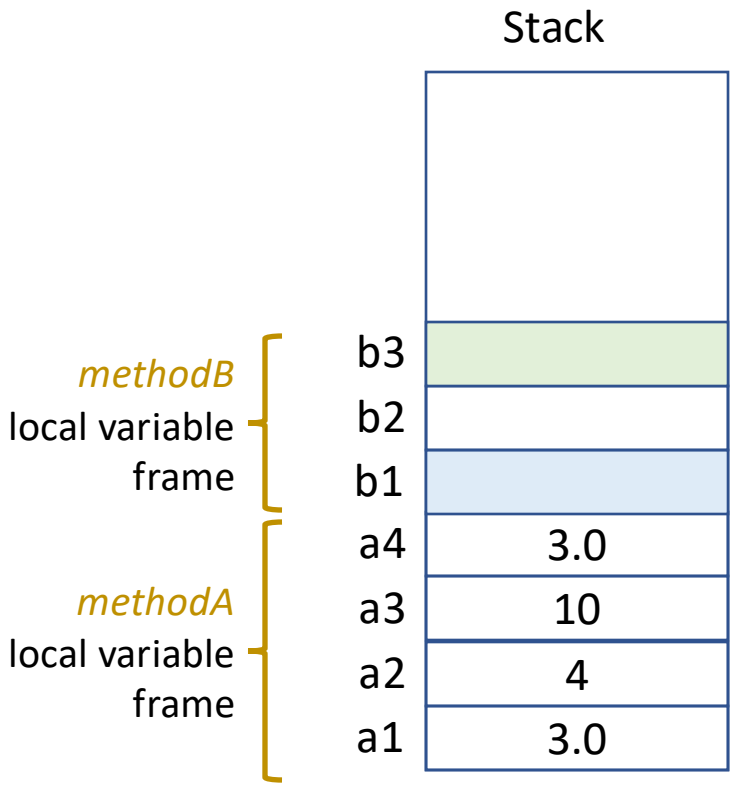
```
void methodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

                              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);
}
```

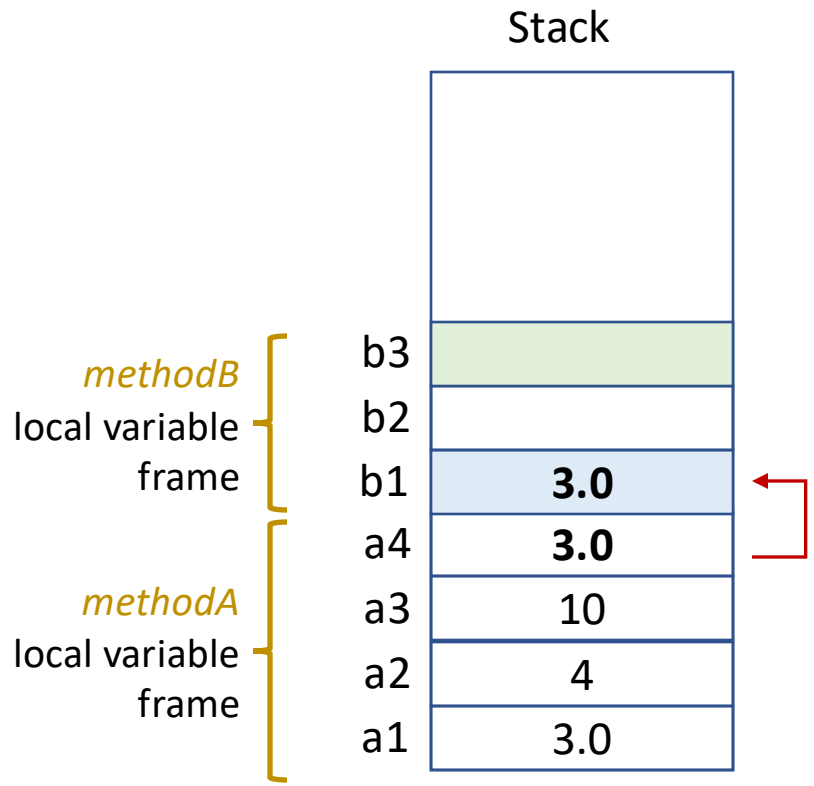


Method Invocation

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*

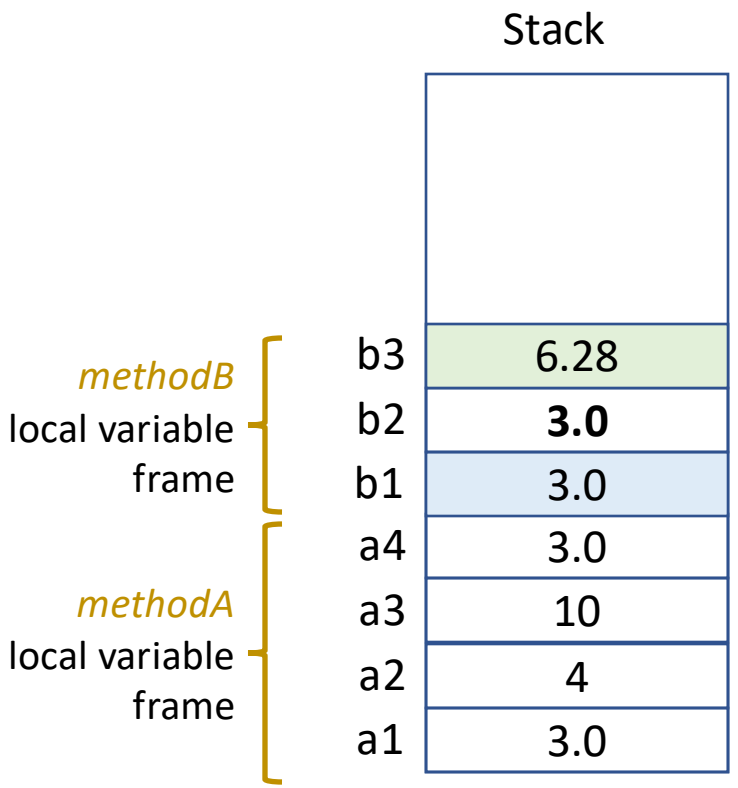
```
void methodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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



Method Invocation

Eventually, the value 3.0 will be stored inside *b2*

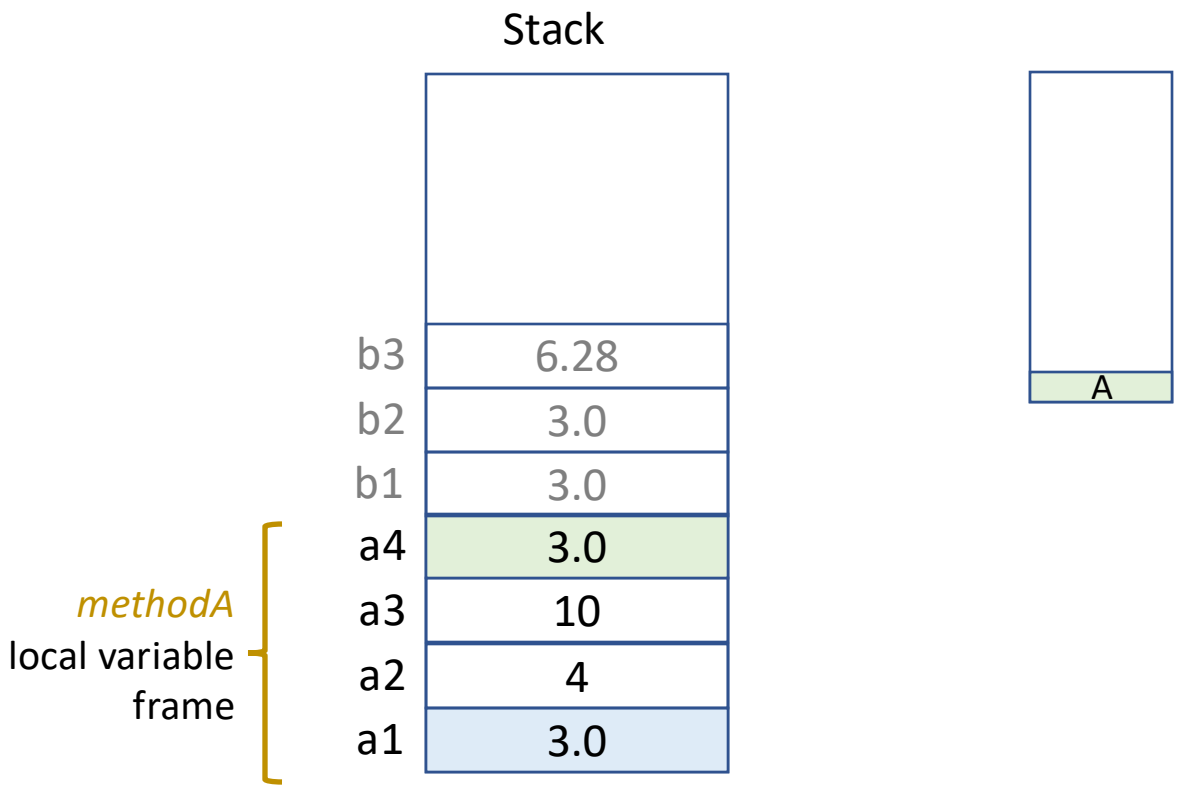
```
void methodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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

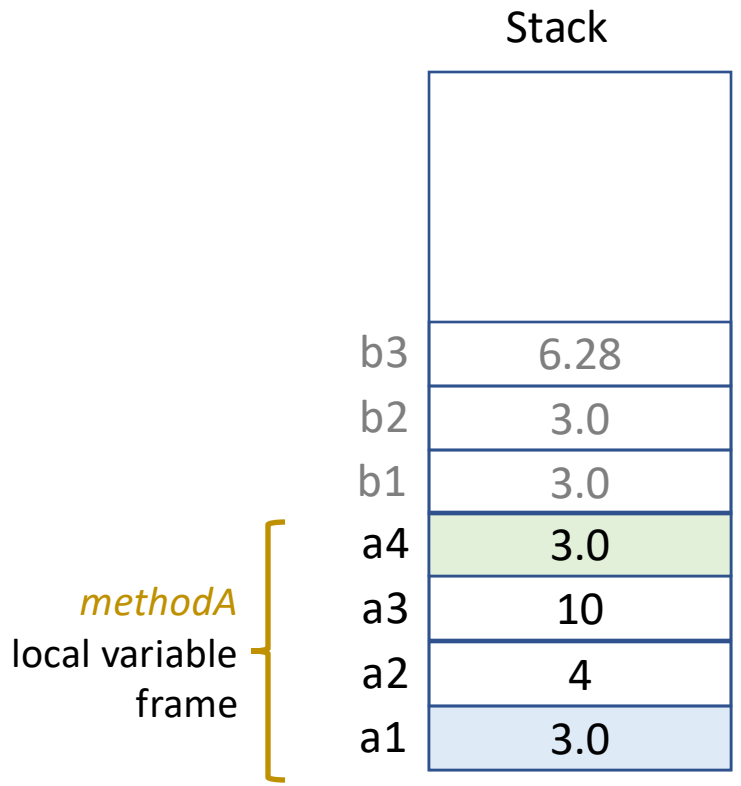
```
void methodB(double b1)
{
  double b2 = b1;
  double b3 = 6.28;
}

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*?

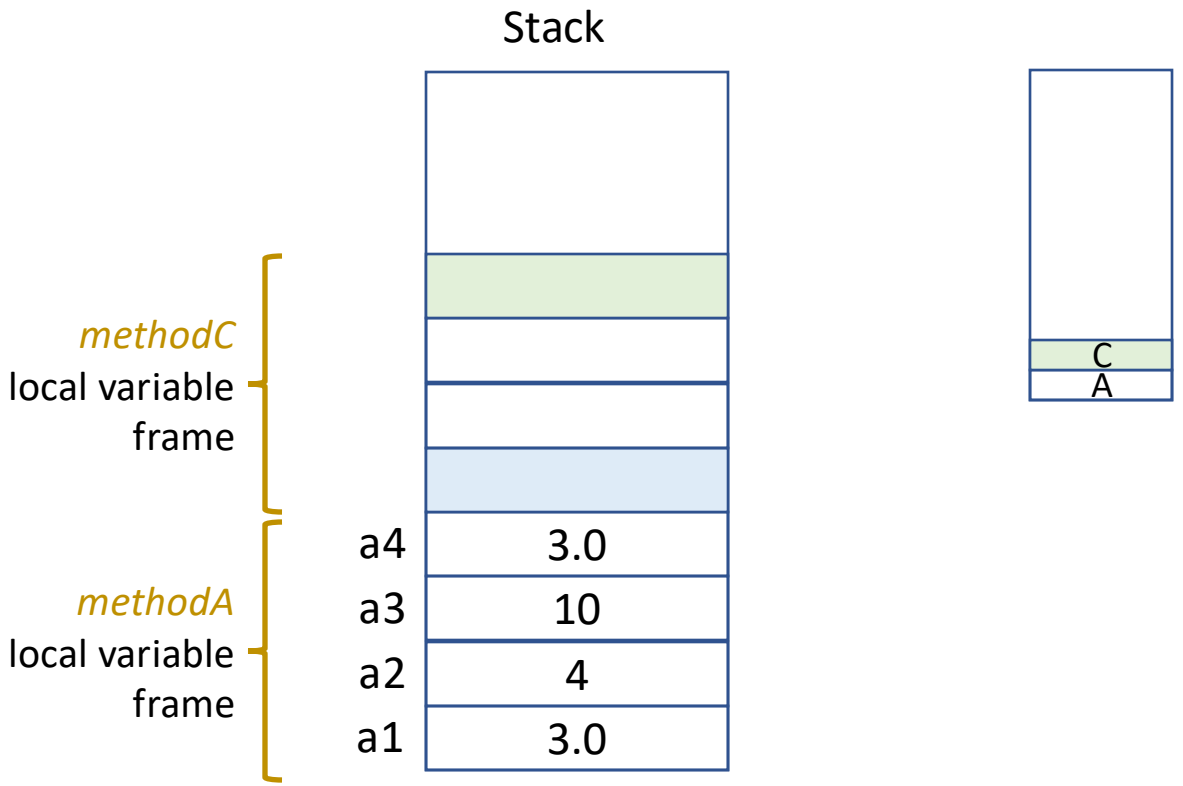
```
void methodC( ... )
{
   ...

}

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 the space previously assigned to *methodB*'s one

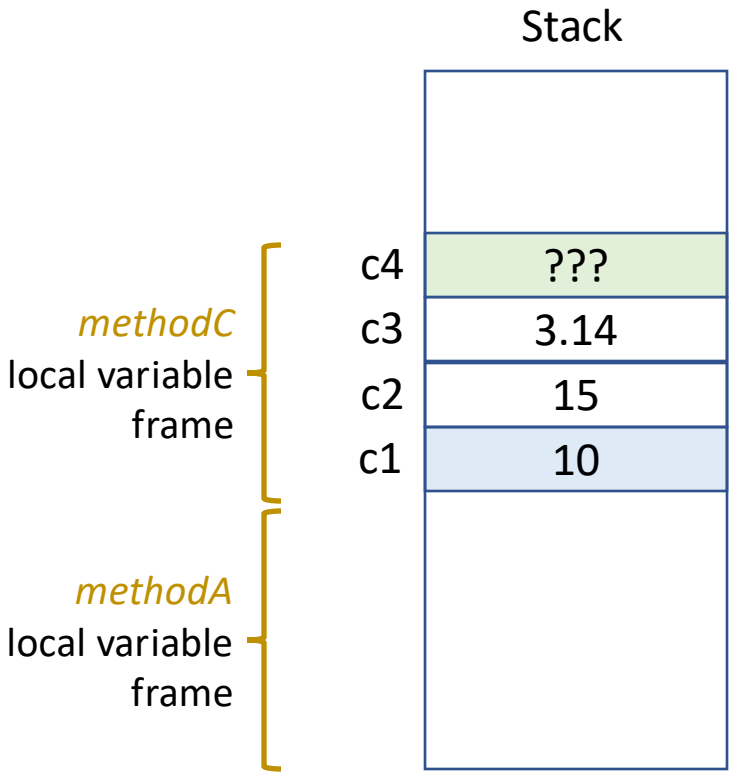
```
void methodC( ... )
{
   ...

}

void methodA(double a1, int a2)
{
   int a3 = 10;
   double a4 = a1;
   methodB(a4);
   methodC(a3);
}
```



Method invocation: Object 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: overview

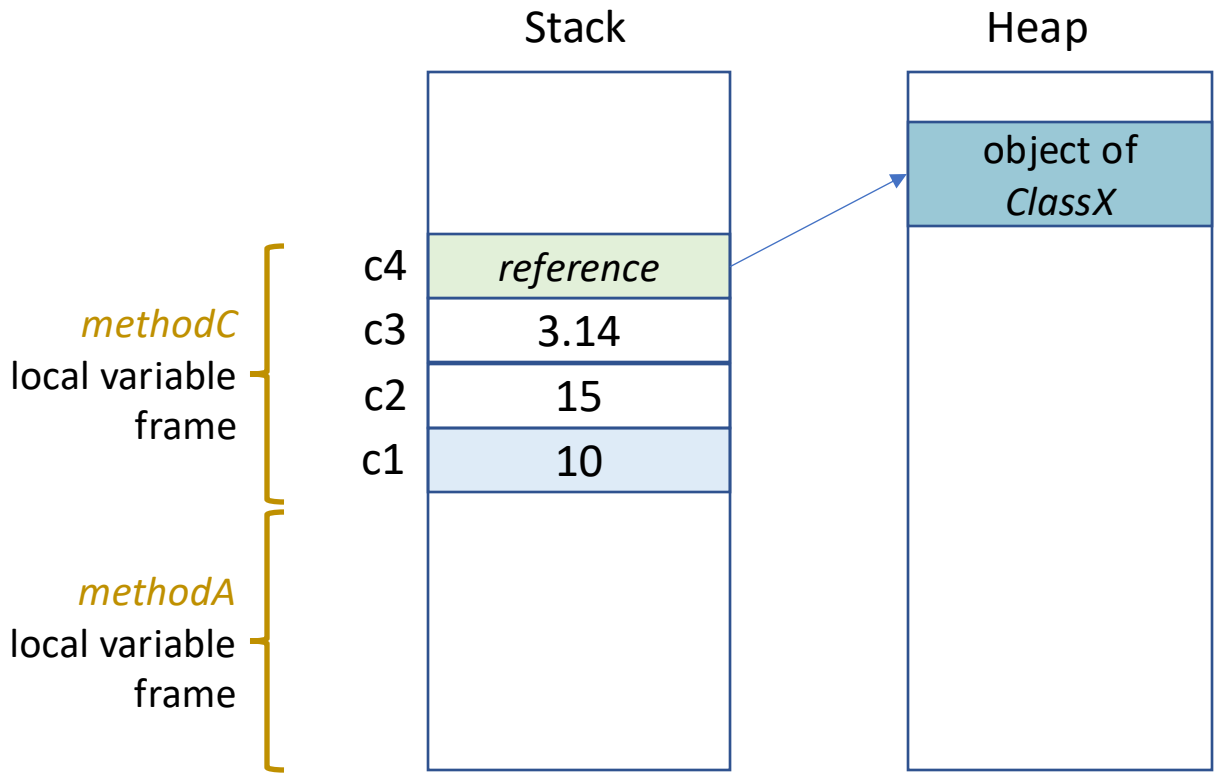
- *Primitive types* (**value types**) methods' *parameters* and *local variables* declared inside methods are allocated on the ***stack***

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

Method invocation: Object allocation

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

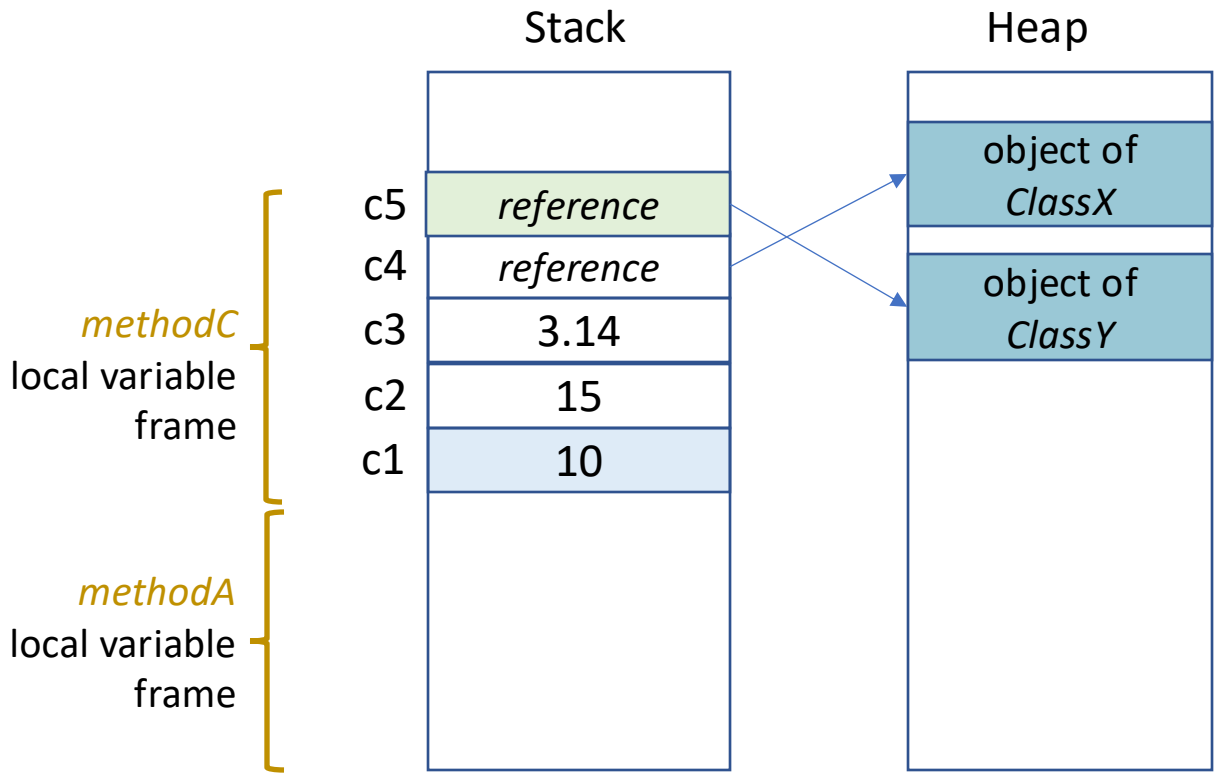
```
void methodC(int c1)
{
  int c2 = 15;
  double c3 = 3.14;
  ClassX c4 = new ClassX();
}

void methodA(double a1, int a2)
{
  int a3 = 10;
  double a4 = a1;
  methodB(a4);
  methodC(a3);
}
```



Method invocation: Object allocation

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

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

void methodA(double a1, int a2)
{
  int a3 = 10;
  double a4 = a1;
  methodB(a4);
  methodC(a3);
}
```

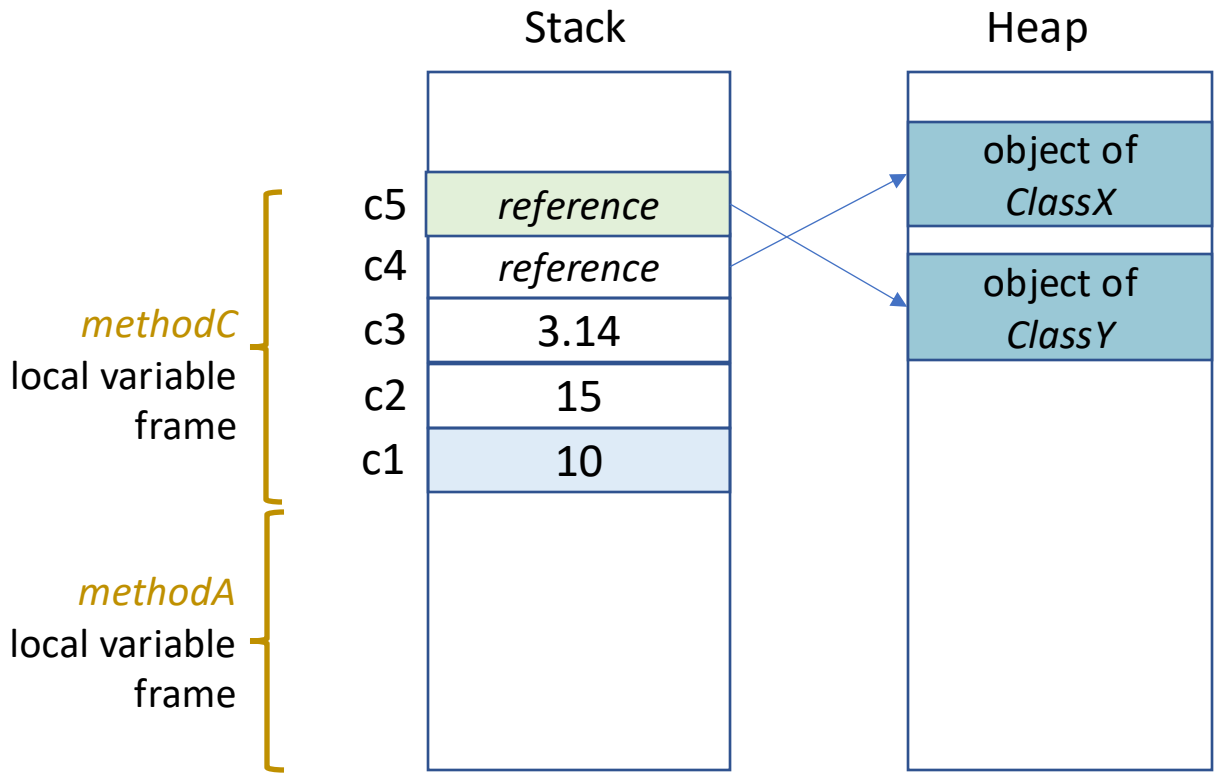


Method invocation: Object allocation

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

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

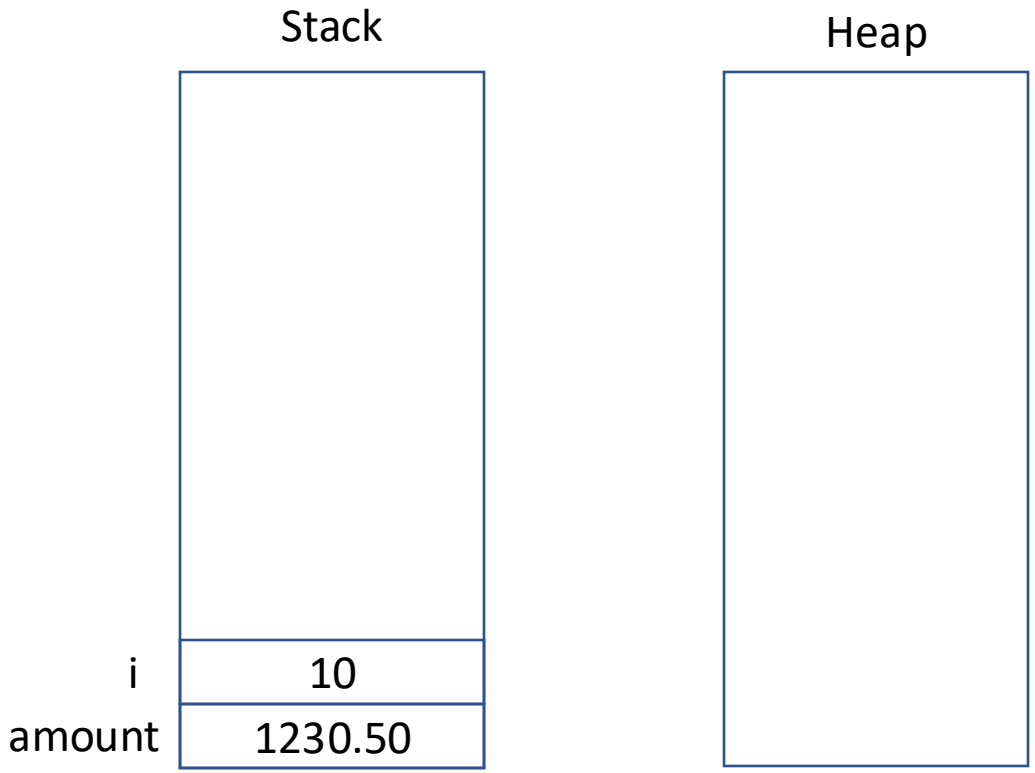


Outline

- Value Types and Reference Types
  - Definition
- Stack and Heap Memory
  - Concepts and Introduction
  - Usage During Method Invocation
- Value Types and Reference Types
  - **Memory Allocation Examples**
  - Assignment and Equality Check
  - Parameter Passing During Method Invocation

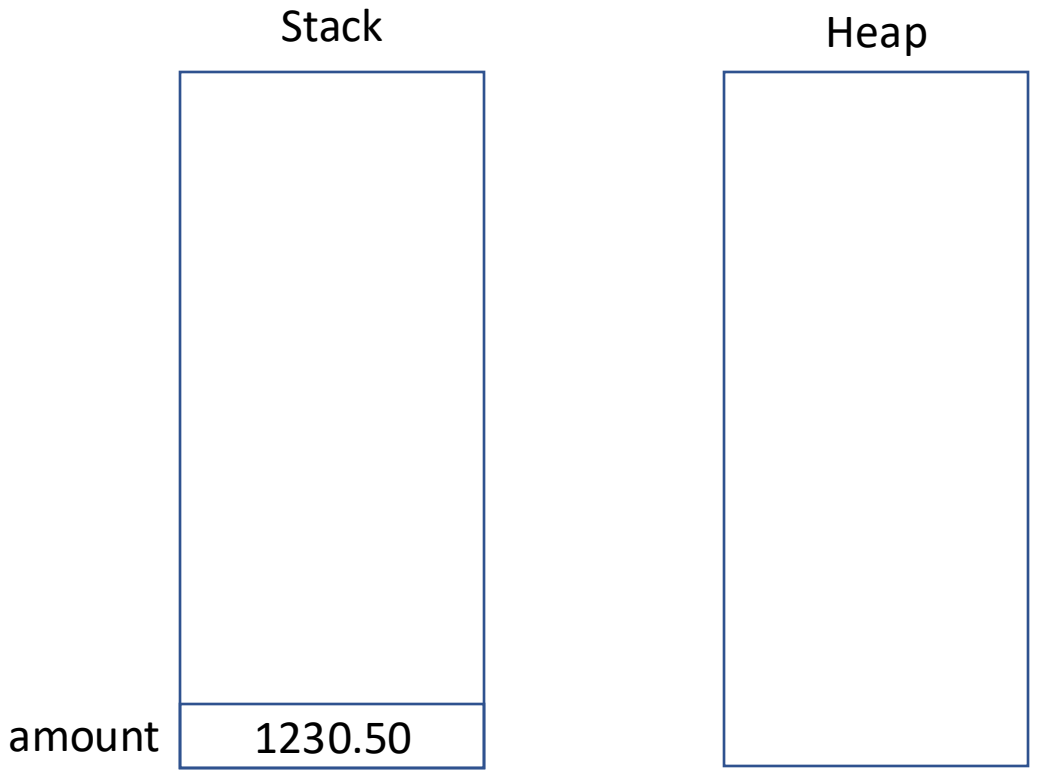
Example 1: double and int

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



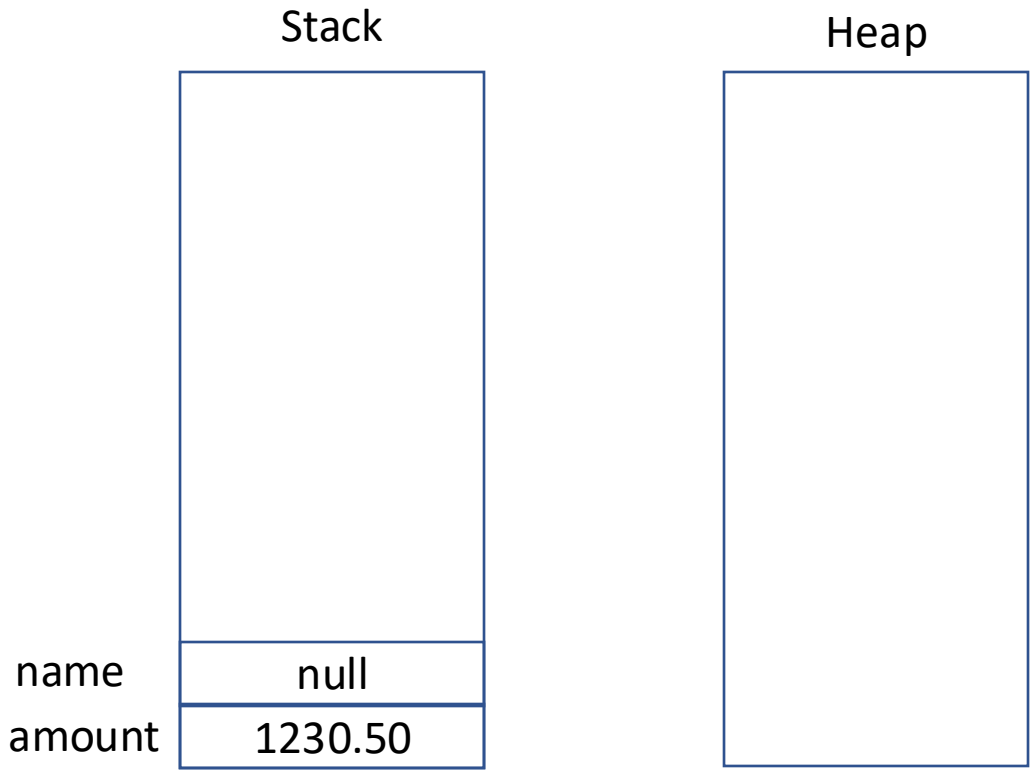
Example 2: double and String

```
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

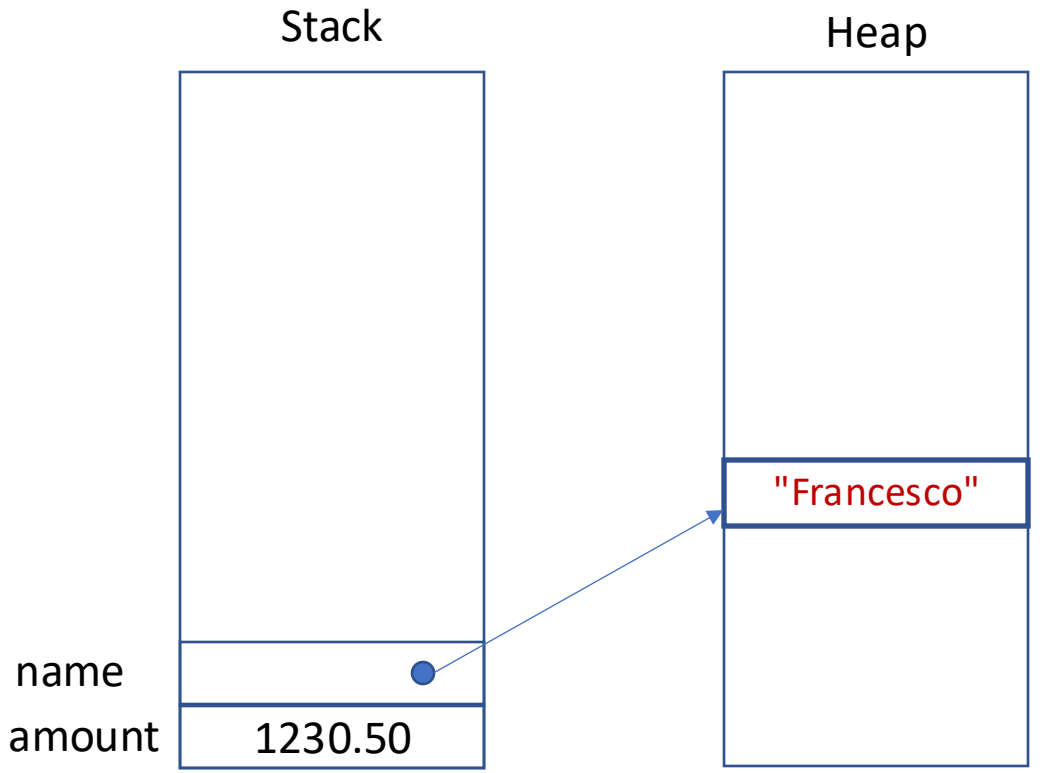
null means that no object is referenced

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

Example 2: double and String

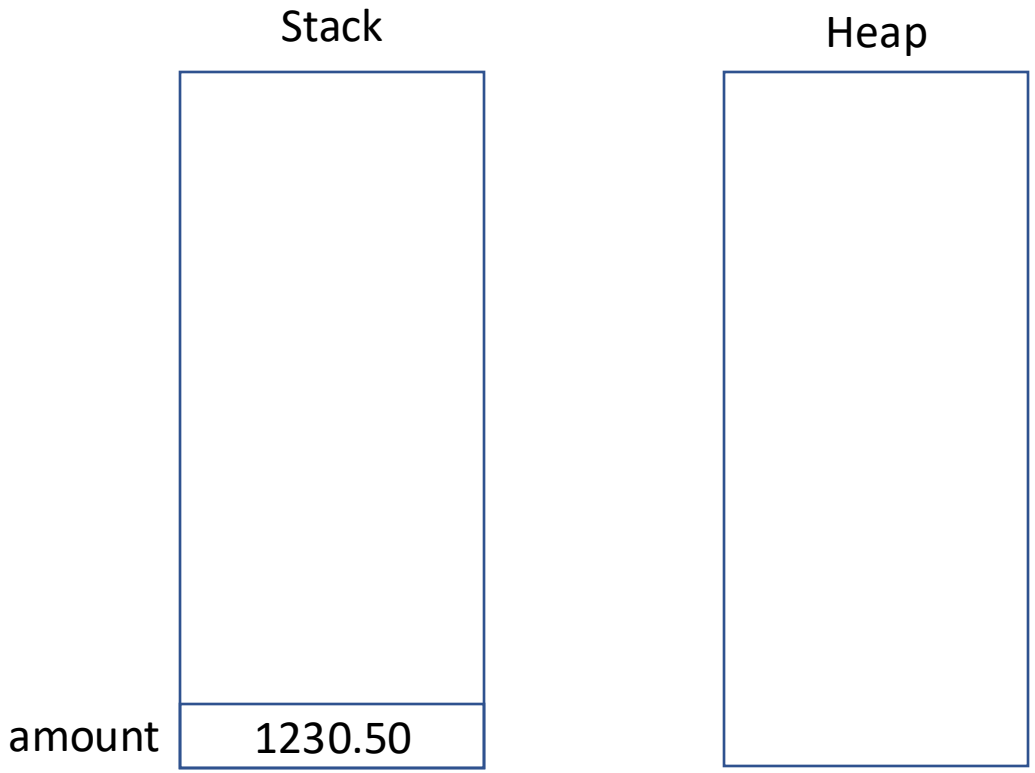
```
class Program
{
    public static void main()
    {
        double amount = 1230.50;
        String name = "Francesco";
    }
}
```

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



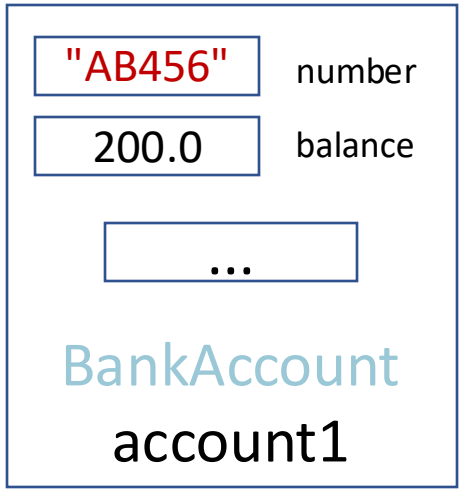
Example 3: double and *BankAccount*

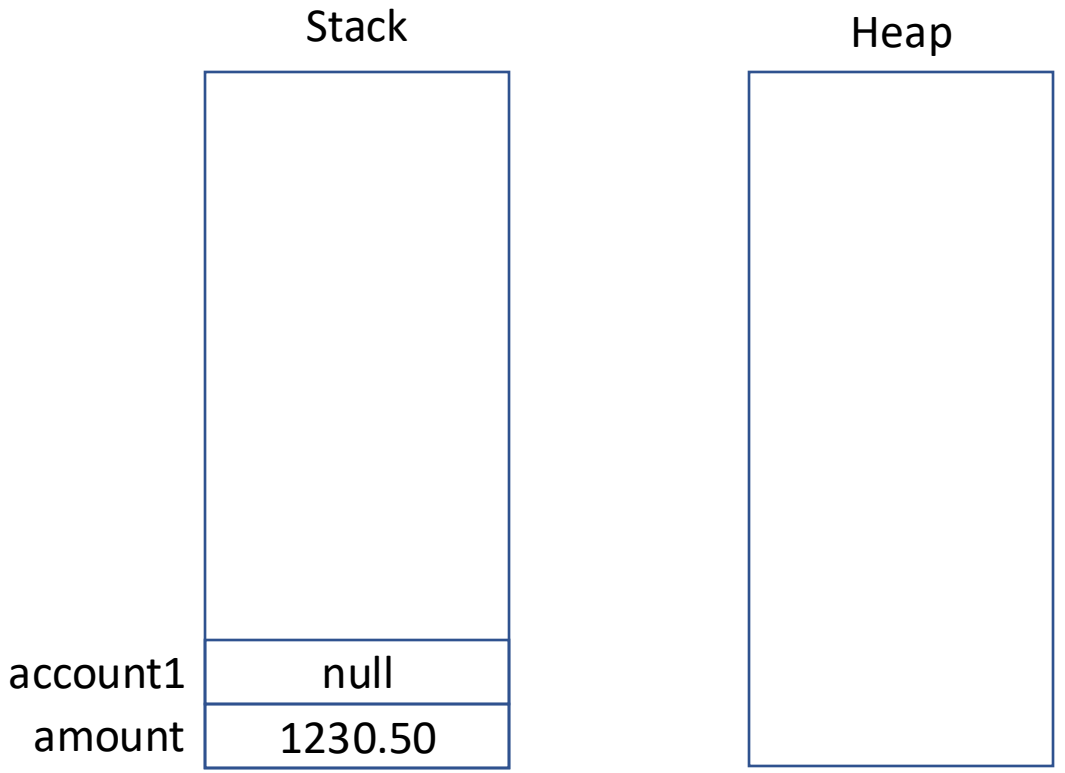
```
class Program
{
    public static void main()
    {
        double amount = 1230.50;


    }
}
```



Example 3: double and *BankAccount*

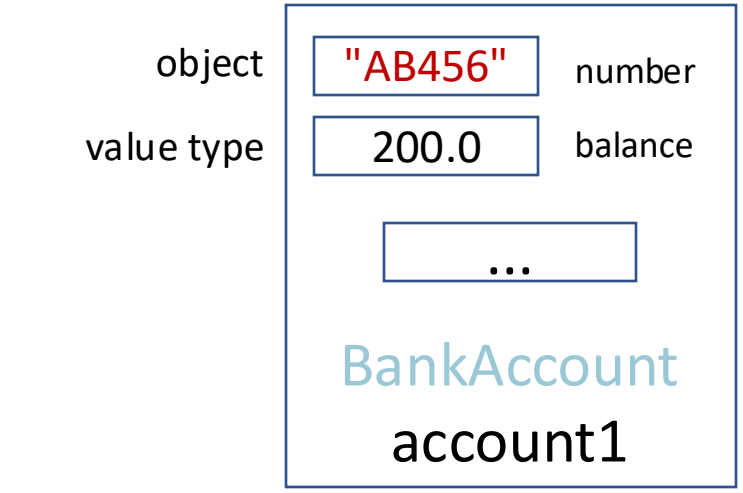
```
class Program
{
   public static void main()
   {
      double amount = 1230.50;
      BankAccount  account1;

   }
}
```

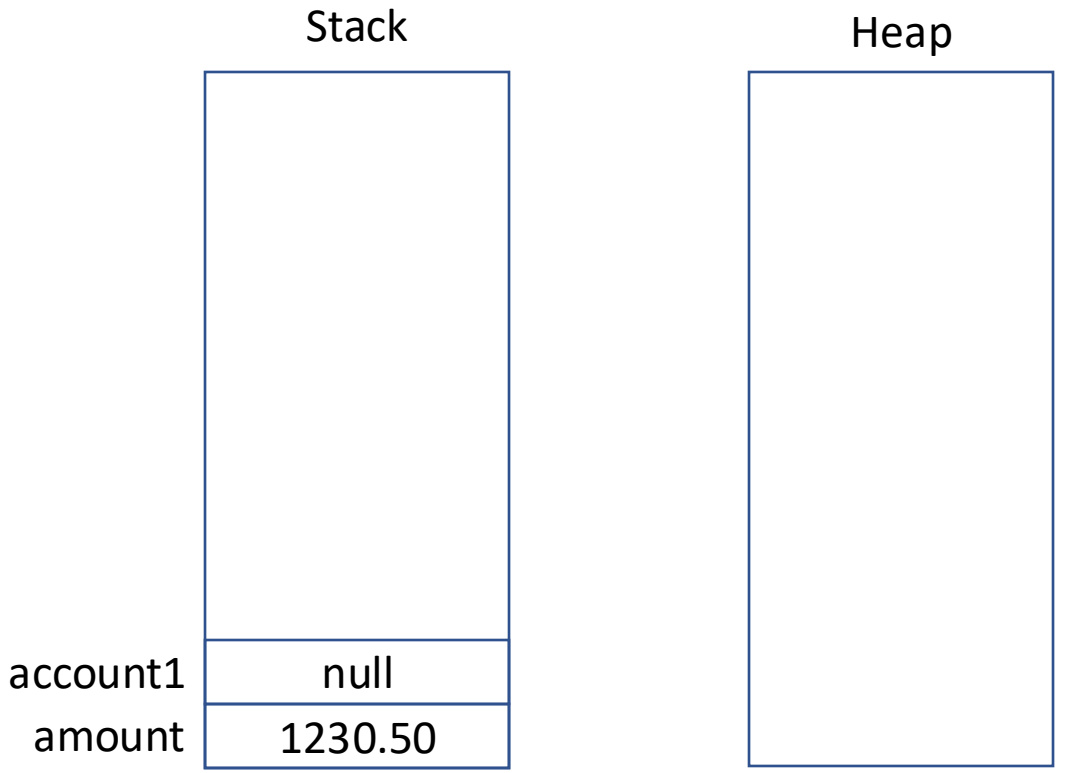




Example 3: double and *BankAccount*

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

   }
}
```





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

Question

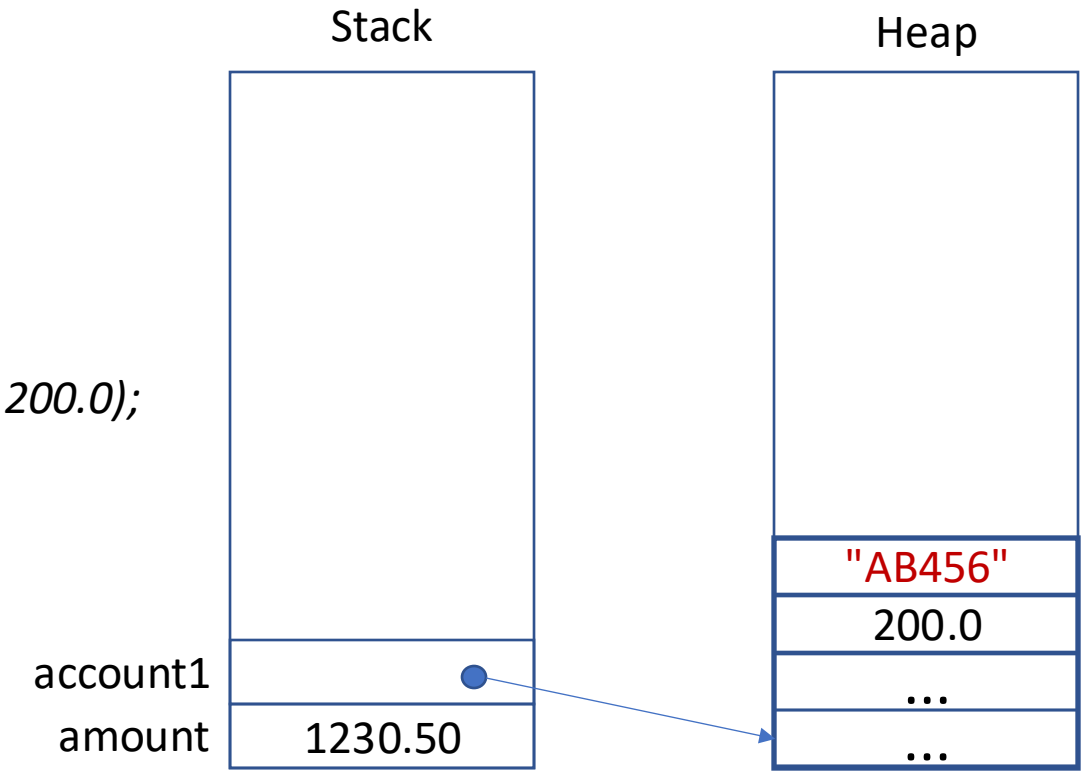
Where will the *value type attributes* be stored when the object is created?

Answer on PollEveryWhere

https://pollev.com/francescotusa

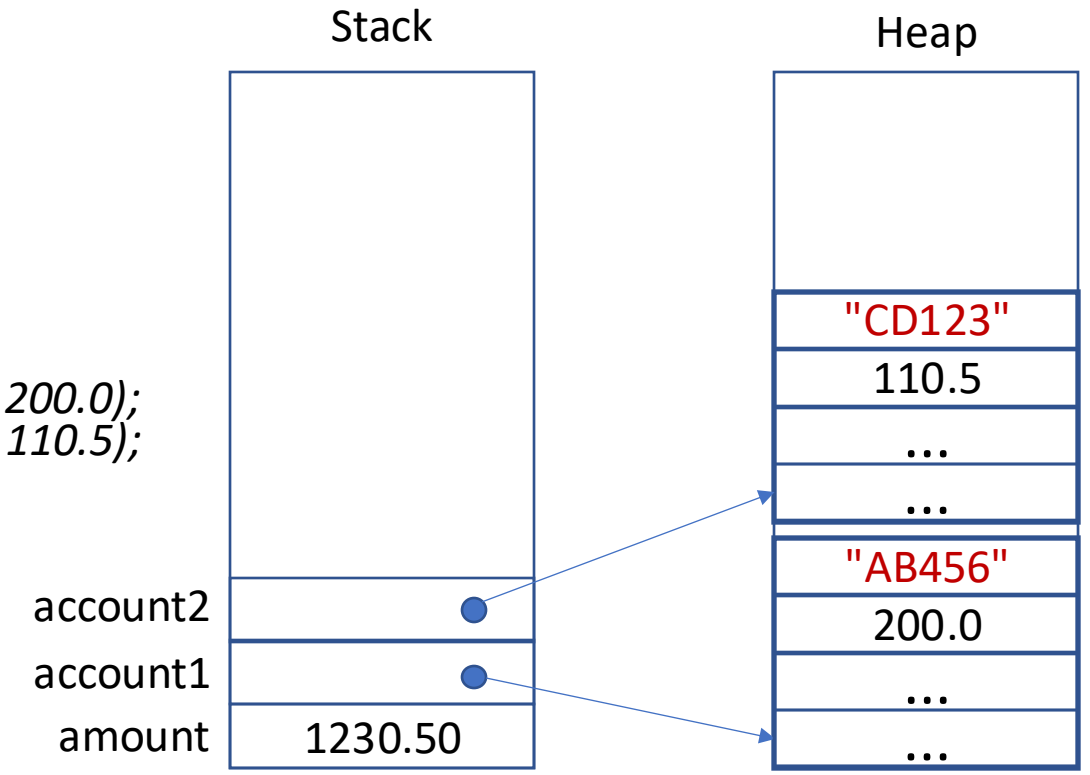
Example 3: double and *BankAccount*

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

   }
}
```



Example 3: double and *BankAccount*

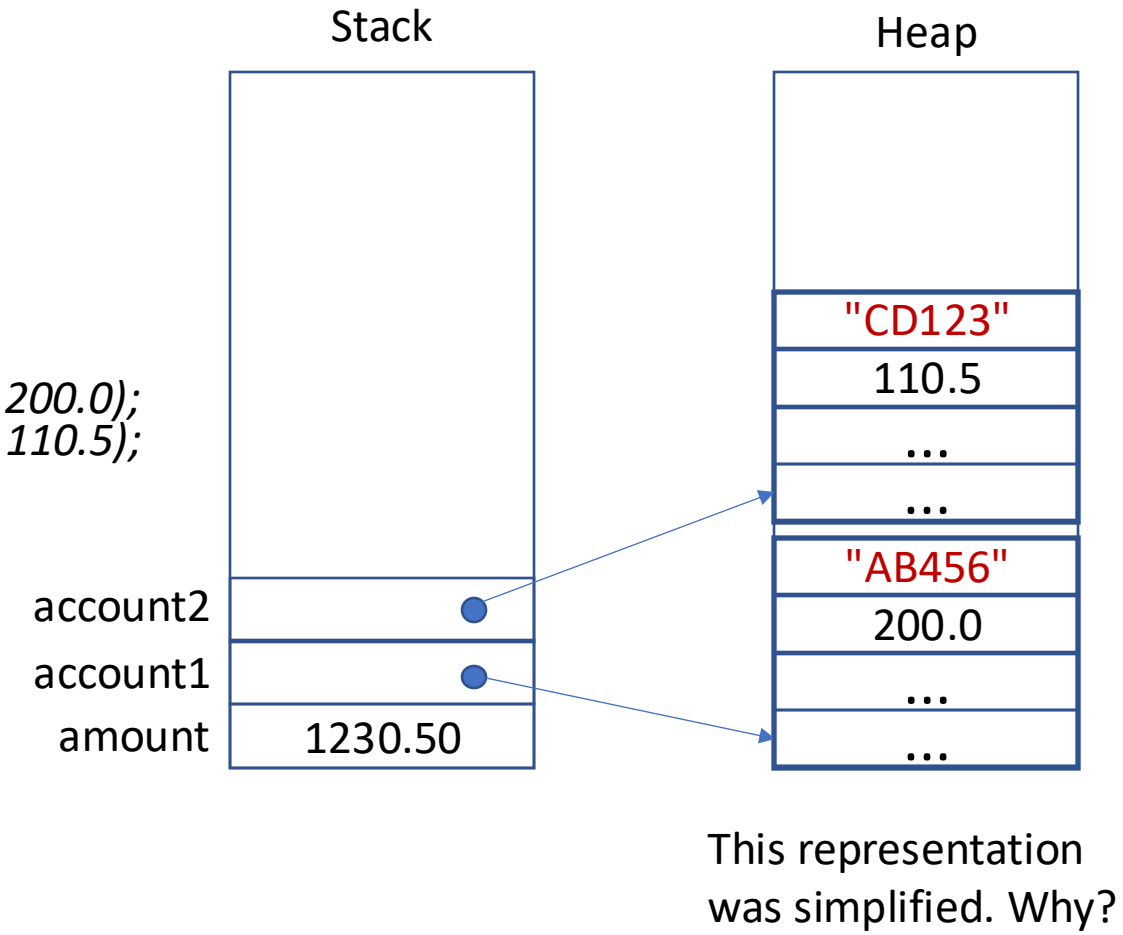
```
class Program
{
   public static void main()
   {
      double amount = 1230.50;
      BankAccount  account1 = new BankAccount("AB456", 200.0);
      BankAccount  account2 = new BankAccount("CD123", 110.5);
   }
}
```



balance is a value type (double) representing an attribute of an object; it is stored with the object on the heap

Example 3: double and *BankAccount*

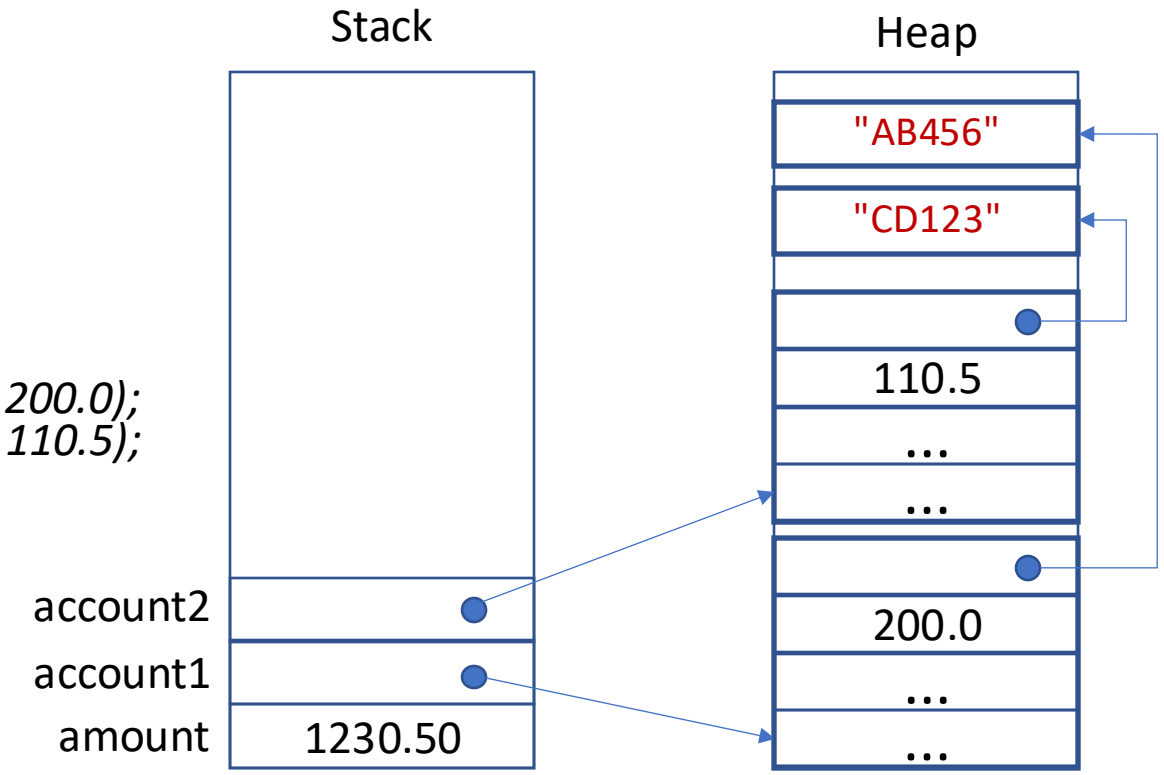
```
class Program
{
   public static void main()
   {
      double amount = 1230.50;
      BankAccount  account1 = new BankAccount("AB456", 200.0);
      BankAccount  account2 = new BankAccount("CD123", 110.5);
   }
}
```



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);
   }
}
```

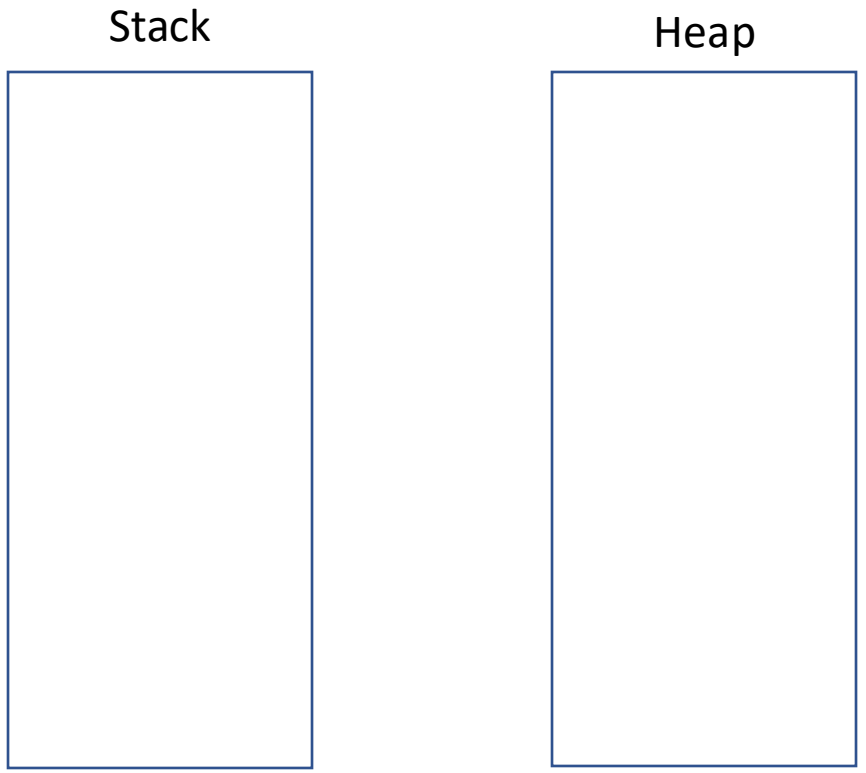


the attribute *number* of *BankAccount* is a reference type (String)

its content may be stored somewhere else on the heap

Example 4: *int[]*

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


    }
}
```



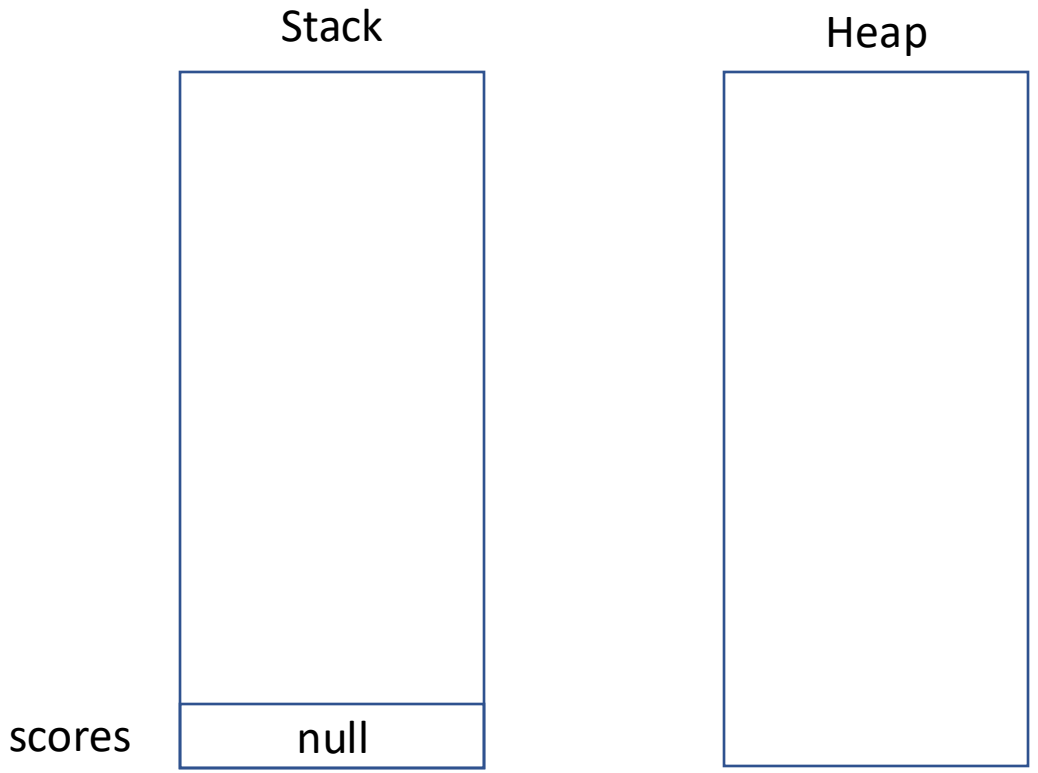
scores is a local array of *primitive* (value) types:
stack or *heap*?

Example 4: *int[]*

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


    }
}
```

scores is a local reference type variable allocated on the stack

the default value is *null* (not initialised)



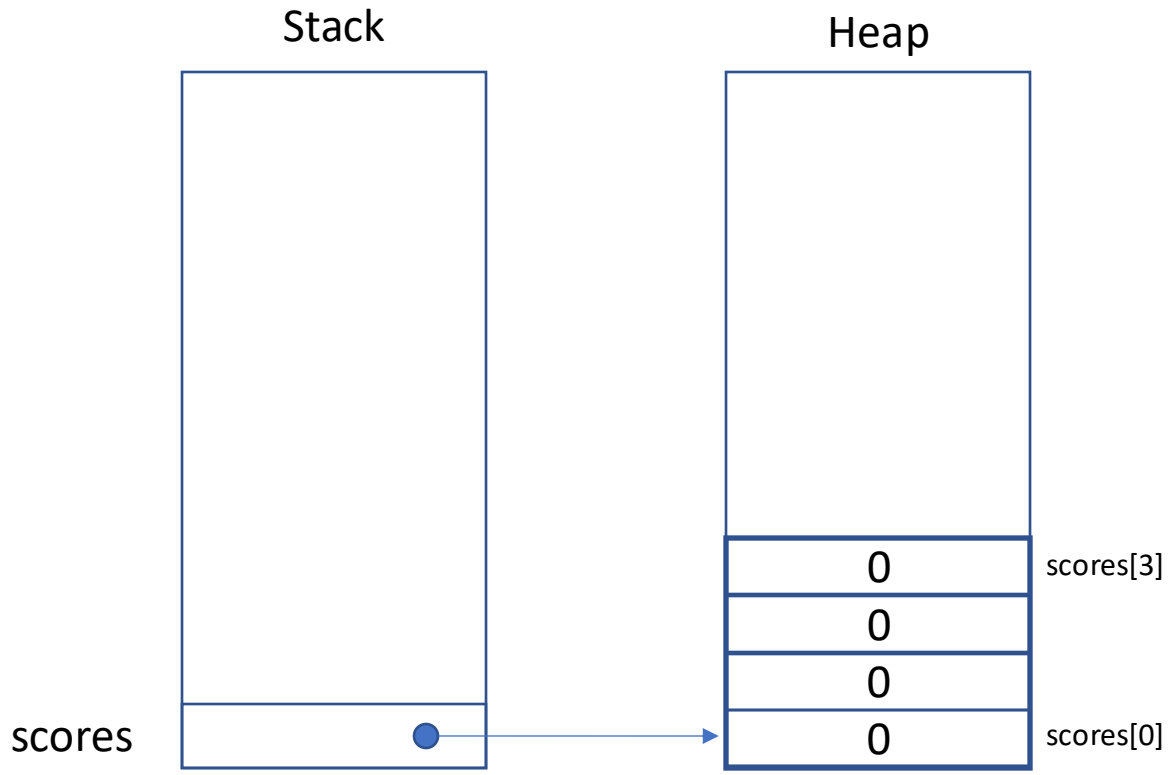
Example 4: *int[]*

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


   }
}
```

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

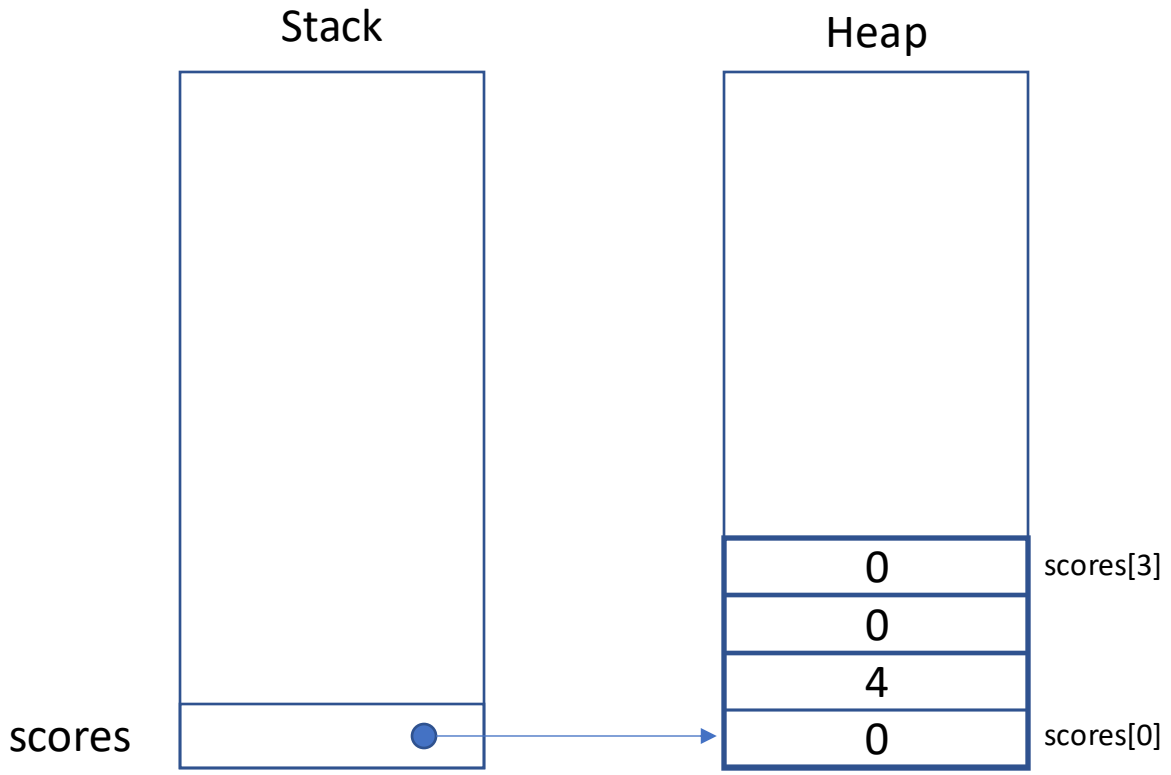
the reference is assigned to *scores*

the elements of the array are initialised to 0



Example 4: *int[]*

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

    }
}
```

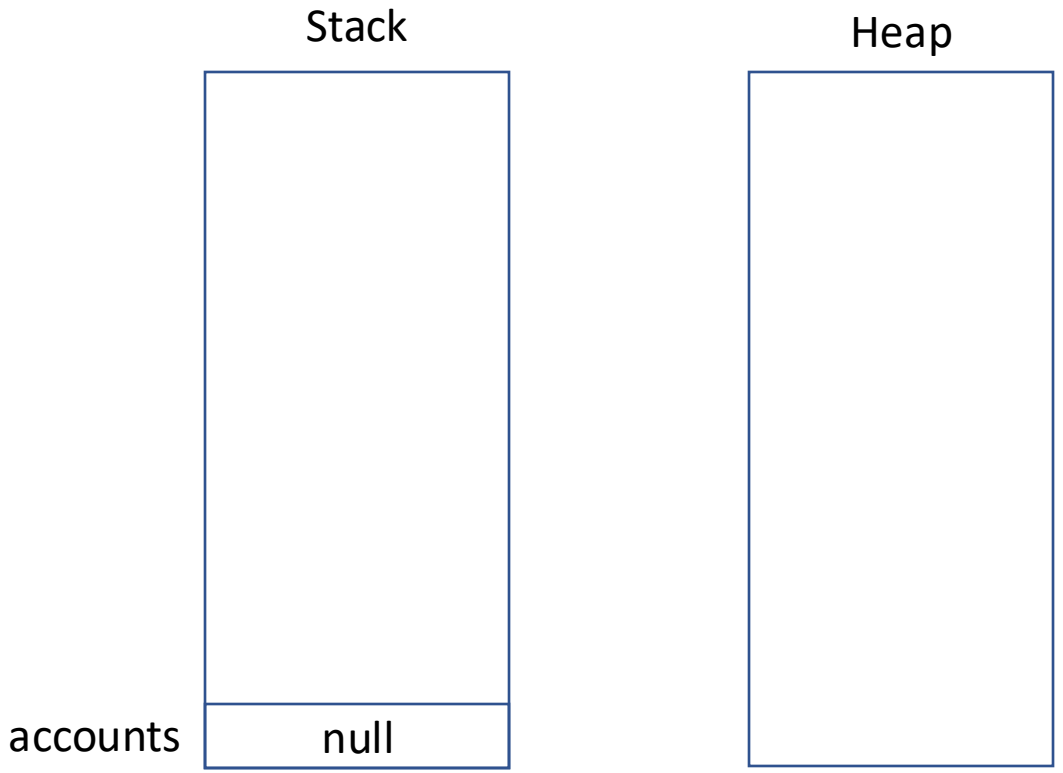


Example 5: *BankAccount[]*

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


   }
}
```

accounts is a reference type variable allocated on the stack

default value is *null* (not initialised)



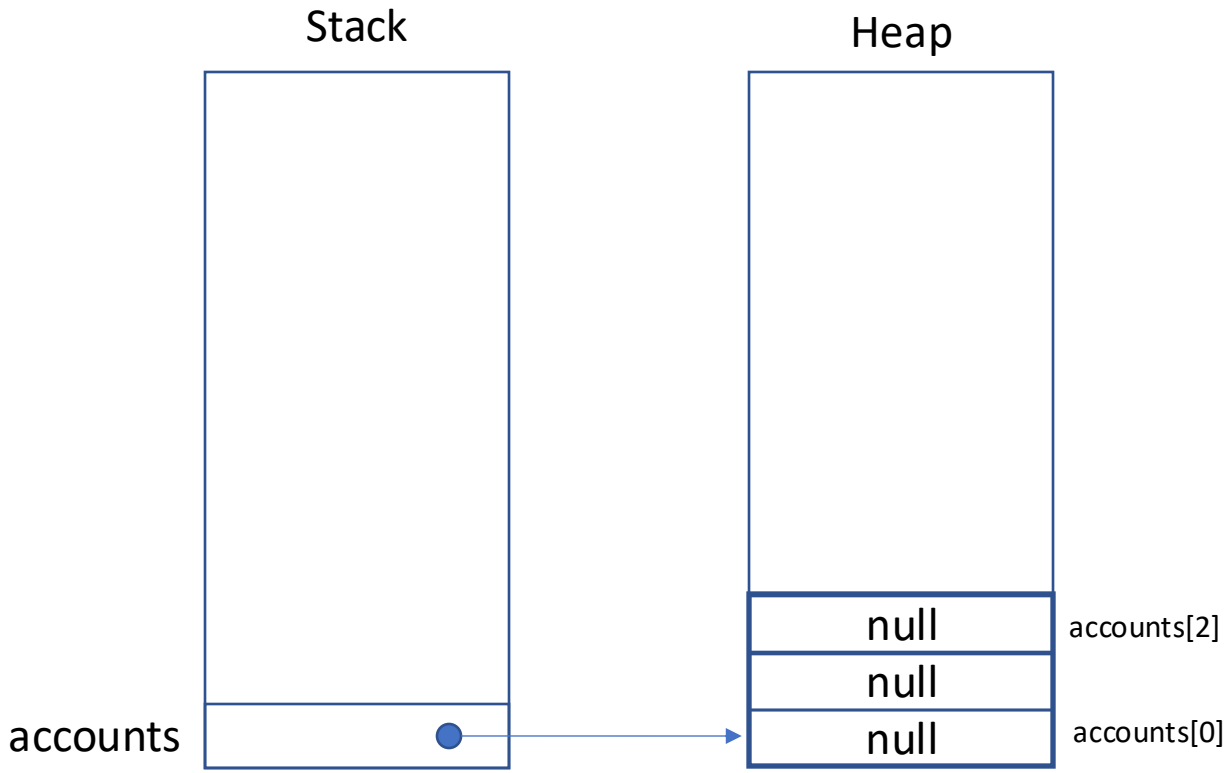
Example 5: *BankAccount[]*

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


   }
}
```

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

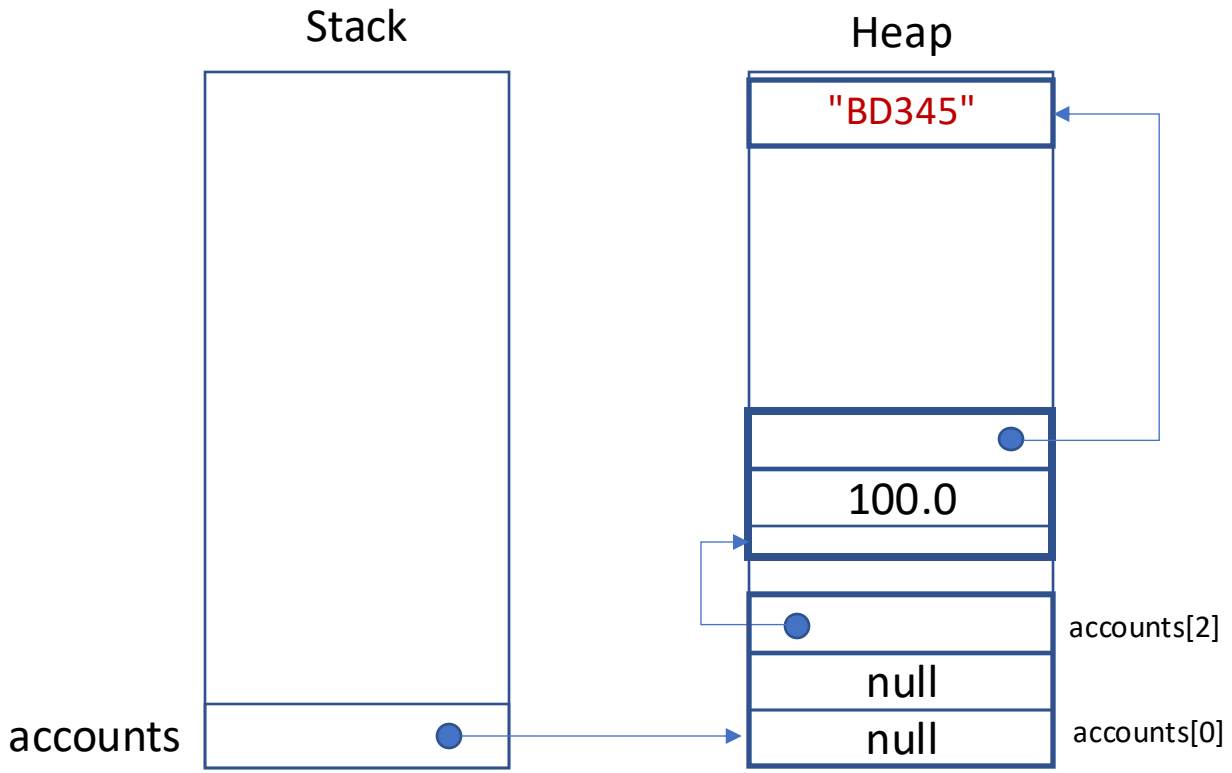
the reference is assigned to *accounts*

the elements of the array are initialised to *null*



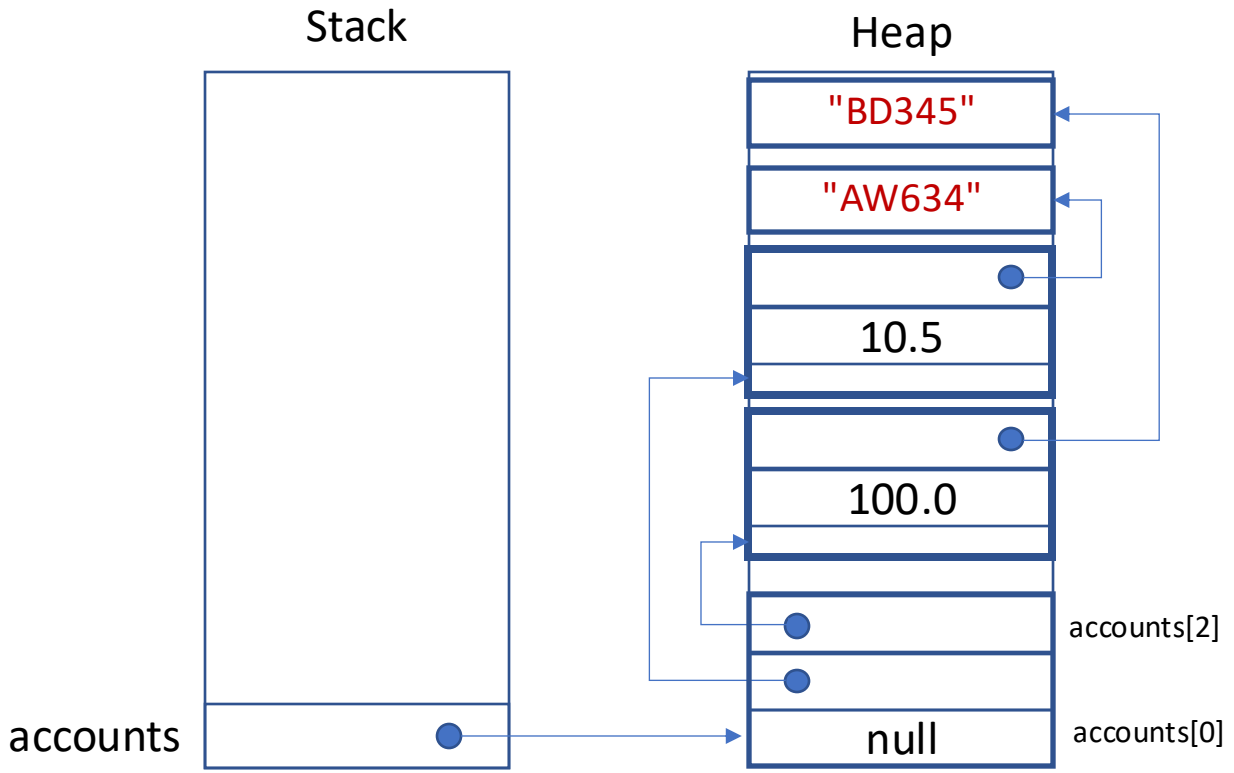
Example 5: *BankAccount[]*

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

    }
}
```



Example 5: *BankAccount[]*

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



Memory stack and heap: summary

- *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 even if of *primitive types*
  - The **reference** variable for that object is allocated on the ***stack***

Reflect on the following

- 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

- Value Types and Reference Types
  - Definition
- Stack and Heap Memory
  - Concepts and Introduction
  - Usage During Method Invocation
- Value Types and Reference Types
  - Memory Allocation Examples
  - **Assignment and Equality Check**
  - Parameter Passing During Method Invocation

The == operator

```
class Program
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = 10;
        if (a == b)
            System.out.println("a is equal to b");



    }
}
```

The == operator

```
class Program
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = 10;
        if (a == b)
            System.out.println("a is equal to b");

        BankAccount account1 = new BankAccount("AB456", 200.0);
        BankAccount account2 = new BankAccount("AB456", 200.0);
        if (account1 == account2)
            System.out.println("account1 is equal to account2");
    }
}
```

Question

What will the previous program print?

Answer on PollEveryWhere

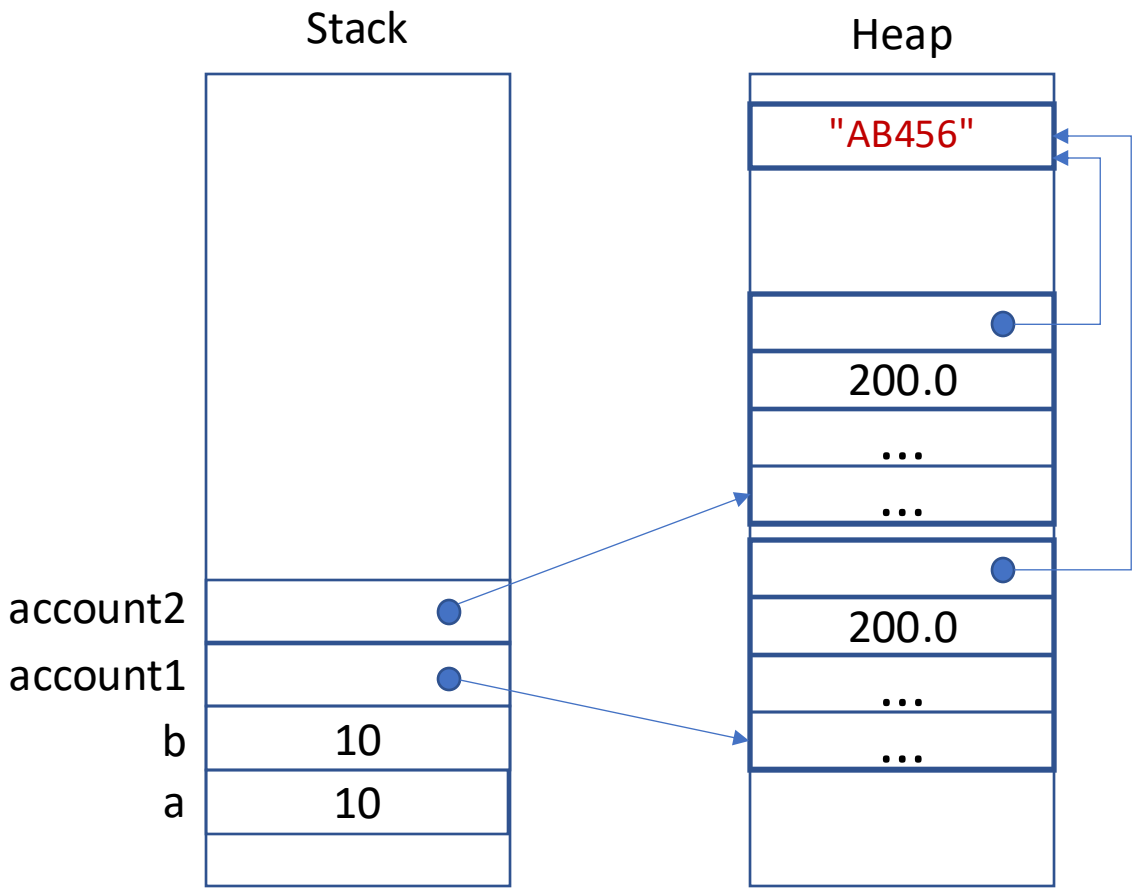
https://pollev.com/francescotusa

Answer

a and *b* are primitive int **value** type variables

They contain the actual values (10 and 10)

'==' will test those values
value equality



Answer

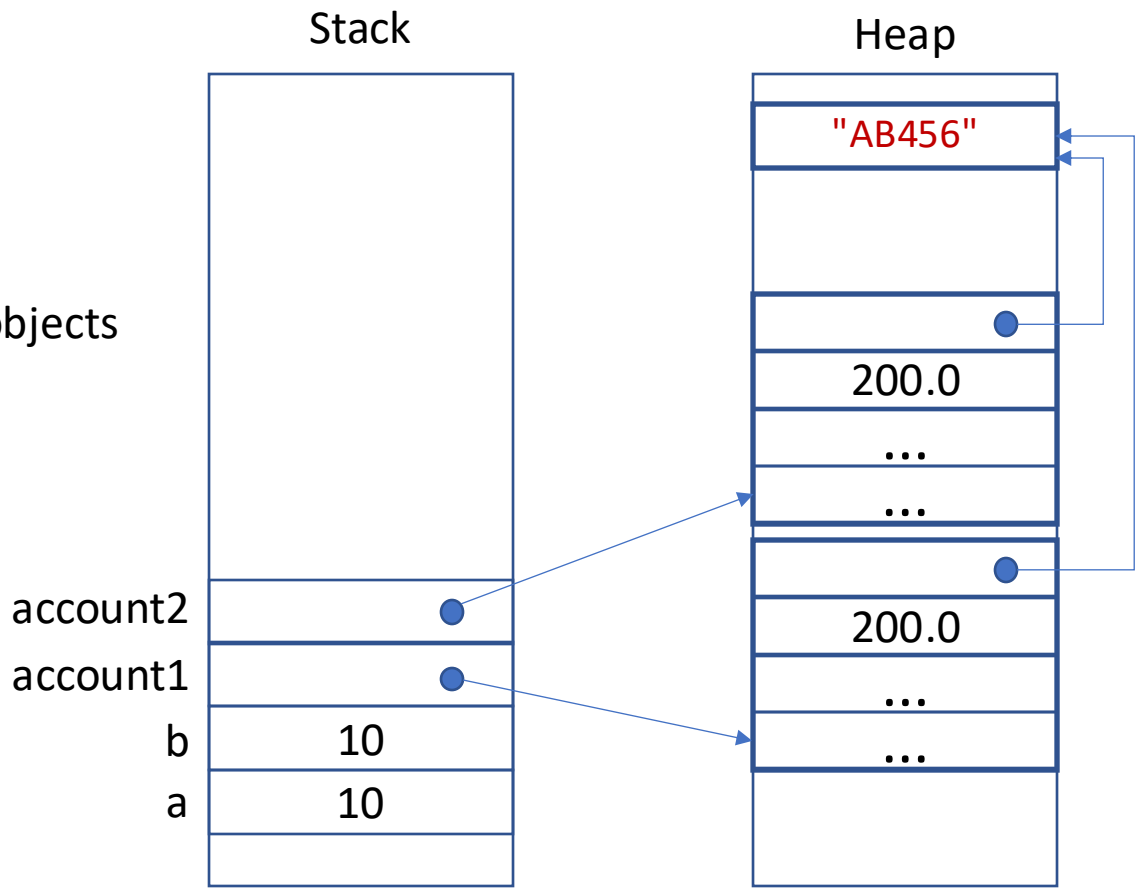
account1 and *account2* are **reference** type variables

They contain the reference (address) of the actual objects

'==' will compare those references, i.e., addresses
reference equality

Those addresses are **different**!

The *equals* method (later) should be used for
value equality between objects

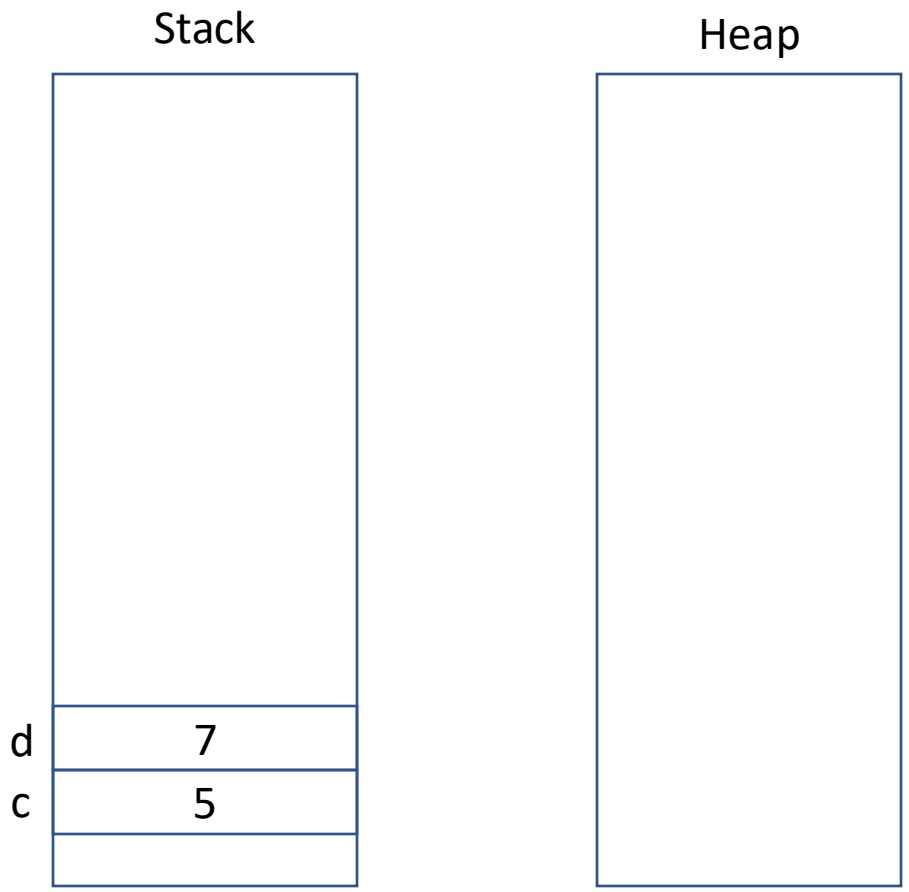


The = operator

```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```

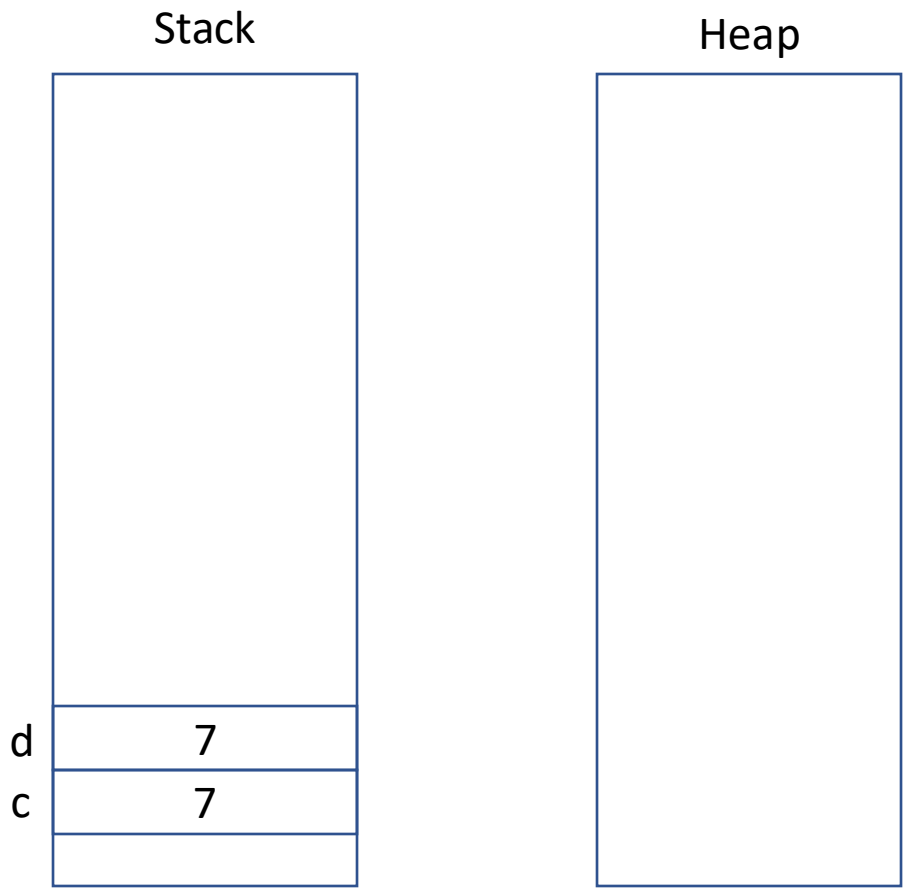
Answer

```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```



Answer

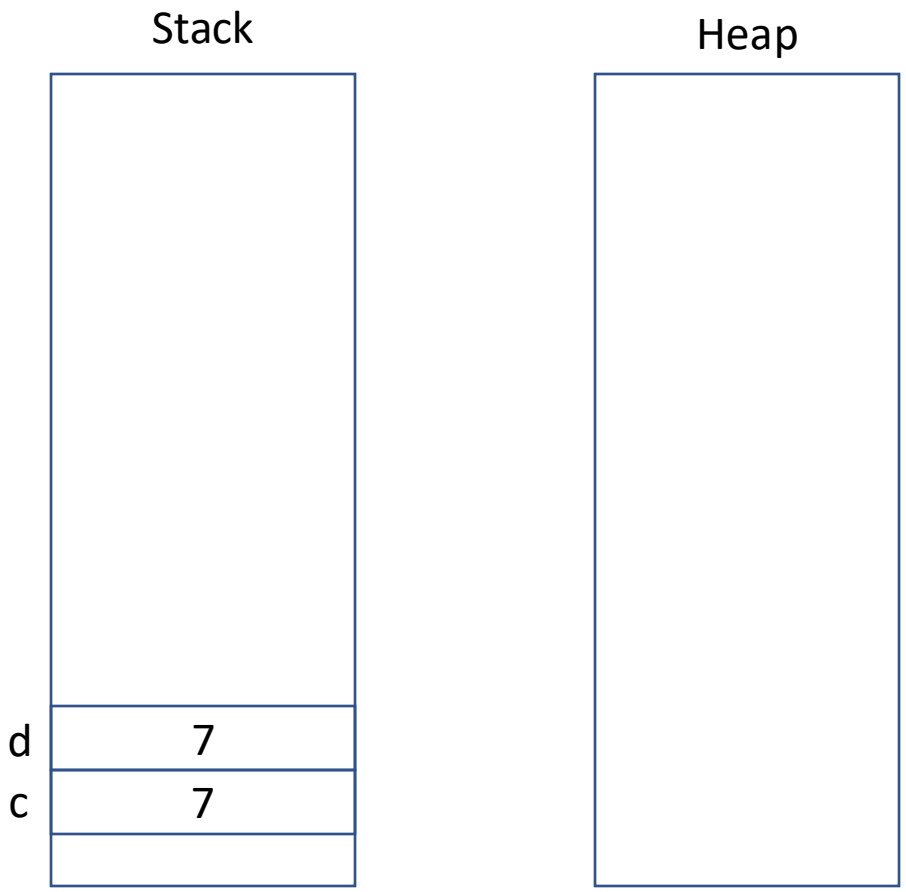
```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```



Answer

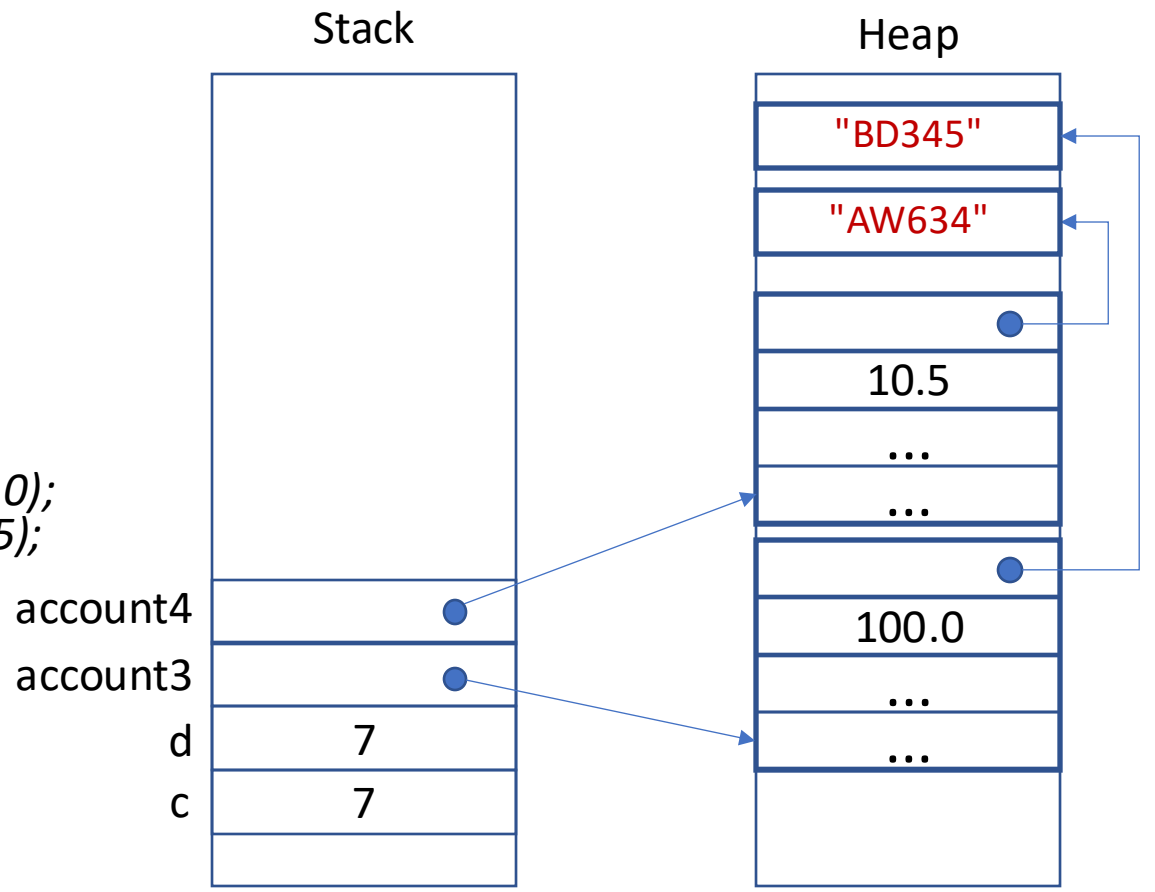
```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```

output:
7
7



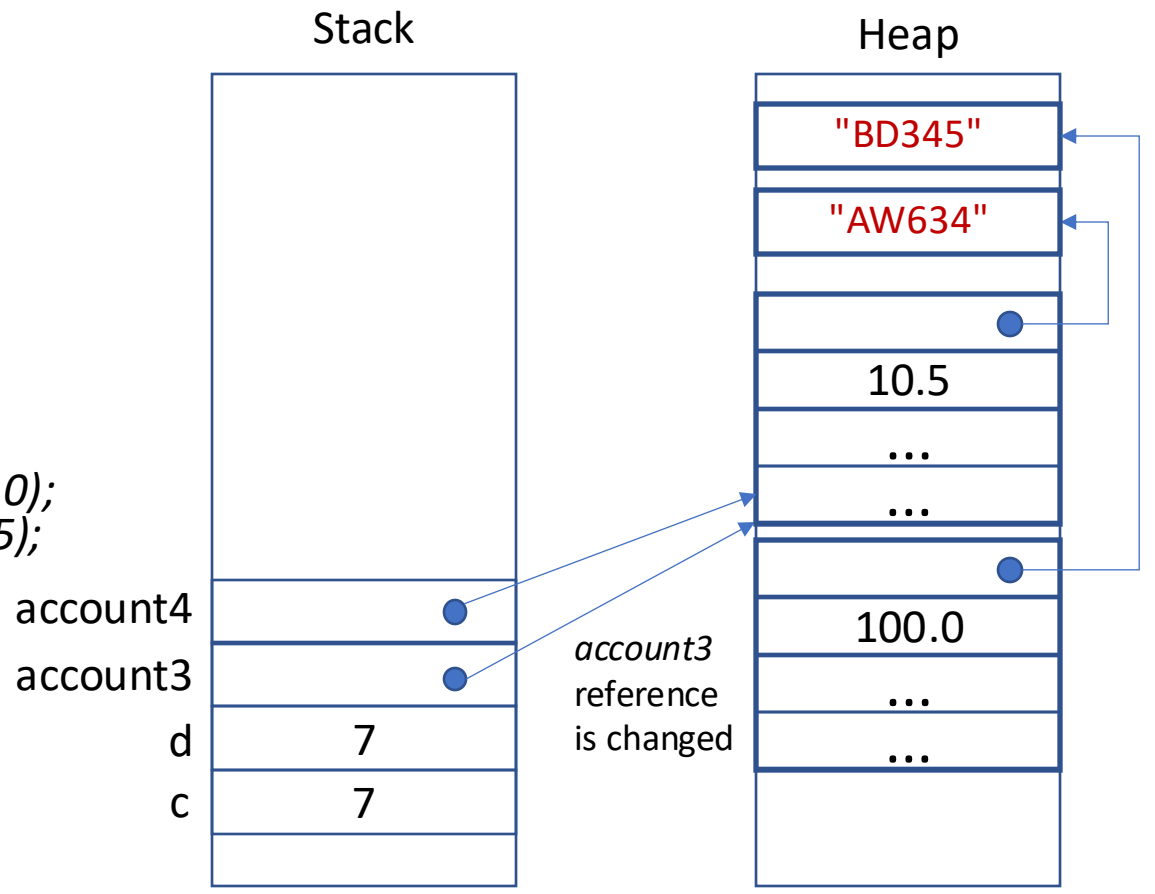
Answer

```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```



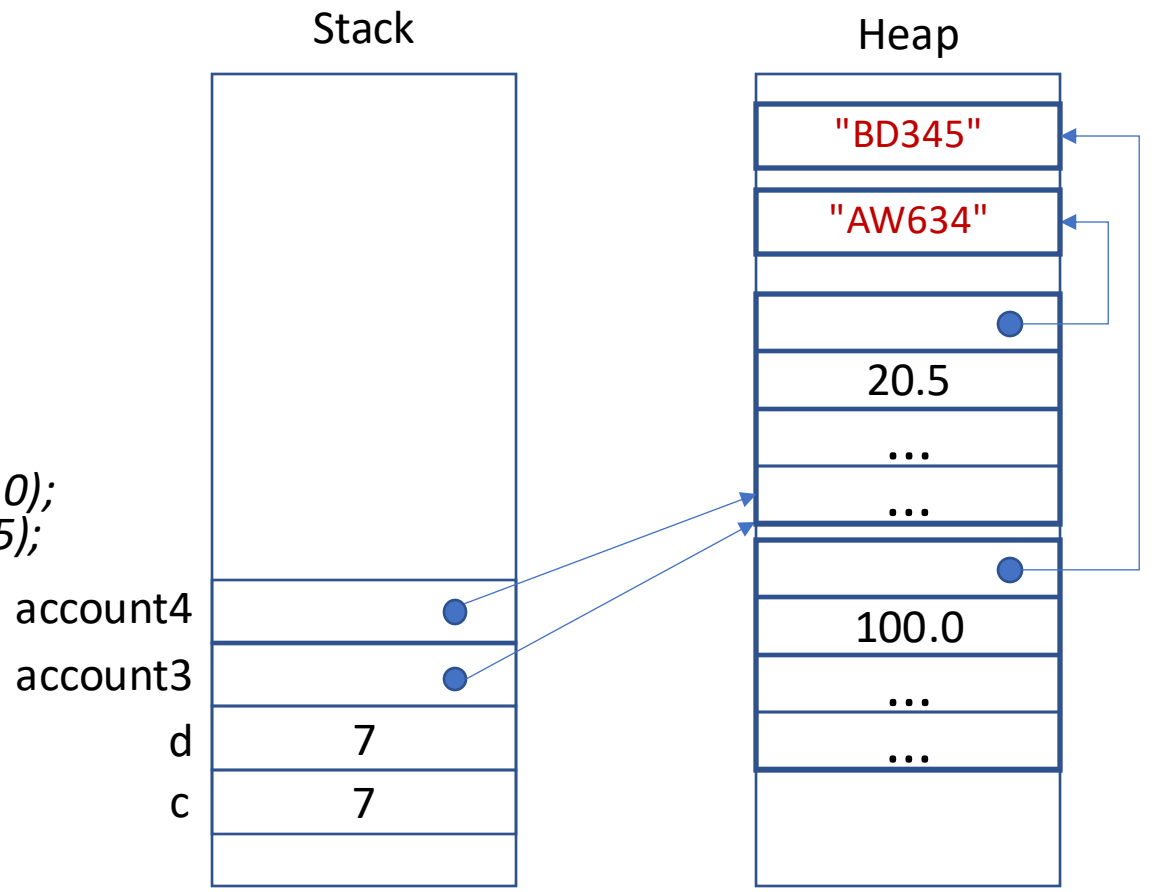
Answer

```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```



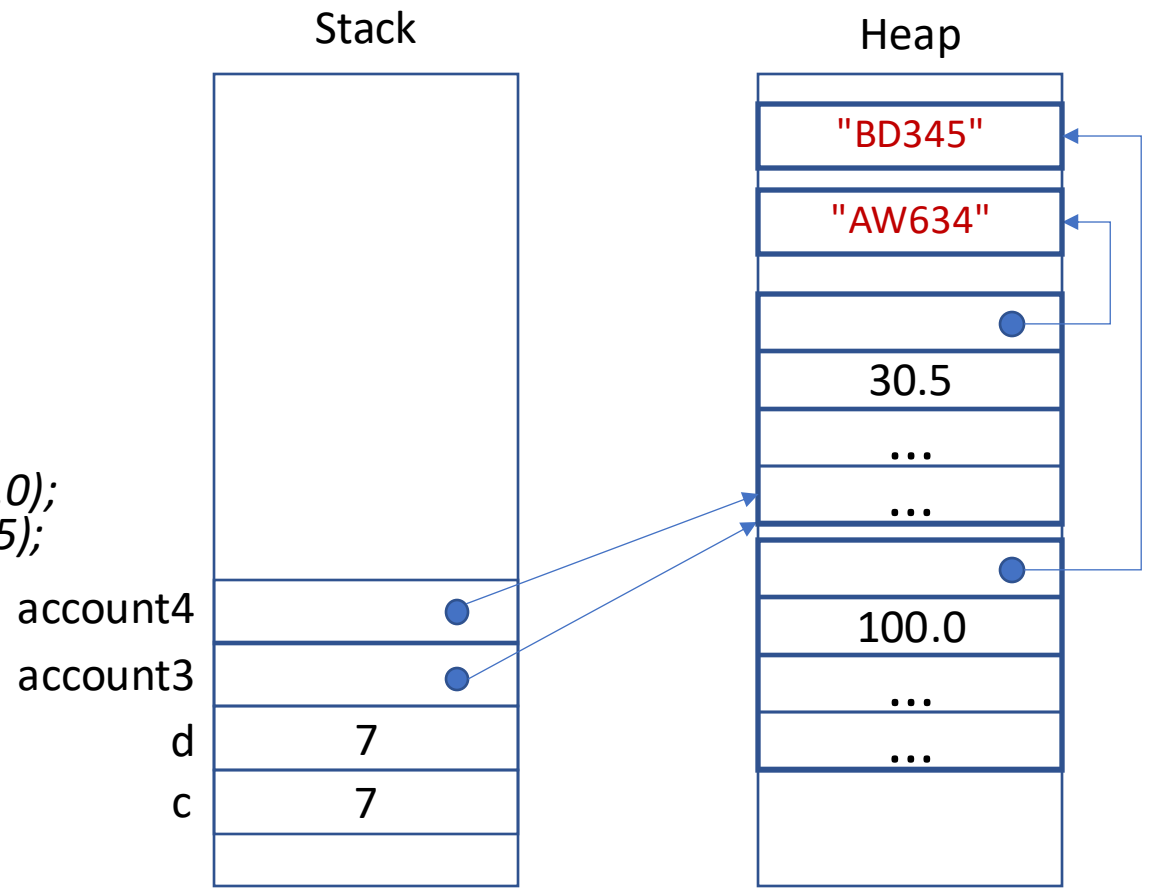
Answer

```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```



Answer

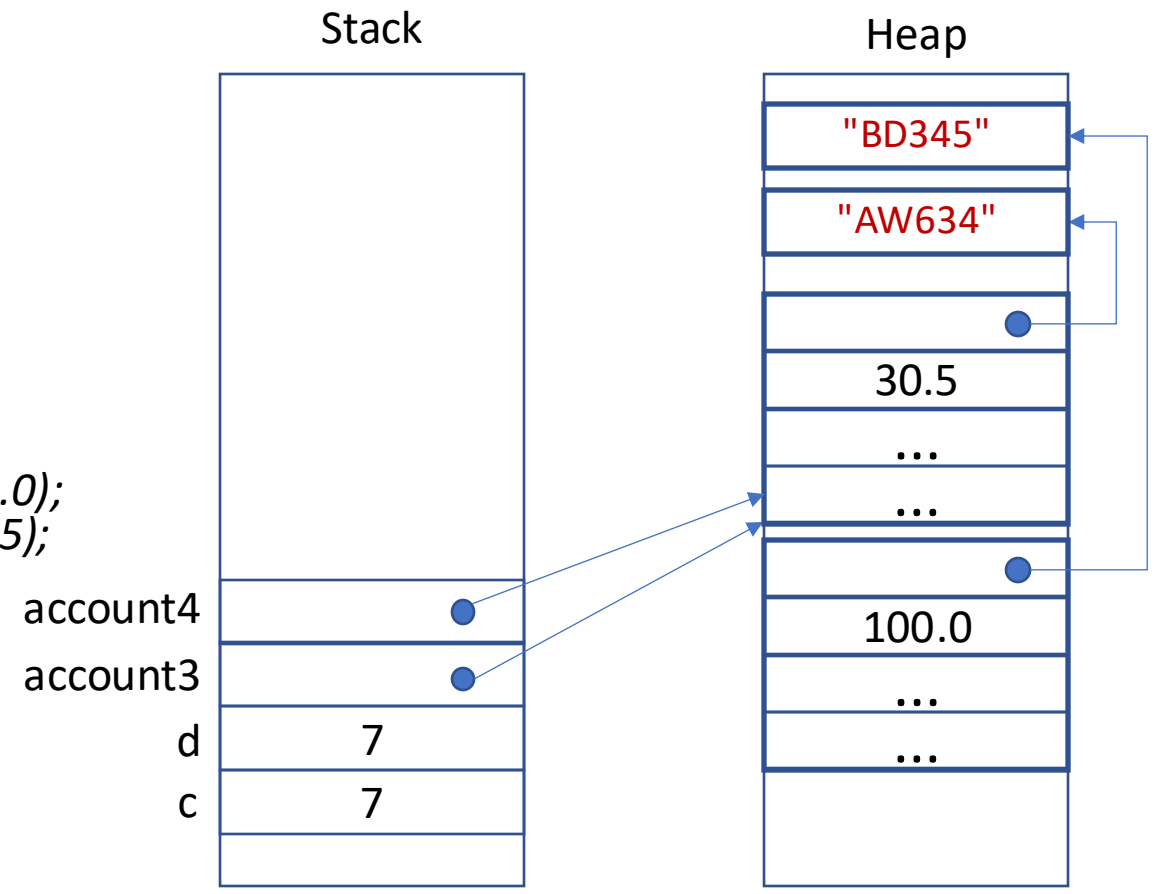
```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```



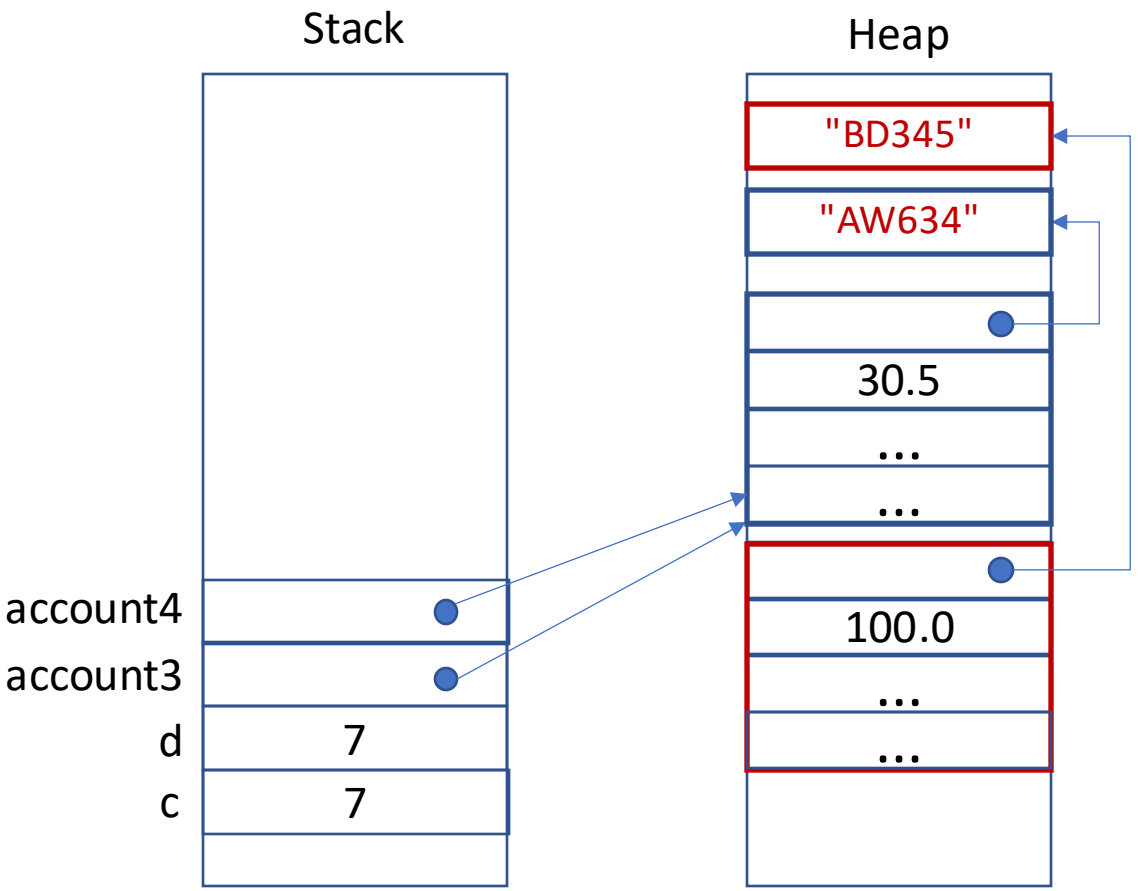
Answer

```
class Program
{
    public static void main(String[] args)
    {
        int c = 5;
        int d = 7;
        c = d;
        System.out.println(c);
        System.out.println(d);

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

        System.out.println(account3.getBalance());
        System.out.println(account4.getBalance());
    }
}
```

output:
30.5
30.5



Garbage Collection



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

The Garbage Collector (GC) of the JVM (Java Virtual Machine) will mark it as "free"

Garbage Collection

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

Outline

- Value Types and Reference Types
  - Definition
- Stack and Heap Memory
  - Concepts and Introduction
  - Usage During Method Invocation
- Value Types and Reference Types
  - Memory Allocation Examples
  - Assignment and Equality Check
  - **Parameter Passing During Method Invocation**

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

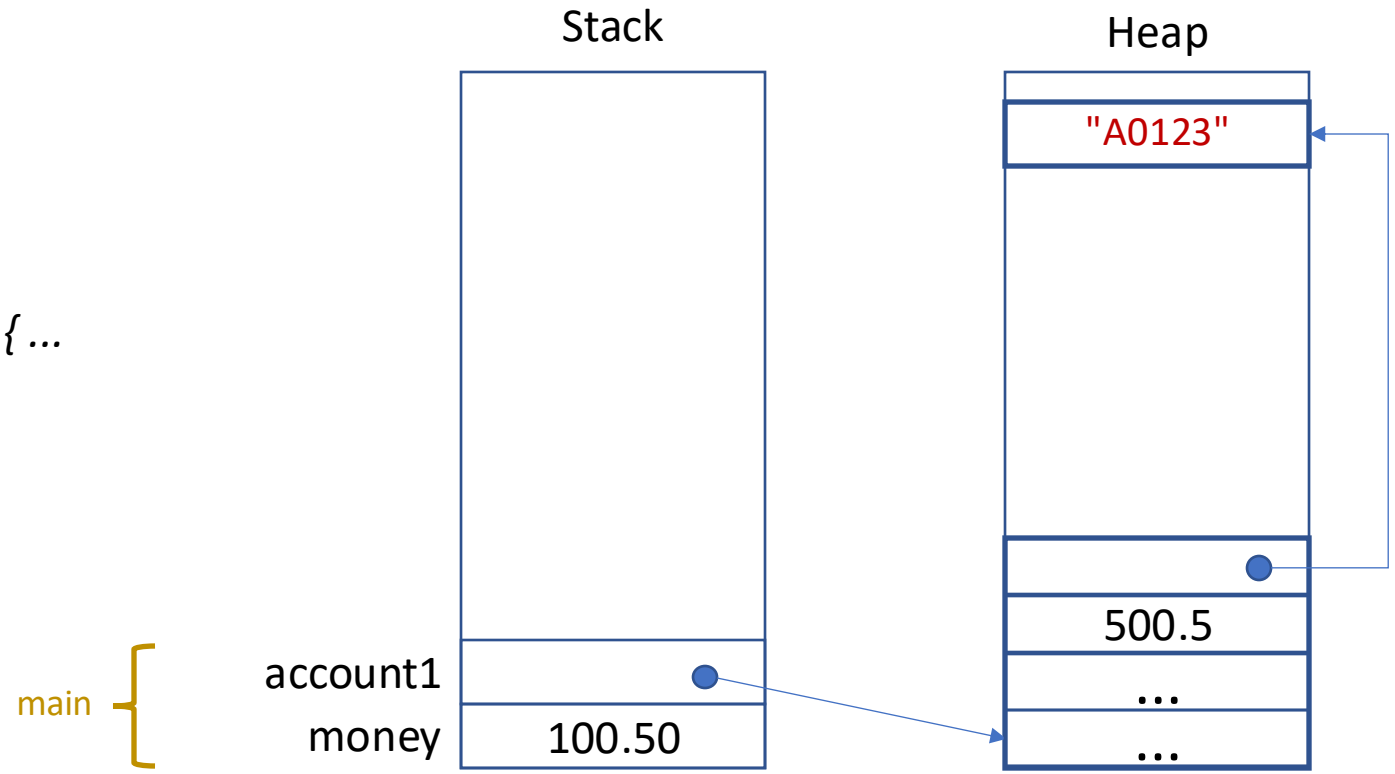
```
class BankAccount
{
    private String number;
    private double balance;

    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%.

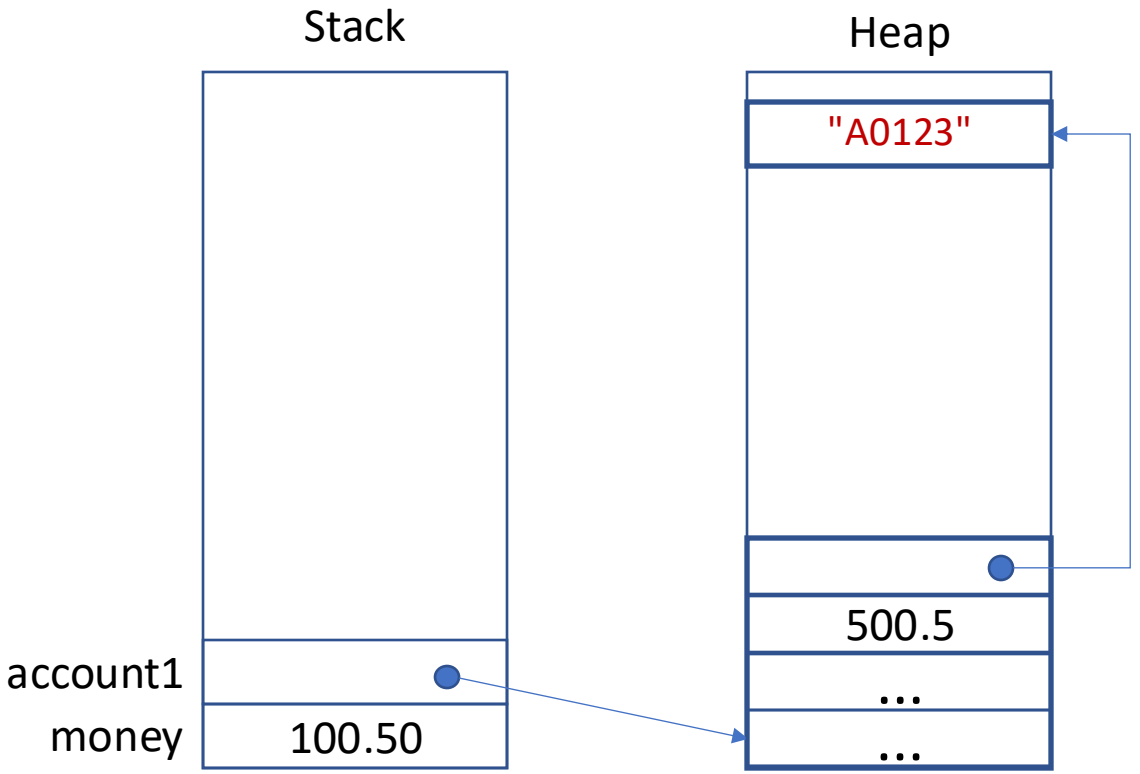
Methods parameters: value types

```
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()
    {
        double money = 100.50;
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        account1.deposit(money);
        ...
}
```



- 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: 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
    }
}
```

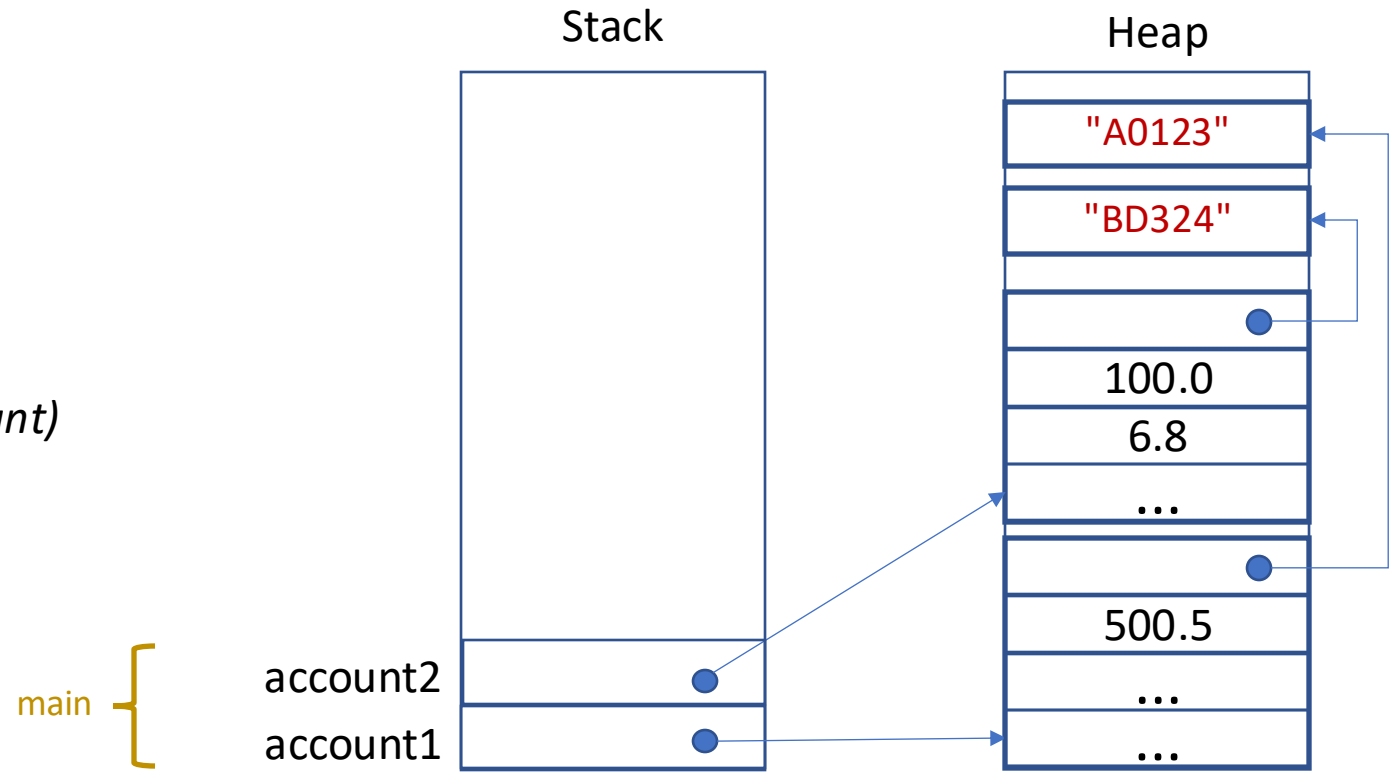
```
class Program
{
    public static void main()
    {
        BankAccount  account1 = new BankAccount("A0123", 500.5);
        SavingAccount  account2 = new SavingAccount("BD324", 100.0, 6.8);

        ...
}
```

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

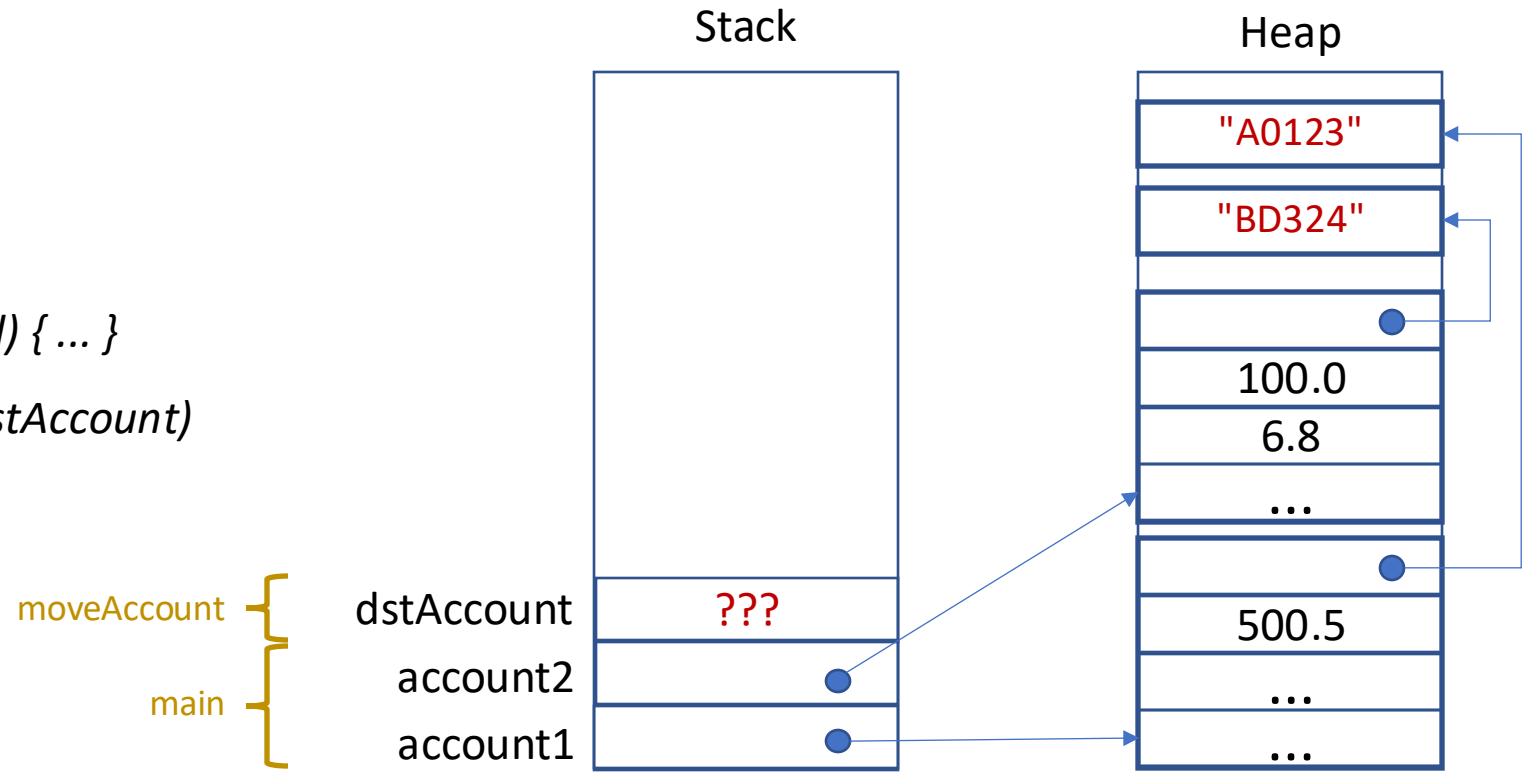
Methods parameters: reference types

```
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?

Questions