Programarea orientata pe obiecte

Curs 4 – addon

-Transferul parametrilor prin referinta

-Cum se compara (corect) variabilele

Reminder (1)

- Orice variabila (declarata in clasa, statica/non statica, declarata in metoda, declarata in antentul metodei) este o referinta la o zona de memorie.
- Orice variabila e de fapt o referinta.
  - Exceptie: Primitivele
- Valoarea implicita pentru o referinta este `null`
- Compilatorul aloca memorie pentru a stoca adresa spre care pointeaza variabila

Reminder (2)

- Alocarea memoriei:
  - Folosind operatorul new(): new BigInteger(“22”);
  - Folosind shortcut-uri:
    - int[] Tab={2,3,4,5,6}
    - String nume=“Popescu”.
  - Apeland o metoda ce contine un new() si returneaza o referinta
    - DriverManager.getConnection(“jdbc:mysql://localhost”)
  - Declarand o primitiva: `int i`

Reminder (3)

- O variabila normala, declarata in clasa, este “lipita” de obiect. Memoria este alocata de new(). (ATENTIE! new MyClass() unde MyClass declara variabila despre care vorbim)
- O variabila declarata in metoda este locala metodei si este alocata pe stiva.
- Daca variabila e referinta, atunci se rezerva pe stiva spatiu pentru o adresa de memorie

Apel prin referinta (1)

- Fie o clasa `Contor` simpla:

```
1   package exam;
2   public class Contor {
3       private Integer value;
4
5       public Contor() {
6           this.value = 0;
7       }
8
9       public Integer getValue() {
10          return value;
11      }
12
13      public void setValue(Integer value) {
14          this.value = value;
15      }
16
17  }
```

Apel prin referinta (2)

- In alta clasa, din acelasi pachet:

```
1   package exam;
2   public class References {
3   void m1(){
4       Contor c1=new Contor();
5       c1.setValue(3);
6       System.out.println("C1 == "+c1.getValue());
7       m2(c1);
8       System.out.println("C1 == "+c1.getValue());
9   }
10  void m2(Contor c){
11      c.setValue(-1);
12      c=new Contor();
13      c.setValue(20);
14
15  }}
```

```
------------ Standard Output -
C1 == 3
C1 == -1
------------ -----+----------
```

Apel prin referinta (3)

- De ce?
- In metoda m2():

```
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);
```

- De ce nu se pastreaza noua valoare (20) in variabila c?
- De ce ramane totusi o setare din m2? (c.setvalue(-1))

WTF?? Ambele instructiuni ar trebui sa mearga

Apel prin referinta (4)

- Sa vedem ce se intampla cu memoria.

- Reminder:
  - Variabilele locale se pastreaza pe stiva.
  - Operatorul new() aloca memorie pe heap si returneaza o adresa de memorie
  - TOATE variabilele sunt de fapt referinte spre zone de memorie.
- Clasa Contor declara o variabila de tip Integer.

Apel prin referinta (5.1)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

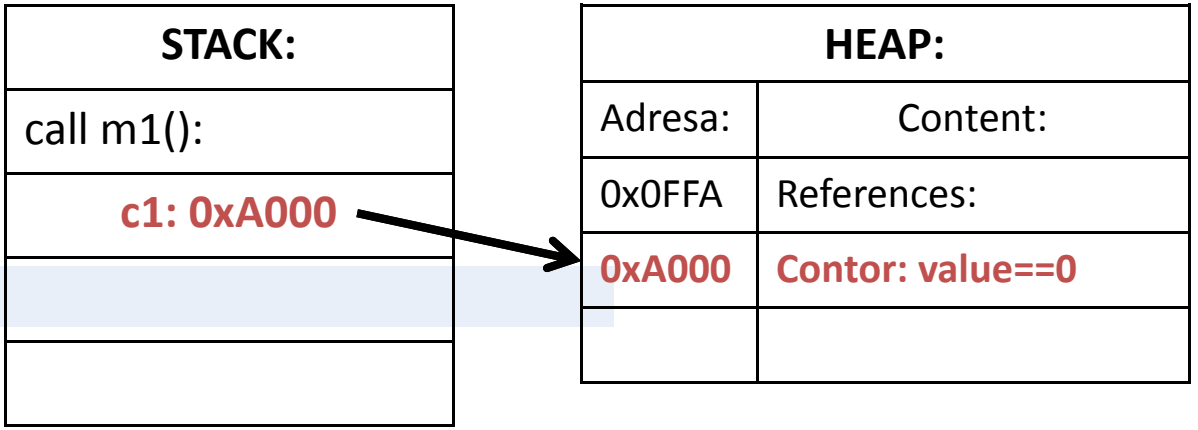
STACK:
call m1():

HEAP:	
Adresa:	Content:
0x0FFA	References:

Apel prin referinta (5.2)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```



Apel prin referinta (5.3)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

STACK:
call m1():
c1: 0xA000

HEAP:	
Adresa:	Content:
0x0FFA	References:
0xA000	Contor: **value==3**

Apel prin referinta (5.4)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

STACK:
call m1():
c1: 0xA000
call m2(c=0xA000)

HEAP:	
Adresa:	Content:
0x0FFA	References:
0xA000	Contor: value==3

Apel prin referinta (5.5)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

STACK:
call m1():
c1: 0xA000
call m2(c=0xA000)
c: 0xA000

HEAP:	
Adresa:	Content:
0x0FFA	References:
0xA000	Contor: value==3

Apel prin referinta (5.6)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

STACK:
call m1():
c1: 0xA000
call m2(c=0xA000)
c: 0xA000

HEAP:	
Adresa:	Content:
0x0FFA	References:
0xA000	Contor: **value==-1**

Apel prin referinta (5.7)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

STACK:
call m1():
c1: 0xA000
call m2(c=0xA000)
c: **0xA0B0**

HEAP:	
Adresa:	Content:
0x0FFA	References:
0xA000	Contor: value==-1
0xA0B0	**Contor: value==0**

Apel prin referinta (5.8)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

STACK:
call m1():
c1: 0xA000
call m2(c=0xA000)
c: 0xA0B0

HEAP:	
Adresa:	Content:
0x0FFA	References:
0xA000	Contor: value==-1
0xA0B0	Contor: **value==20**

Apel prin referinta (5.9)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

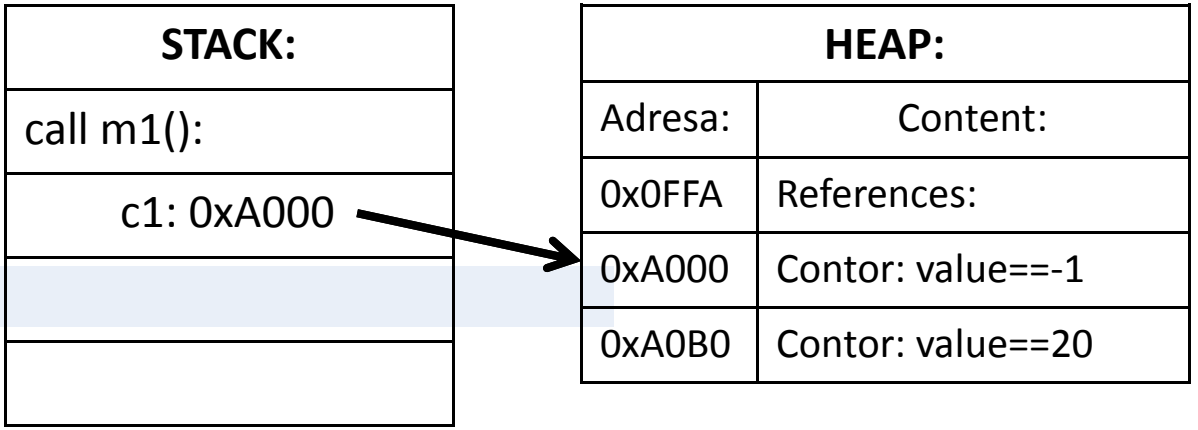
STACK:
call m1():
c1: 0xA000
call m2(c=0xA000)
c: 0xA0B0

HEAP:	
Adresa:	Content:
0x0FFA	References:
0xA000	Contor: value==-1
0xA0B0	Contor: value==20

Apel prin referinta (5.10)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```



c1 refera adresa 0xA000. Acolo, contorul e setat pe -1. ==> se printeaza -1.

Apel prin referinta (5.11)

```
package exam;
public class References {
void m1(){
    Contor c1=new Contor();
    c1.setValue(3);
    System.out.println("C1 == "+c1.getValue());
    m2(c1);
    System.out.println("C1 == "+c1.getValue());
}
void m2(Contor c){
    c.setValue(-1);
    c=new Contor();
    c.setValue(20);

}}
```

STACK:
call m1():
c1: 0xA000

HEAP:	
Adresa:	Content:
0x0FFA	References:
0xA000	Contor: value==-1
0xA0B0	Contor: value==20

Apel prin referinta (6)

- Daca nu ati inteles, mai reluati odata exemplul, cu atentie!

Compararea variabilelor (1)

- Reminder:
  - Orice variabila, in java, e o referinta la o zona de memorie
  - Exceptie: primitivele.

Compararea variabilelor (2)

- Exceptie:
  - Primitivele.
  - Clasele wrapper (Integer, Float, Char, etc.)
- NU si String, si toate celelalte tipuri de variabile

```
Integer i,j;
i=10; j=10;
if(i==j)
    System.out.println("EGAL!");
else
    System.out.println("nu e egal");
```

```
String s1,s2;
s1="Abc"; s2="Abc";
if(s1==s2)
    System.out.println("EGAL!");
else
    System.out.println("nu e egal");
```

Compararea variabilelor (2.1)

- Actually, JVM face un “cache” al constantelor string. Din aceasta cauza, s1 si s2 pointeaza spre aceeasi zona de memorie.

Dar atentie, **NU MERGE TOT TIMPUL** !!!!

- Nu contati pe JVM ca va ajuta.

```
String s1,s2;
s1="Abc"; s2="Abc";
if(s1==s2)
    System.out.println("EGAL!");
else
    System.out.println("nu e egal");
```

Compararea variabilelor (3)

```
BigInteger b1,b2;
b1=new BigInteger("123");
b2=new BigInteger("123");
if(b1==b2)
    System.out.println("EGAL!");
else
    System.out.println("DIFERIT");

if(b1.equals(b2))
    System.out.println("EGAL!");
else
    System.out.println("DIFERIT");
```

Metoda equals() garanteaza comparatia corecta.

Cand scrieti clase, trebuie sa o implementati voi!

Compararea variabilelor (4)

```
package exam;
public class Contor {
    private Integer value;

    public Contor() {
        this.value = 0;
    }

    Generate
    Constructor...
    equals() and hashCode()...
    toString()...
    Delegate Method...
    Override Method...
    Add Property...
    etValue() {
    ;
    alue(Integer value) {
     value;
}
```

Cand scrieti clase, trebuie sa implementati voi metoda equals()!

Compararea variabilelor (5)

```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Contor other = (Contor) obj;
    if (!Objects.equals(this.value, other.value)) {
        return false;
    }
    return true;
}
```

Actually, NetBeans o scrie pentru voi! (5 click-uri de mouse)

nota: clasa Objects e doar din java 1.7

Compararea variabilelor (6)

- Metoda **equals()** este mostenita de la clasa de baza Object.
- Pentru a lucra corect, metoda trebuie sa verifice daca parametrul primit nu este null, daca are aceeasi clasa cu obiectul curent, si abia apoi se compara efectiv variabilele.
- Atentie, variabilele membru se compara folosind tot **equals()**

Compararea variabilelor (7)

- Cand comparati chiar si cu equals, trebuie sa va asigurati ca obiectul pe care apelati **equals()** nu e null.
- **b1.equals(b2)** -> trebuie ca b1 sa nu fie null.
- **If(b1!=null && b1.equals(b2))**
  - **&&** e operator “short circuit”
  - Daca **b1!=null** e false, nu se mai face **b1.equals()**
  - => Nu se mai arunca exceptie
- Cand comparati String:
  - **If(“Popescu”.equals(s2))**
  - “Popescu” -> shortcut de la **new String(“Popescu”)**

Compararea variabilelor (8)

- Mai multe, cand lucrati si va loviti de problemele ce le impun compararea variabilelor
- Problemele de la comparare sunt similare cu problemele care apar cand vrem sa duplicam o variabila.
- == compara adrese
- = asigneaza adrese

Compararea variabilelor

- Un dezavantaj mai putin popularizat al limbajului Java.
- Cu putina disciplina, problemele nu apar.