

ΤΕΧΝΙΚΕΣ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΟΥΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Αναφορές
Στοίβα και Σωρός
Αναφορές-Παράμετροι
String Interning

new

- Όπως είδαμε για να δημιουργήσουμε ένα αντικείμενο χρειάζεται να καλέσουμε τη **new**.
 - Για τον πίνακα είπαμε ότι έτσι δίνουμε χώρο στον πίνακα και δεσμεύουμε την απαιτούμενη μνήμη.
- Τι ακριβώς συμβαίνει όταν καλούμε την new?

Η μνήμη του υπολογιστή

- Η **κύρια μνήμη** (main memory) του υπολογιστή κρατάει τα **δεδομένα** (και τις εντολές) για την εκτέλεση των προγραμμάτων.
 - Η μνήμη είναι προσωρινή, τα δεδομένα χάνονται όταν ολοκληρωθεί το πρόγραμμα.
- Η μνήμη είναι χωρισμένη σε **bytes** (8 bits)
 - Ο χώρος που χρειάζεται για ένα **χαρακτήρα ASCII**.
- Το κάθε byte έχει μια **διεύθυνση**, με την οποία μπορούμε να προσπελάσουμε τη συγκεκριμένη θέση μνήμης
 - **Random Access Memory (RAM)**
 - Σε 32-bit συστήματα μια διεύθυνση είναι 32 bits, σε 64-bit συστήματα μια διεύθυνση είναι 64 bits.

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	'a'
0001	'b'
0010	'c'
0011	'd'
0100	'e'
0101	'f'
0110	'g'
0111	'h'

Αποθήκευση μεταβλητών

- Η **κύρια μνήμη** (main memory) του υπολογιστή κρατάει τις **μεταβλητές** ενός προγράμματος
- Μια μεταβλητή μπορεί να απαιτεί χώρο περισσότερο από 1 byte.
 - Π.χ., οι μεταβλητές τύπου double χρειάζονται 8 bytes.
 - Η μεταβλητή τότε αποθηκεύεται σε συνεχόμενα bytes στη μνήμη.
- Η **θέση μνήμης** (διεύθυνση) της μεταβλητής θεωρείται το **πρώτο byte** από το οποίο ξεκινάει η αποθήκευση του της μεταβλητής.
 - Στο παράδειγμα μας η μεταβλητή βρίσκεται στη θέση 0000
 - Αν ξέρουμε την αρχή και το μέγεθος της μεταβλητής μπορούμε να τη διαβάσουμε.
- Άρα μία **θέση μνήμης** αποτελείται από μία **διεύθυνση** και το **μέγεθος**.

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	8.5
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Αποθήκευση μεταβλητών πρωταρχικού τύπου

- Για τις μεταβλητές **πρωταρχικού** τύπου (char, int, double,...) ξέρουμε εκ των προτέρων το μέγεθος της μνήμης που χρειαζόμαστε.
- Όταν ο μεταγλωττιστής δει τη **δήλωση** μιας μεταβλητής πρωταρχικού τύπου **δεσμεύει** μια θέση μνήμης αντίστοιχου μεγέθους
 - Η δήλωση μιας μεταβλητής ουσιαστικά **δίνει ένα όνομα** σε μία θέση μνήμης
 - Συχνά λέμε η **θέση μνήμης x** για τη μεταβλητή **x**.

```
int x = 5;  
int y = 3;
```

	Διεύθυνση μνήμης	Περιεχόμενο μνήμης
x	0000	5
	0001	
	0010	
	0011	
y	0100	3
	0101	
	0110	
	0111	

Αποθήκευση αντικειμένων

- Για τα αντικείμενα δεν ξέρουμε πάντα εκ των προτέρων το μέγεθος της μνήμης που θα πρέπει να δεσμεύσουμε.

```
String s; // δεν ξερουμε το μέγεθος του s  
s = "ab"; // το s έχει μέγεθος 2 χαρακτήρες  
s = "abc"; // το s έχει μέγεθος 3 χαρακτήρες
```

- Παρομοίως αν δηλώσουμε

```
int[] A;
```

μας λέει ότι έχουμε ένα πίνακα από ακέραιους αλλά δεν μας λέει πόσο μεγάλος θα είναι αυτός ο πίνακας.

```
A = new int[2];  
A = new int[3];
```

Αποθήκευση αντικειμένων

- Οι θέσεις μνήμης των αντικειμένων κρατάνε μια **διεύθυνση** στο χώρο στον οποίο αποθηκεύεται το αντικείμενο
- Η διεύθυνση αυτή λέγεται **αναφορά**.
- Οι αναφορές είναι παρόμοιες με τους **δείκτες** σε άλλες γλώσσες προγραμματισμού με τη διαφορά ότι η Java δεν μας αφήνει να πειράξουμε τις διευθύνσεις.
 - Εμείς χρησιμοποιούμε μόνο τη μεταβλητή του αντικειμένου, όχι το περιεχόμενο της
- Το **dereferencing** το κάνει η Java αυτόματα.

```
String s = "ab";
```

	Διεύθυνση μνήμης	Περιεχόμενο μνήμης
s	0000	0100
	0001	
	0010	
	0011	
	0100	a
	0101	b
	0110	
	0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Η δεσμευμένη λέξη **null** σημαίνει μια **κενή αναφορά** (μια διεύθυνση που δεν δείχνει πουθενά)

	Διεύθυνση μνήμης	Περιεχόμενο μνήμης
A	0000	null
	0001	
	0010	
	0011	
	0100	
	0101	
	0110	
	0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Με την εντολή **new** δεσμεύουμε δύο θέσεις ακεραίων και η αναφορά του A δείχνει σε αυτό το χώρο που δεσμεύσαμε

A

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0011
0001	
0010	
0011	0
0100	0
0101	
0110	
0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Ο τελεστής [] για τον πίνακα μας πάει στην αντίστοιχη θέση του χώρου που κρατήσαμε

A

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0011
0001	
0010	
0011	10
0100	0
0101	
0110	
0111	

Παράδειγμα - πινάκες

```
int[] A;  
A = new int[2];  
A[0] = 10;  
A = new int[3];
```

Με νέα κλήση της **new** δεσμεύουμε νέο χώρο για το A, και αν δεν έχουμε κρατήσει την προηγούμενη αναφορά σε κάποια άλλη μεταβλητή τότε χάνεται (garbage collection), όπως και οι τιμές που είχαμε αποθηκεύσει στον πίνακα.

A

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0101
0001	
0010	
0011	
0100	
0101	
0110	0
0111	0

Αντικείμενα κλάσεων

- Τι γίνεται με τα αντικείμενα κλάσεων που ορίσαμε εμείς?
- Παράδειγμα: Η κλάση `Person` (ToyClass από το βιβλίο).

```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber) {
        name = initName;
        number = initNumber;
    }

    public void set(String newName, int newNumber) {
        name = newName;
        number = newNumber;
    }

    public String toString() {
        return (name + " " + number);
    }
}
```

Παράδειγμα

```
Person varP = new Person ("Bob", 1);
```

varP

Η κλήση της **new** δημιουργεί ένα χώρο μνήμης για την αποθήκευση του αντικειμένου τύπου Person το οποίο κρατάει ένα string και ένα ακέραιο (δεσμεύεται χώρος και γι αυτά).

Η μεταβλητή **varP** κρατάει την διεύθυνση του χώρου στην μνήμη όπου αποθηκεύσαμε αυτό το αντικείμενο.

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	
0010	"Bob"
0011	
0100	
0101	
0110	1
0111	

Αναθέσεις μεταξύ αντικειμένων

Τι θα τυπώσει το παρακάτω πρόγραμμα?

```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Αναθέσεις μεταξύ αντικειμένων

varP1

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	
0010	"Bob"
0011	
0100	
0101	1
0110	
0111	

```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

Αναθέσεις μεταξύ αντικειμένων

varP1

varP2

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	null
0010	"Bob"
0011	
0100	
0101	1
0110	
0111	

```
Person varP1 = new Person("Bob", 1);
```

```
Person varP2;
```

```
varP2 = varP1;
```

```
varP2.set("Ann", 2);
```

```
System.out.println(varP1);
```

Αναθέσεις μεταξύ αντικειμένων

Η ανάθεση του `varP1` στο `varP2` έχει αποτέλεσμα η μεταβλητή `varP2` να δείχνει στην ίδια θέση μνήμης όπως και η `varP1`

`varP1`

`varP2`

```
Person varP1 = new Person("Bob", 1);
Person varP2;
varP2 = varP1;
varP2.set("Ann", 2);
System.out.println(varP1);
```

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	0010
0010	
0011	"Bob"
0100	
0101	1
0110	
0111	

Αναθέσεις μεταξύ αντικειμένων

Η αλλαγή θα γίνει στο χώρο μνήμης που δείχνει ο `varP2`
Αυτός είναι ο ίδιος όπως αυτός που δείχνει και ο `varP1`

`varP1`

`varP2`

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	0010
0010	"Ann"
0011	
0100	
0101	2
0110	
0111	

```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

Αναθέσεις μεταξύ αντικειμένων

Τυπώνει "Ann 2"

Αλλάζοντας τα περιεχόμενα της θέσης μνήμης στην οποία δείχνει ο `varP2` αλλάζουμε και το `varP1`

`varP1`

`varP2`

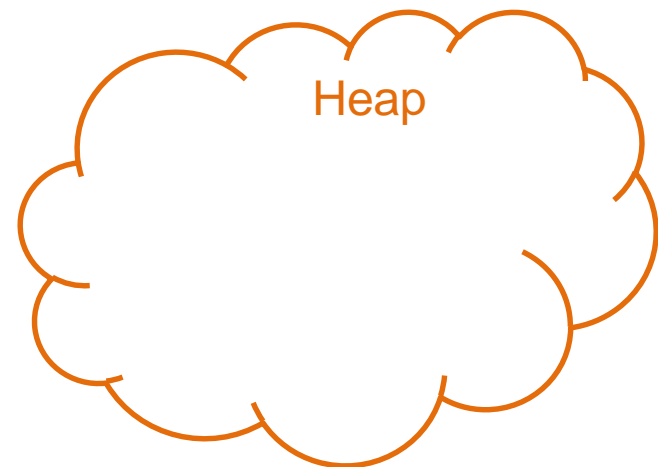
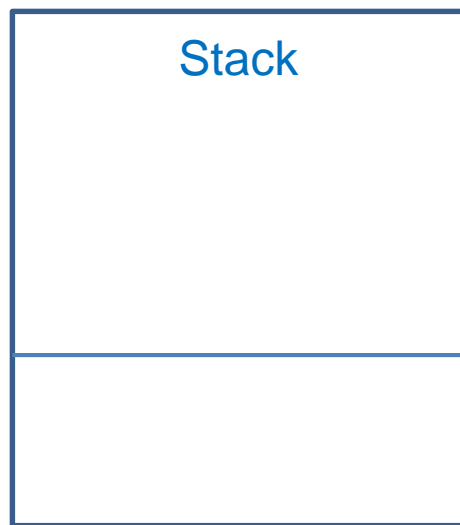
```
Person varP1 = new Person("Bob", 1);  
Person varP2;  
varP2 = varP1;  
varP2.set("Ann", 2);  
System.out.println(varP1);
```

Διεύθυνση μνήμης	Περιεχόμενο μνήμης
0000	0010
0001	0010
0010	←
0011	"Ann"
0100	
0101	2
0110	
0111	

ΣΤΟΙΒΑ ΚΑΙ ΣΩΡΟΣ

Διαχείριση μνήμης από το JVM

- Η μνήμη χωρίζεται σε δύο τμήματα
 - Τη στοίβα (**stack**) που χρησιμοποιείται για να κρατάει πληροφορία για τις **τοπικές μεταβλητές** κάθε μεθόδου/block.
 - Το σωρό (**heap**) που χρησιμοποιείται για να δεσμεύουμε **μνήμη για τα αντικείμενα**



Stack

- Κάθε φορά που καλείται μία μέθοδος, δημιουργείται ένα «πλαίσιο» (**frame**) για την μέθοδο στη στοίβα
 - Δημιουργείται ένας **χώρος μνήμης** που αποθηκεύει τις **παραμέτρους** και τις **τοπικές μεταβλητές** της μεθόδου.
- Αν η μέθοδος καλέσει μία άλλη μέθοδο θα δημιουργηθεί ένα νέο πλαίσιο και θα τοποθετηθεί (push) στην **κορυφή της στοίβας**.
- Όταν βγούμε από την μέθοδο το πλαίσιο **αφαιρείται** (pop) από την κορυφή της στοίβας και επιστρέφουμε στην προηγούμενη μέθοδο
- Στη βάση της στοίβας είναι η μέθοδος **main**.

Παράδειγμα

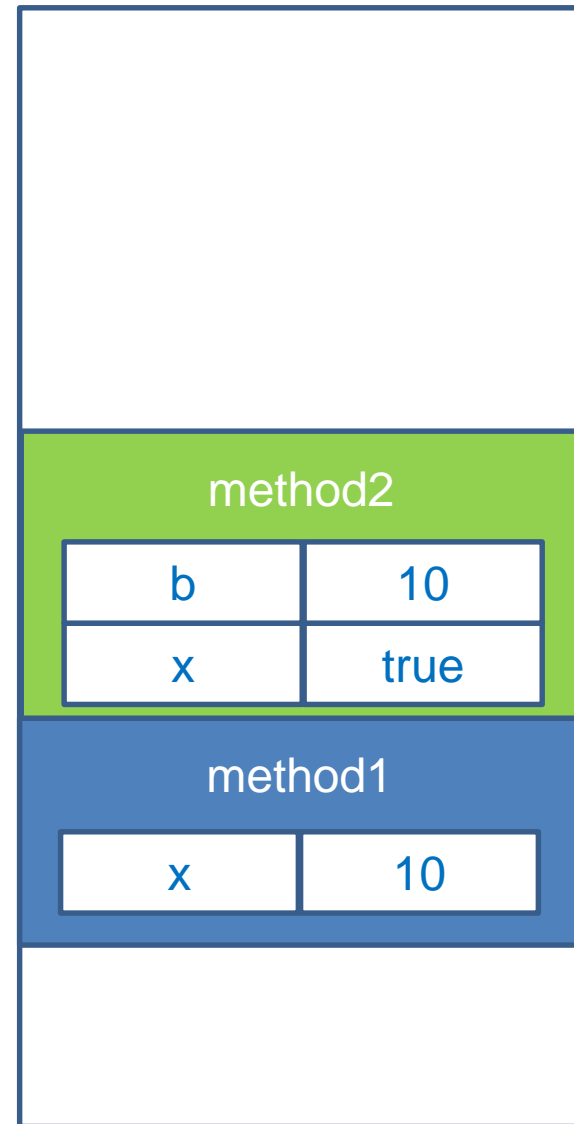
```
public void method1() {  
    int x = 10;  
    method2(x);  
}
```



Παράδειγμα

```
public void method1() {  
    int x = 10;  
    method2(x);  
}
```

```
public void method2(int b) {  
    boolean x = true;  
    method3();  
}
```

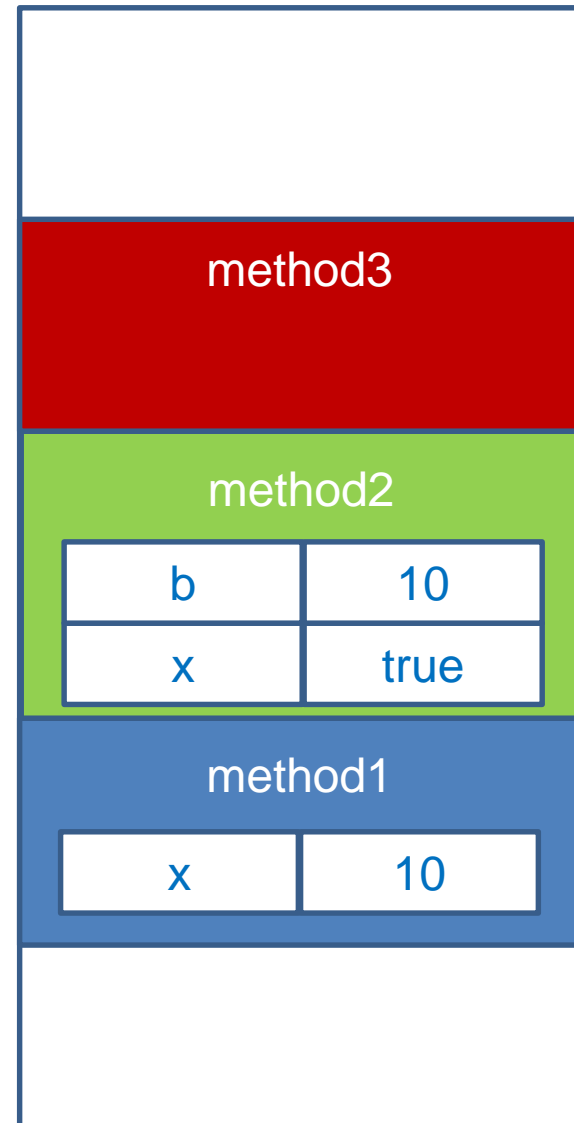


Παράδειγμα

```
public void method1 () {  
    int x = 10;  
    method2 (x) ;  
}
```

```
public void method2 (int b) {  
    boolean x = true;  
    method3 () ;  
}
```

```
public void method3 ()  
{...}
```



Παράδειγμα

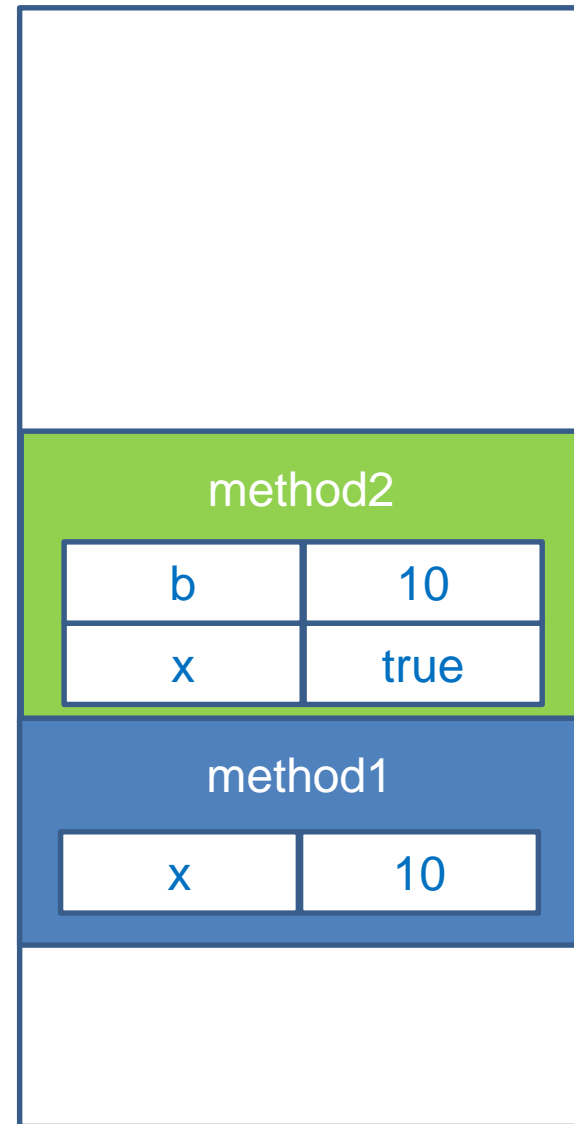
```
public void method1() {  
    int x = 10;  
    method2(x);  
    method3();  
}
```



Παράδειγμα

```
public void method1() {  
    int x = 10;  
    method2(x);  
    method3()  
}
```

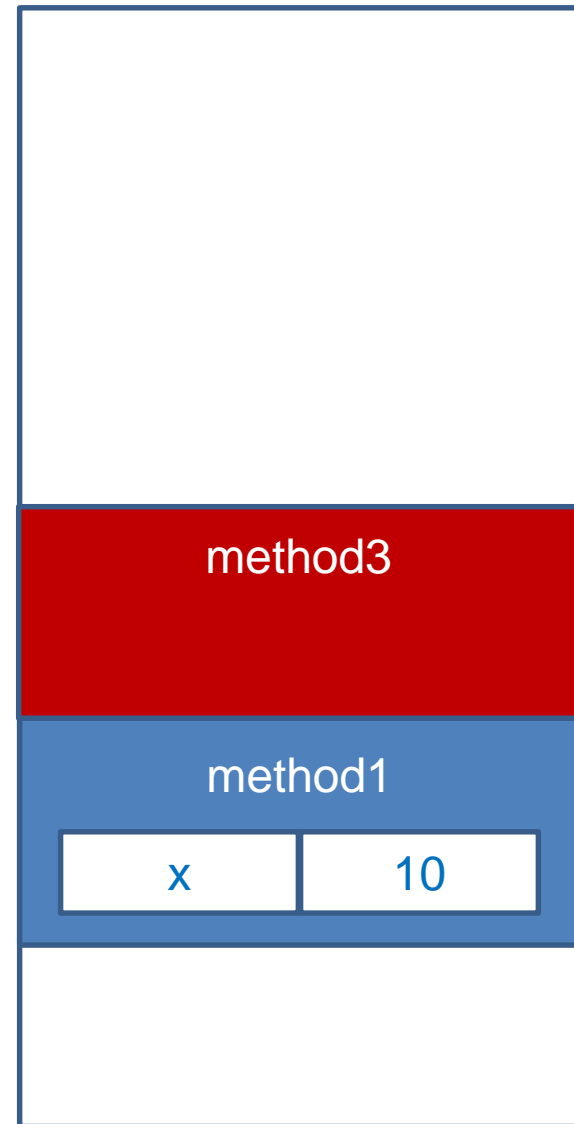
```
public void method2(int b) {  
    boolean x = (b==10);  
    ...  
}
```



Παράδειγμα

```
public void method1 () {  
    int x = 10;  
    method2 (x) ;  
}
```

```
public void method3 ()  
{...}
```



Heap

- Όταν μέσα σε μία μέθοδο δημιουργούμε ένα αντικείμενο με την **new** γίνονται τα εξής
 - στο πλαίσιο (frame) της μεθόδου (στη στοίβα) υπάρχει μια **τοπική μεταβλητή** που κρατάει την **αναφορά** στο αντικείμενο
 - Η κλήση της **new** δεσμεύει **χώρο μνήμης** στο σωρό (heap) για να κρατήσει τα πεδία του αντικειμένου.
 - Η **αναφορά** δείχνει στη **θέση μνήμης** που δεσμεύτηκε.

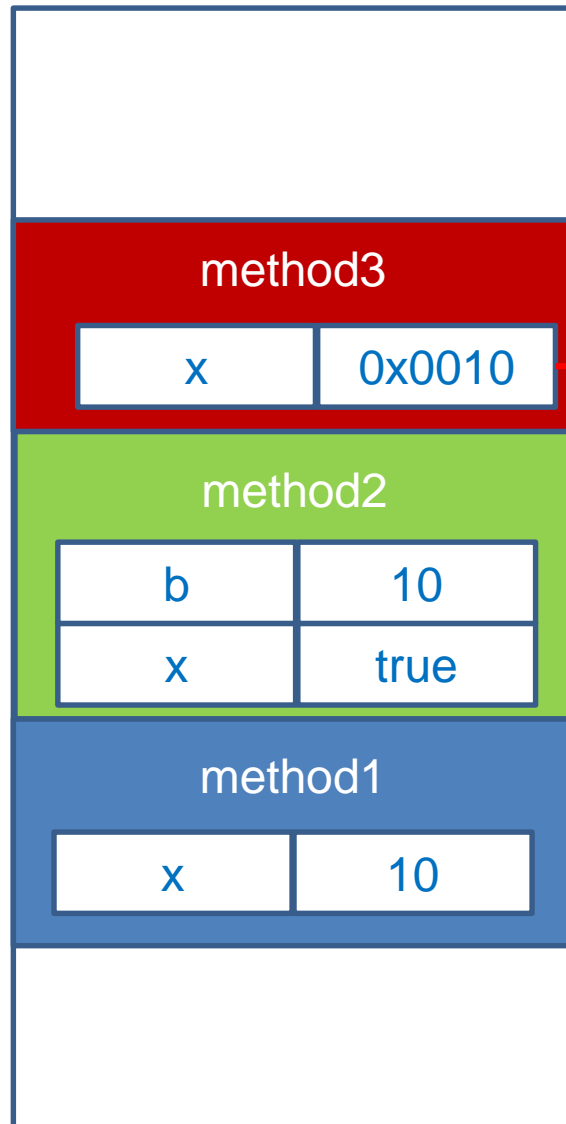
```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber) {
        name = initName;
        number = initNumber;
    }

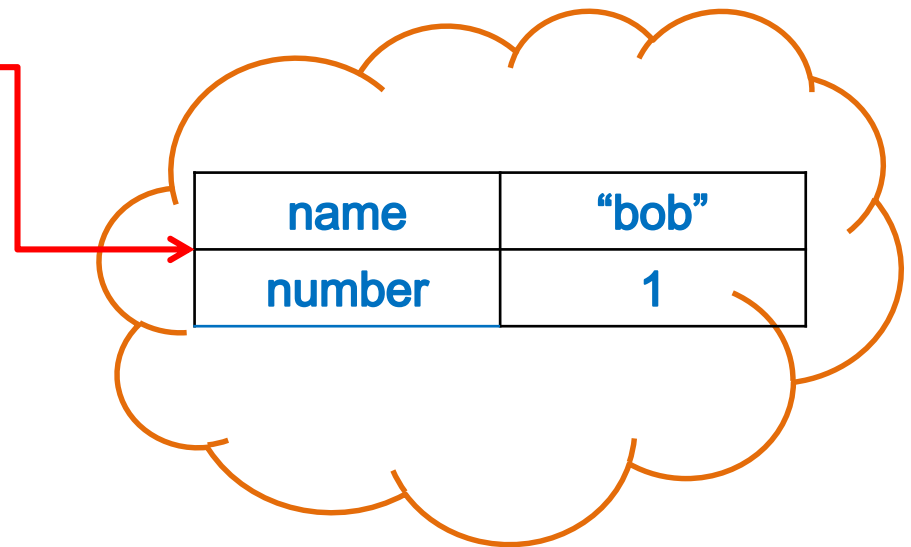
    public void set(String name, int number) {
        this.name = name;
        this.number = number;
    }

    public String toString() {
        return (name + " " + number);
    }
}
```

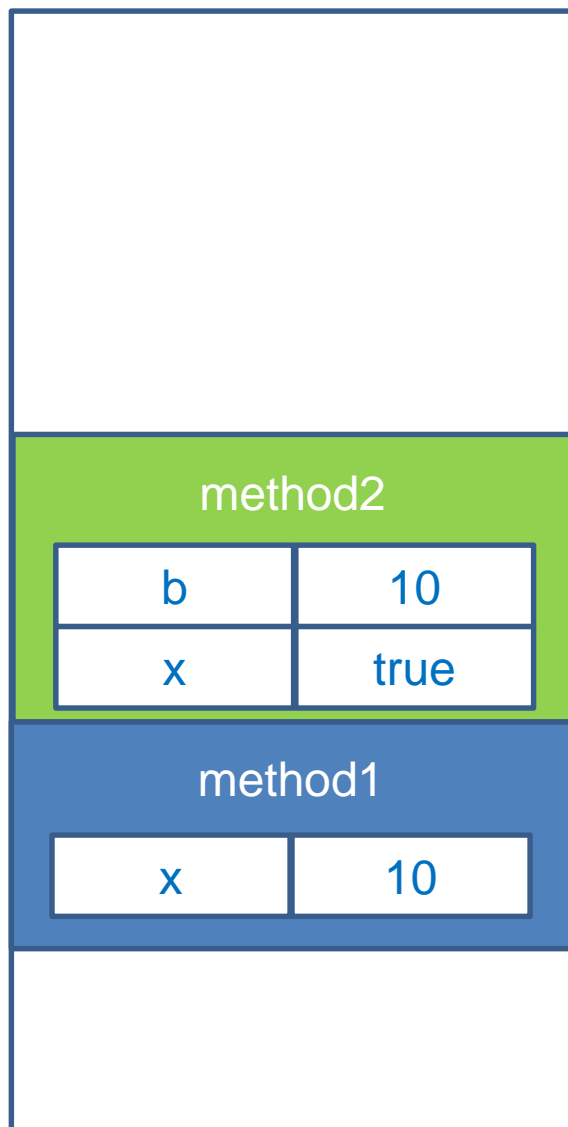

Παράδειγμα



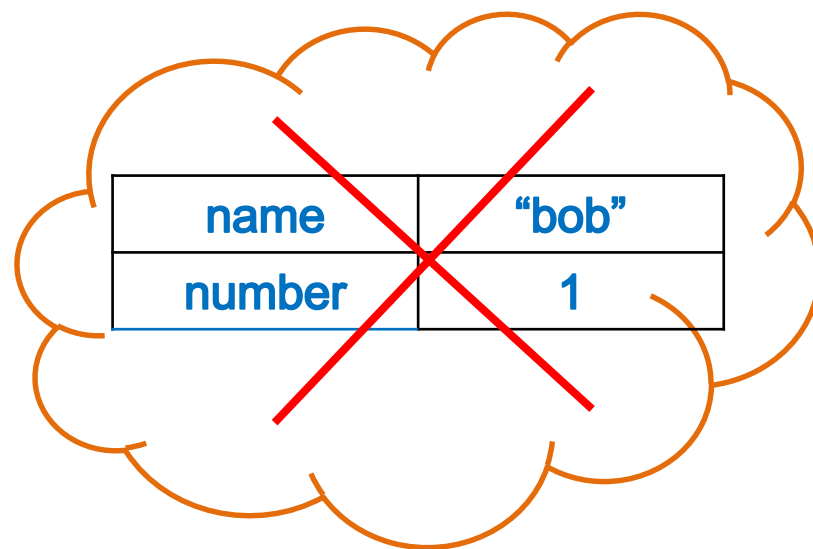
```
public void method3()  
{  
    Person x = new Person("bob",1);  
}
```



Παράδειγμα

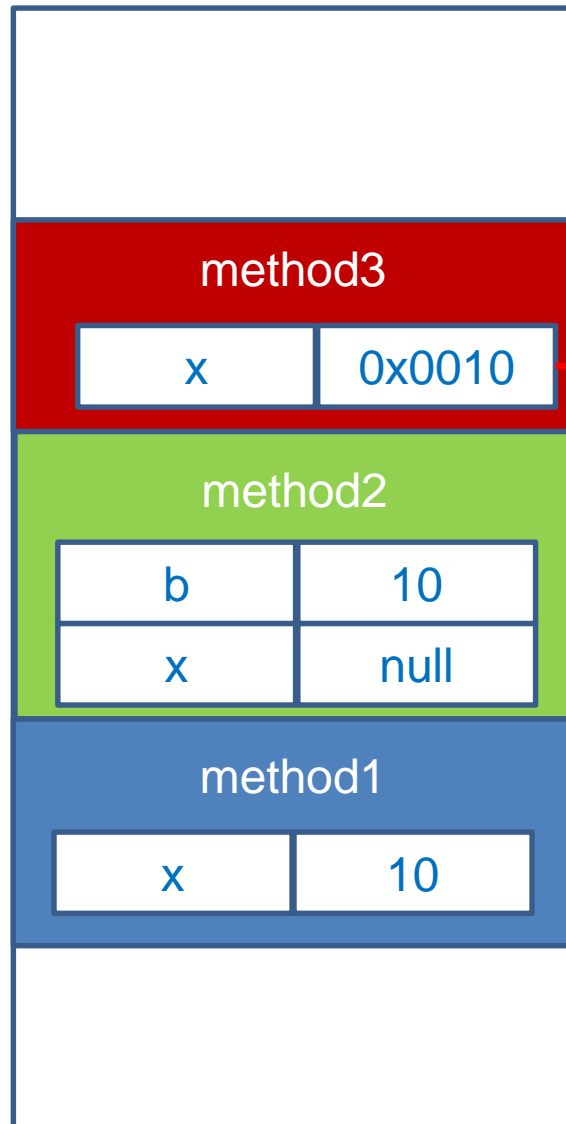


Όταν επιστρέφουμε από την μέθοδο method3 η αναφορά προς το αντικείμενο Person παύει να υπάρχει.

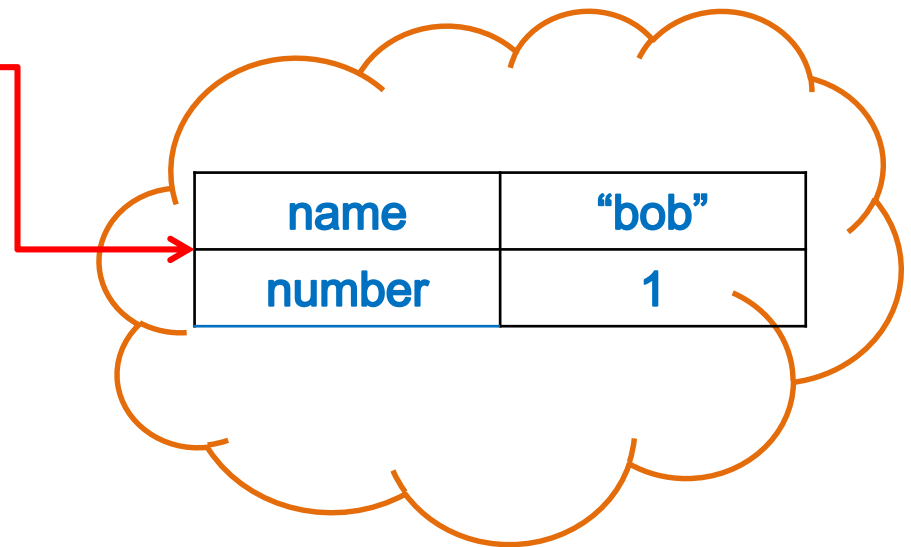


Αν δεν υπάρχουν άλλες αναφορές στο αντικείμενο τότε ο garbage collector αποδεσμεύει τη μνήμη του αντικειμένου

Παράδειγμα



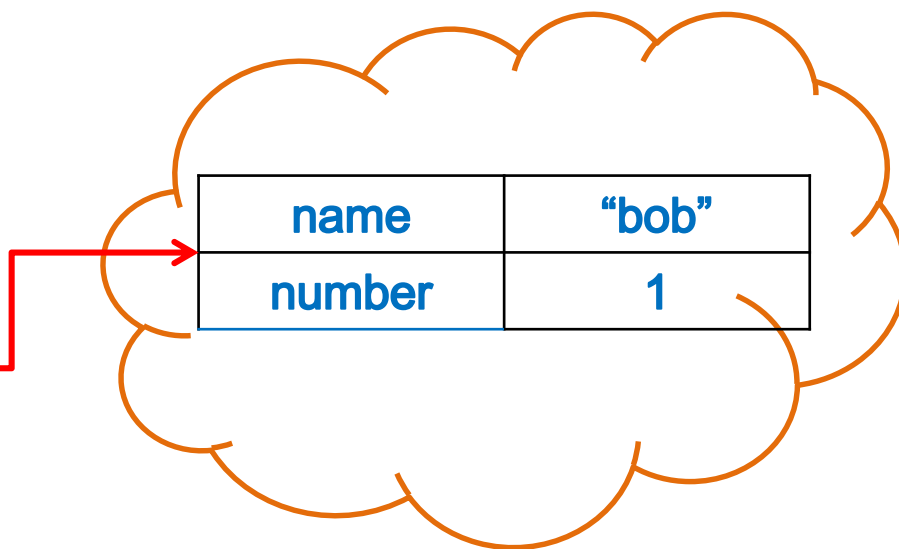
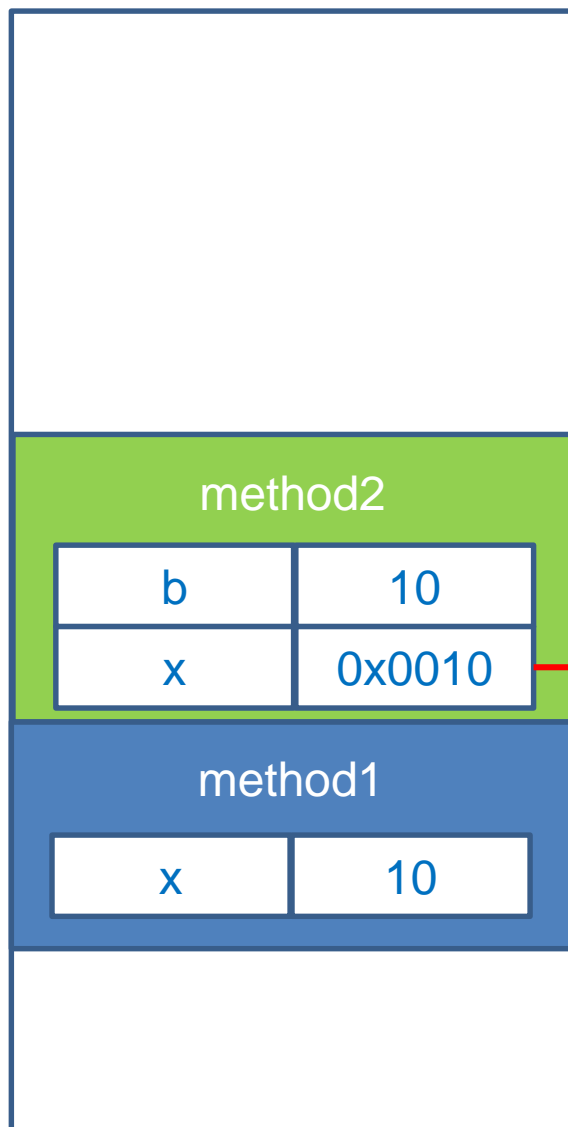
```
public Person method3()  
{  
    Person x = new Person("bob",1);  
    return x;  
}
```



Παράδειγμα

```
public void method2()  
{  
    Person x = method3()  
}
```

Στην περίπτωση αυτή η αναφορά στο αντικείμενο επιστρέφεται και αποθηκεύεται στην μεταβλητή x μέθοδο method2

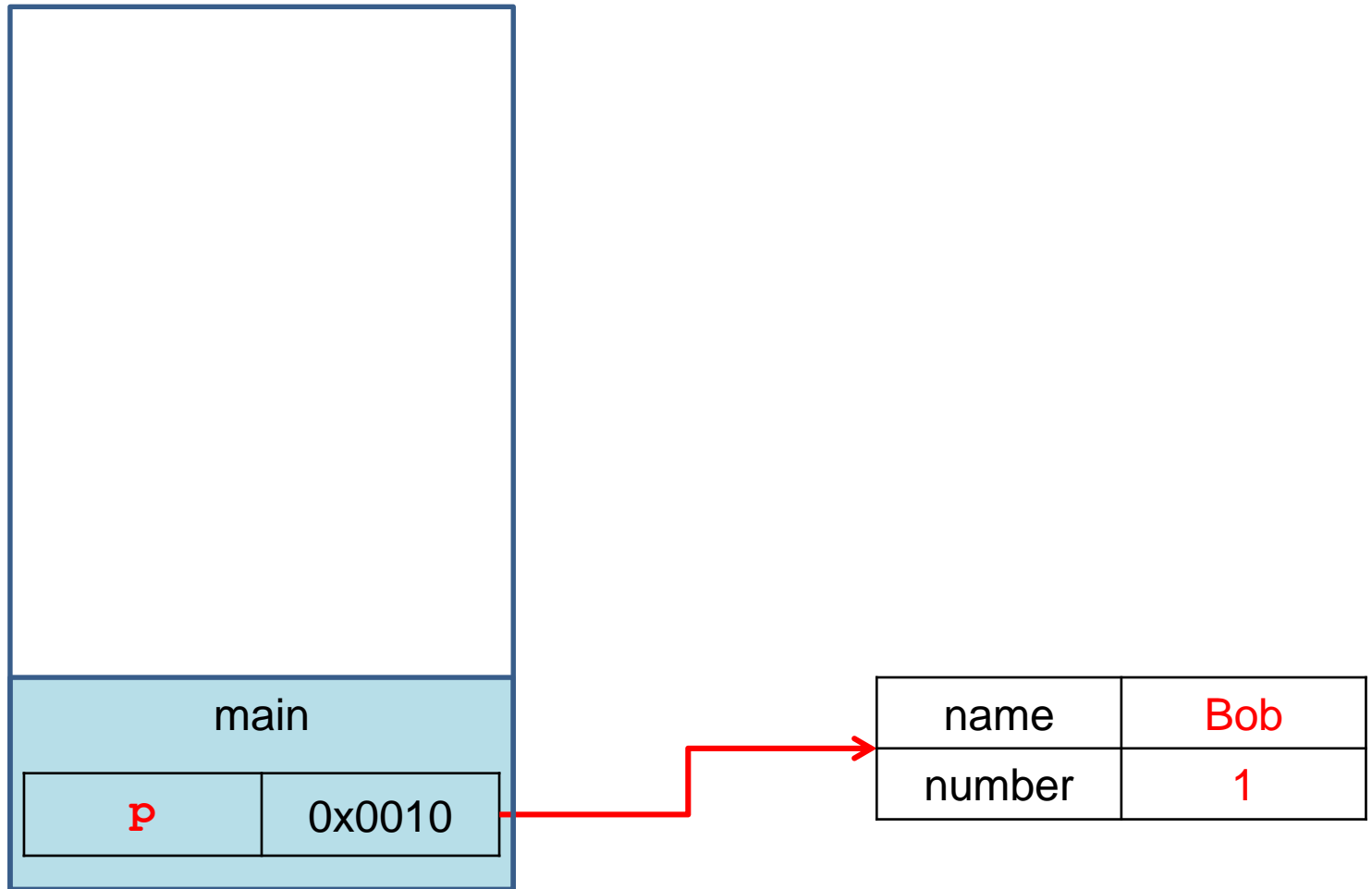


Η αναφορά δεν χάνεται και το αντικείμενο διατηρείται όσο υπάρχει αναφορά σε αυτό

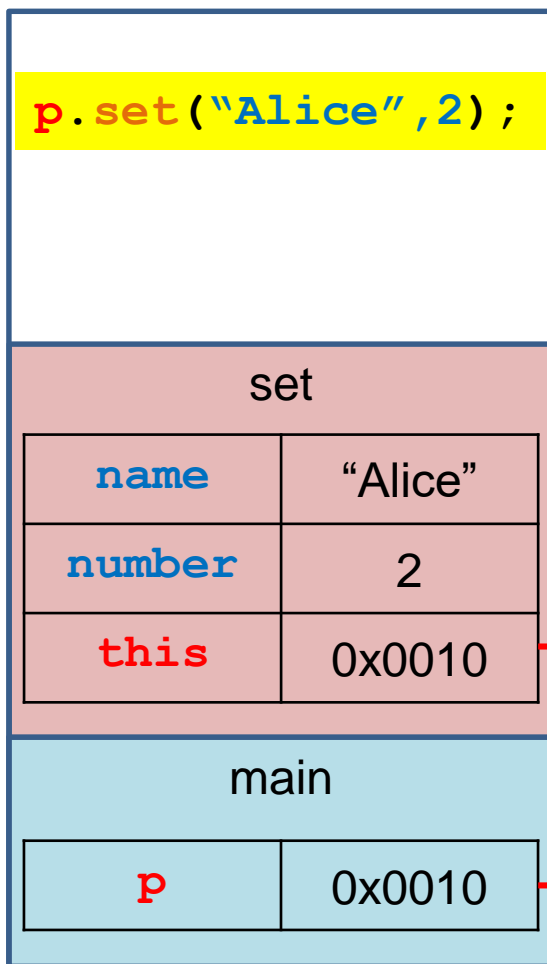
Κλήση μεθόδου από αντικείμενο

```
public class ObjectMethodCallDemo
{
    public static void main(String[] args)
    {
        Person p = new Person("Bob", 1);
        p.set("Alice", 2);
        System.out.println(p);
    }
}
```

Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος

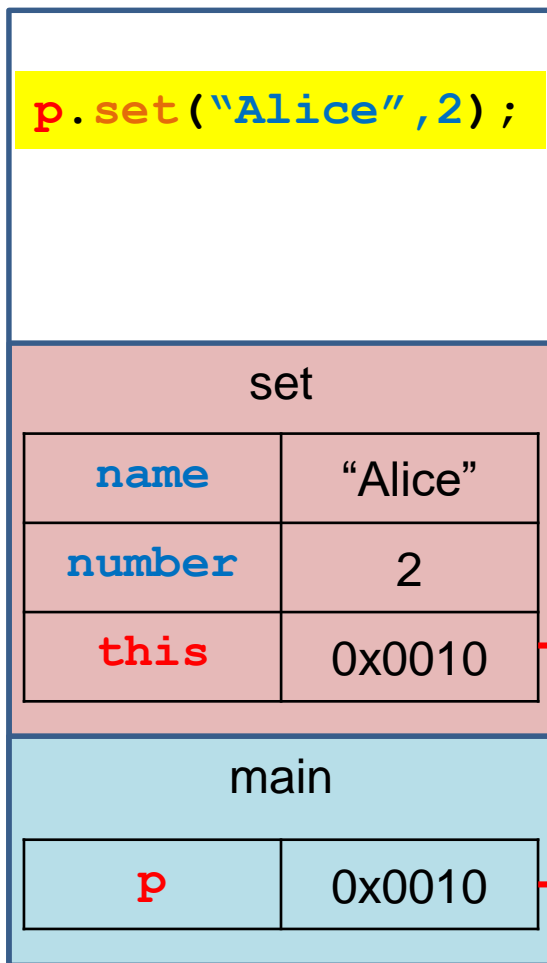


Όταν καλείται μια μέθοδος ενός αντικειμένου αυτόματα δημιουργείται στο frame της μεθόδου και η μεταβλητή **this** η οποία κρατάει μια αναφορά στο αρχικό αντικείμενο που κάλεσε την μέθοδο.

Την μεταβλητή αυτή μπορούμε να την χρησιμοποιήσουμε σαν οποιαδήποτε άλλη μεταβλητή.

name	Bob
number	1

Εξέλιξη του προγράμματος



```
this.name = name;  
this.number = number;
```

Τα this.name, this.number αναφέρονται στα πεδία του αντικειμένου ενώ τα name, number στις τοπικές μεταβλητές.

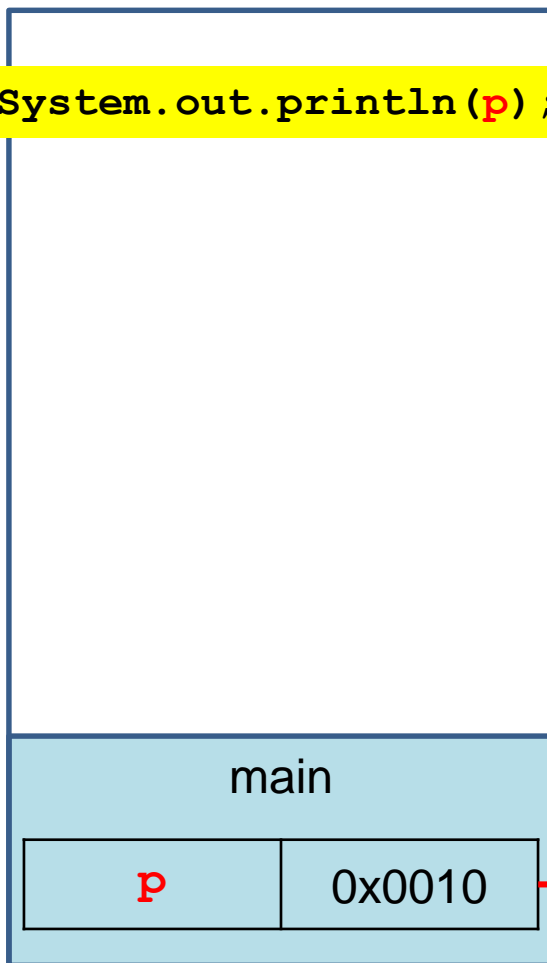
name	"Alice"
number	2

Εξέλιξη του προγράμματος

```
System.out.println(p);
```

Επιστρέφοντας οι αλλαγές που κάναμε στα πεδία του αντικειμένου this διατηρούνται στο χώρο μνήμης του p.

Τυπώνει "Alice 2"



name	"Alice"
number	2

Αντικείμενα ως παράμετροι

- Όταν περνάμε παραμέτρους σε μία μέθοδο το πέρασμα γίνεται πάντα **δια τιμής (call-by-value)**
 - Δηλαδή απλά περνάμε τα **περιεχόμενα της θέσης μνήμης** της συγκεκριμένης μεταβλητής.
 - Για μεταβλητές **πρωταρχικού** τύπου, αλλαγές στην τιμή της παραμέτρου **δεν αλλάζουν** την μεταβλητή που περάσαμε σαν όρισμα.
- Τι γίνεται όμως αν η παράμετρος είναι ένα αντικείμενο?
 - Τα **περιεχόμενα της θέσης μνήμης** μιας μεταβλητής-αντικείμενο είναι μια **αναφορά**.
 - **Αν** μέσα στην μέθοδο **αλλάξουν τα περιεχόμενα του αντικειμένου** (εκεί που δείχνει η αναφορά) τότε **αλλάζει και η μεταβλητή-αντικείμενο** που περάσαμε.

```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber){
        name = initName;
        number = initNumber;
    }

    public void set(String newName, int newNumber){
        name = newName;
        number = newNumber;
    }

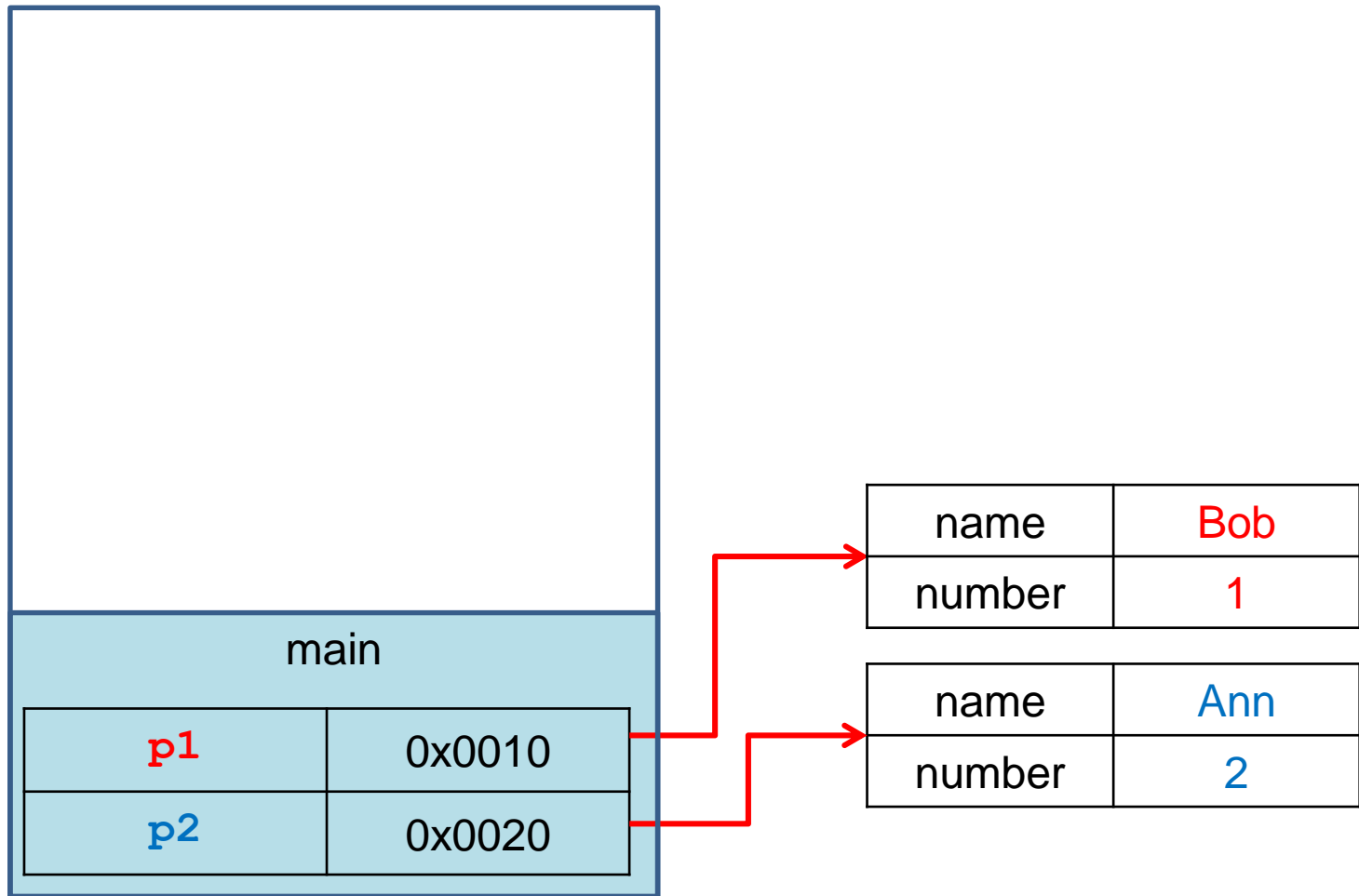
    public void copier(Person other) {
        other.name = name;
        other.number = number;
    }

    public String toString( ){
        return (name + " " + number);
    }
}
```

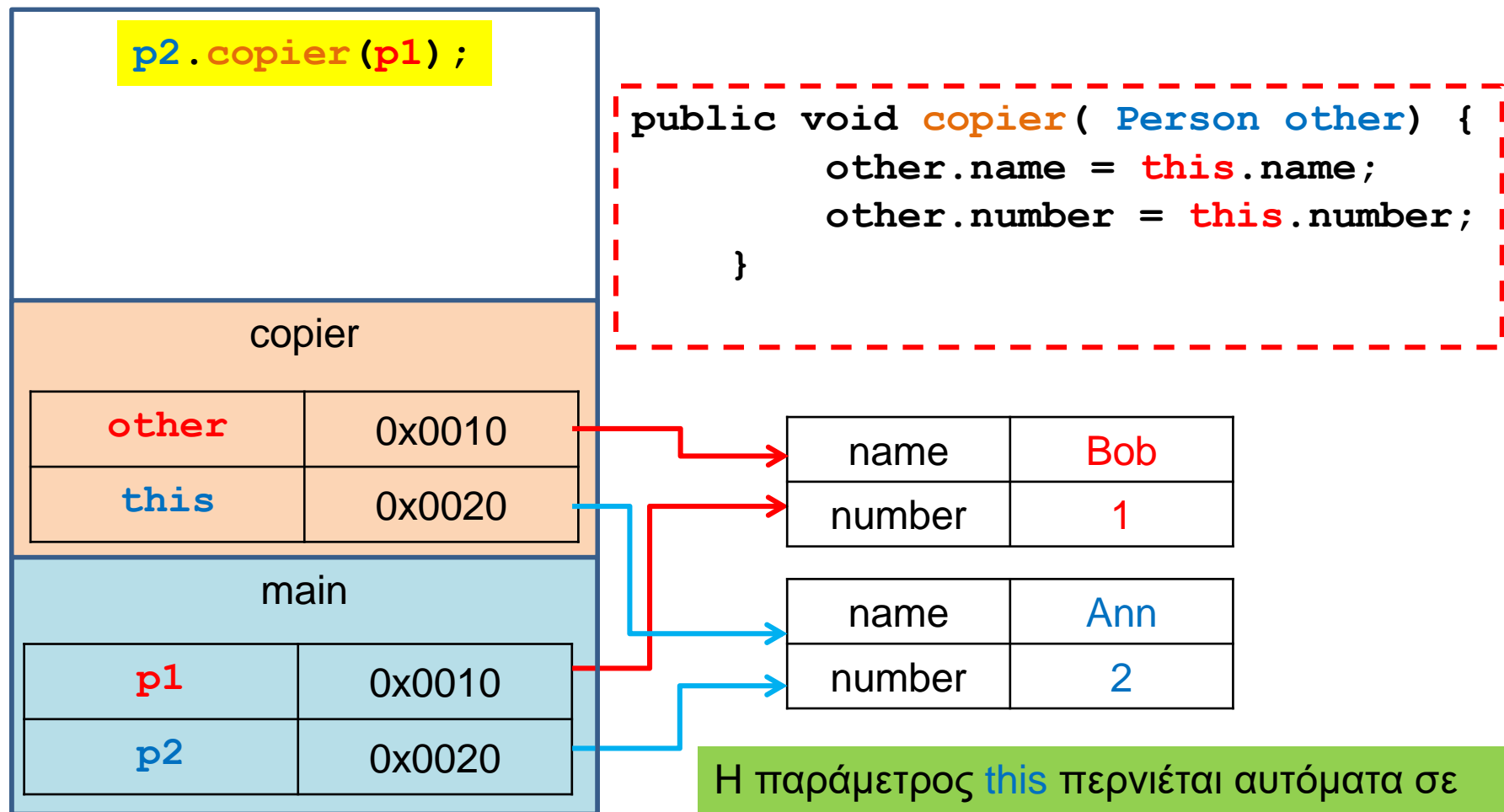
Παράδειγμα

```
public class ClassParameterDemo
{
    public static void main(String[] args)
    {
        Person p1 = new Person("Bob", 1);
        Person p2 = new Person("Ann", 2);
        p2.copier(p1);
        System.out.println(p1);
    }
}
```

Εξέλιξη του προγράμματος

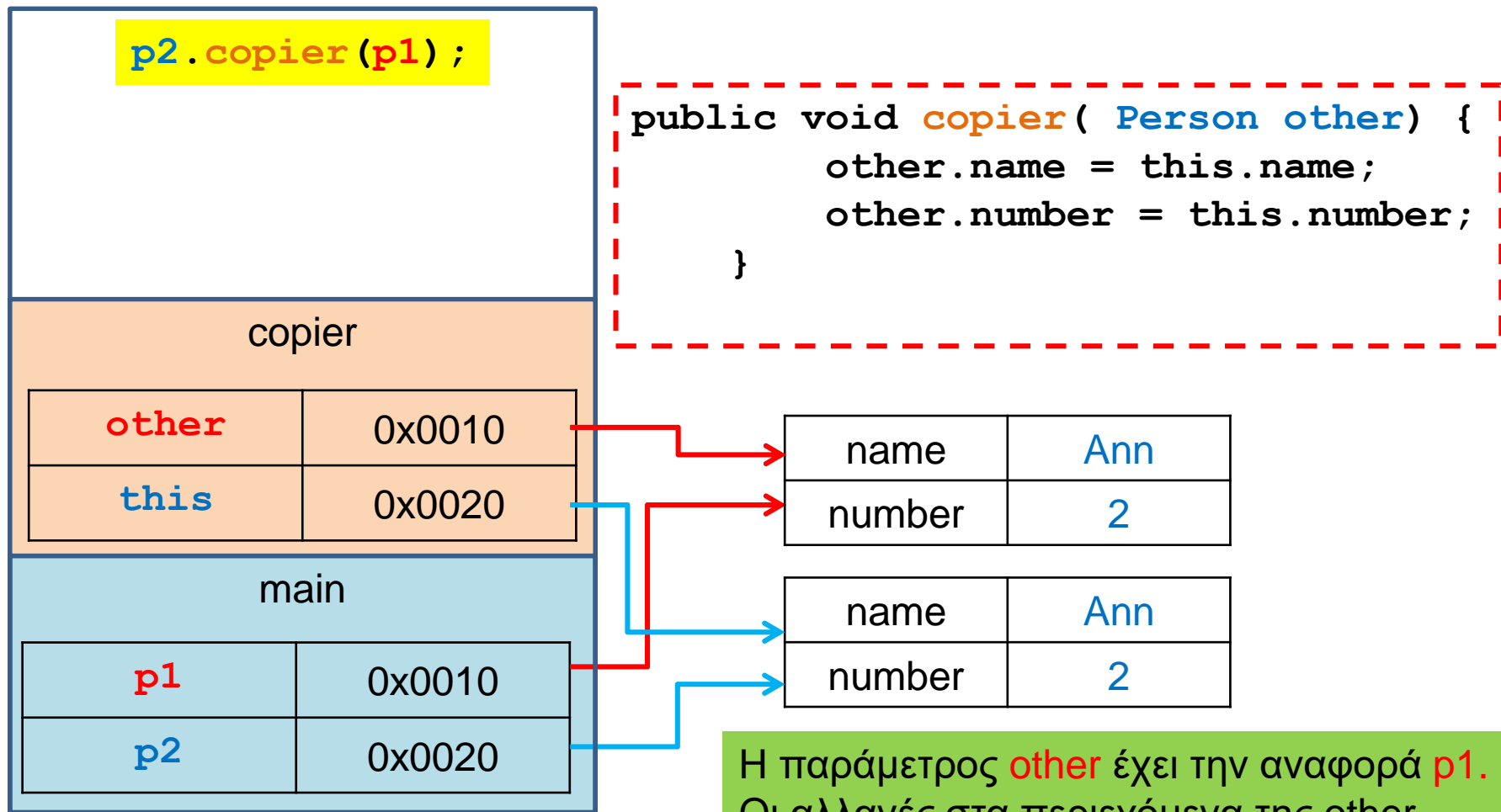


Εξέλιξη του προγράμματος



Η παράμετρος `this` περνιέται αυτόματα σε κάθε κλήση μεθόδου του αντικειμένου.
Η `other` κρατάει την αναφορά του `p1`.

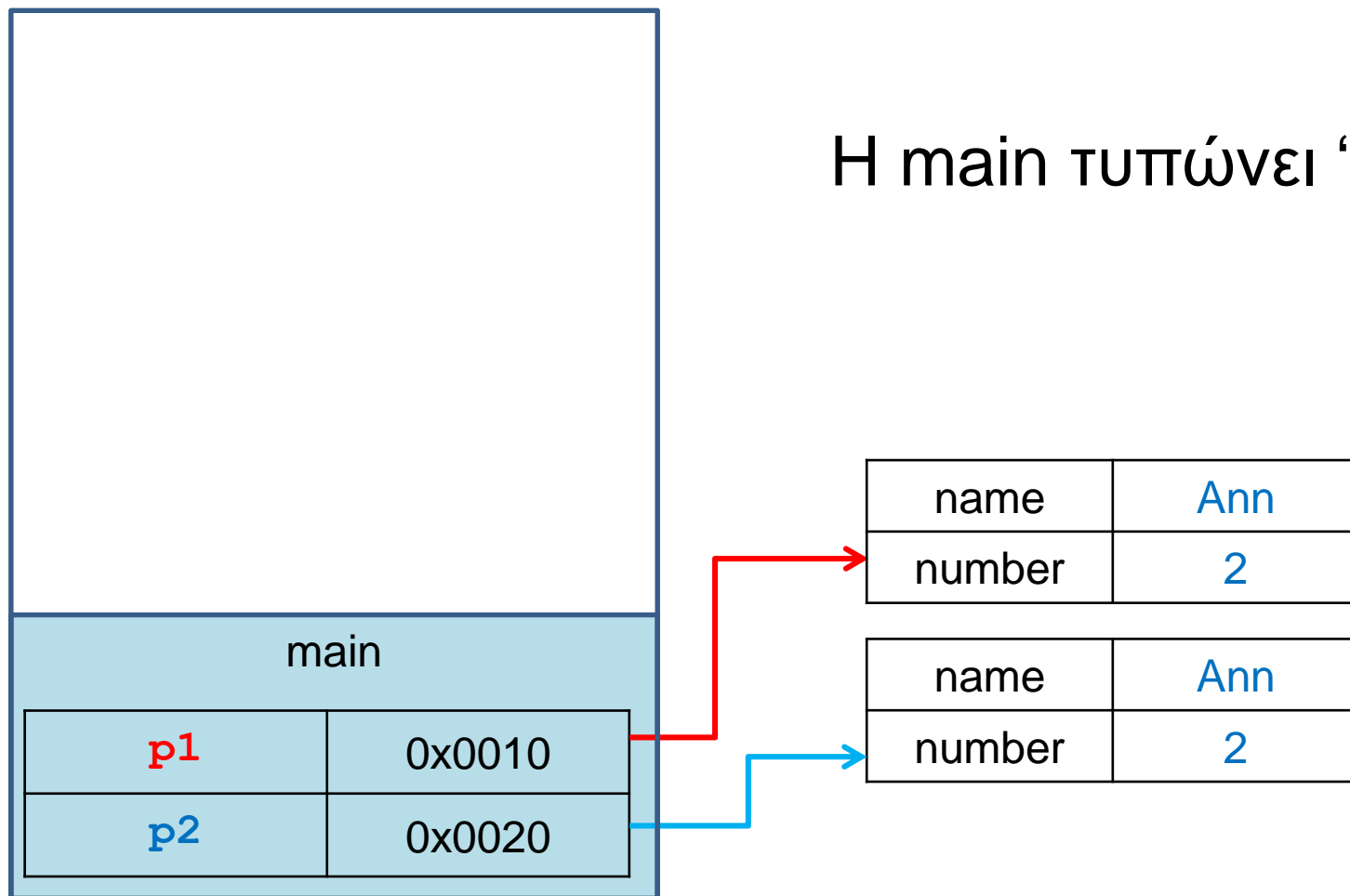
Εξέλιξη του προγράμματος



Η παράμετρος `other` έχει την αναφορά `p1`. Οι αλλαγές στα περιεχόμενα της `other` αλλάζουν και τα περιεχόμενα της `p1`.

Εξέλιξη του προγράμματος

Η main τυπώνει “Ann 2”



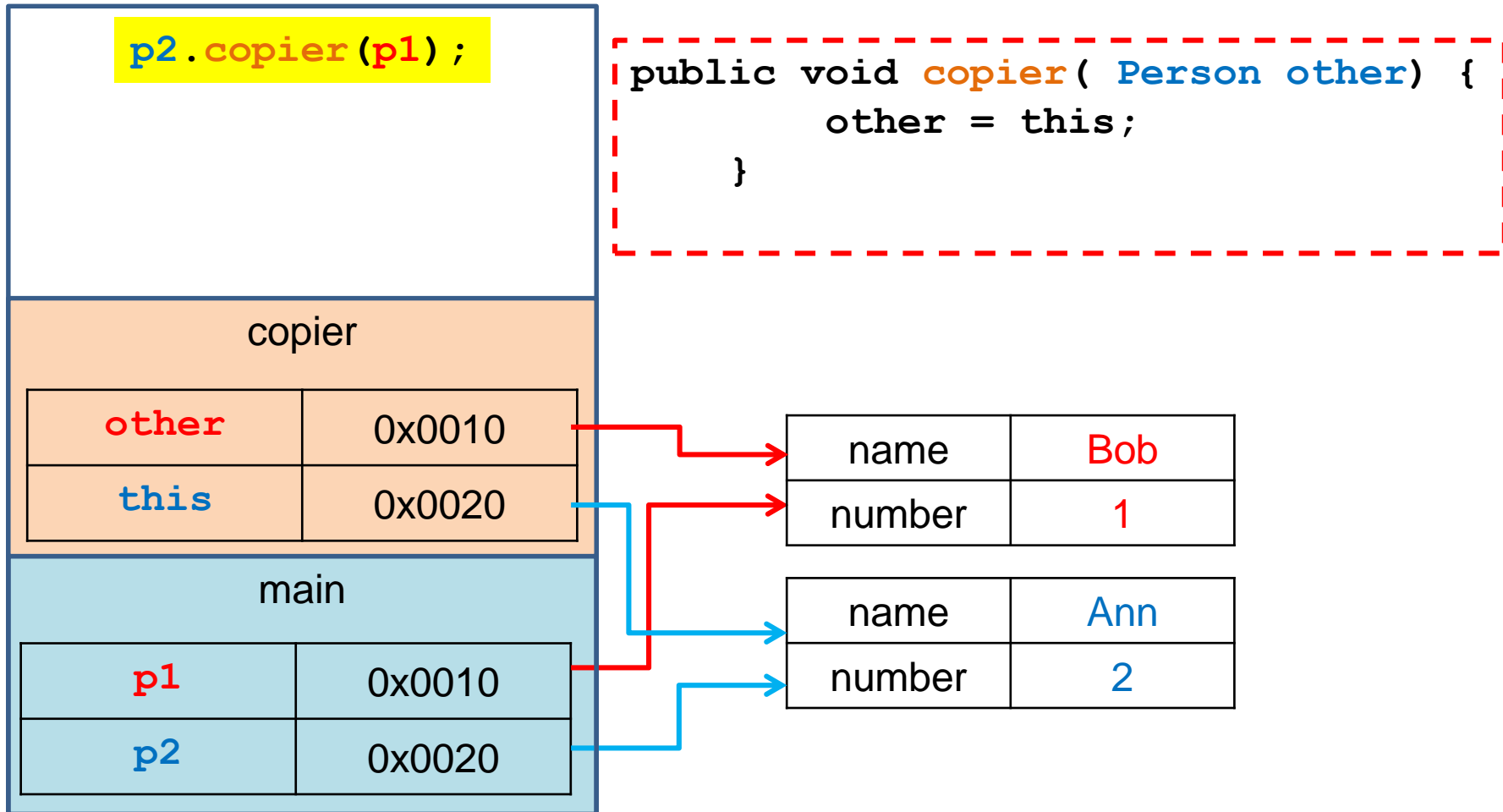
Μια άλλη υλοποίηση της copier

```
public void copier( Person other) {  
    other = this;  
}
```

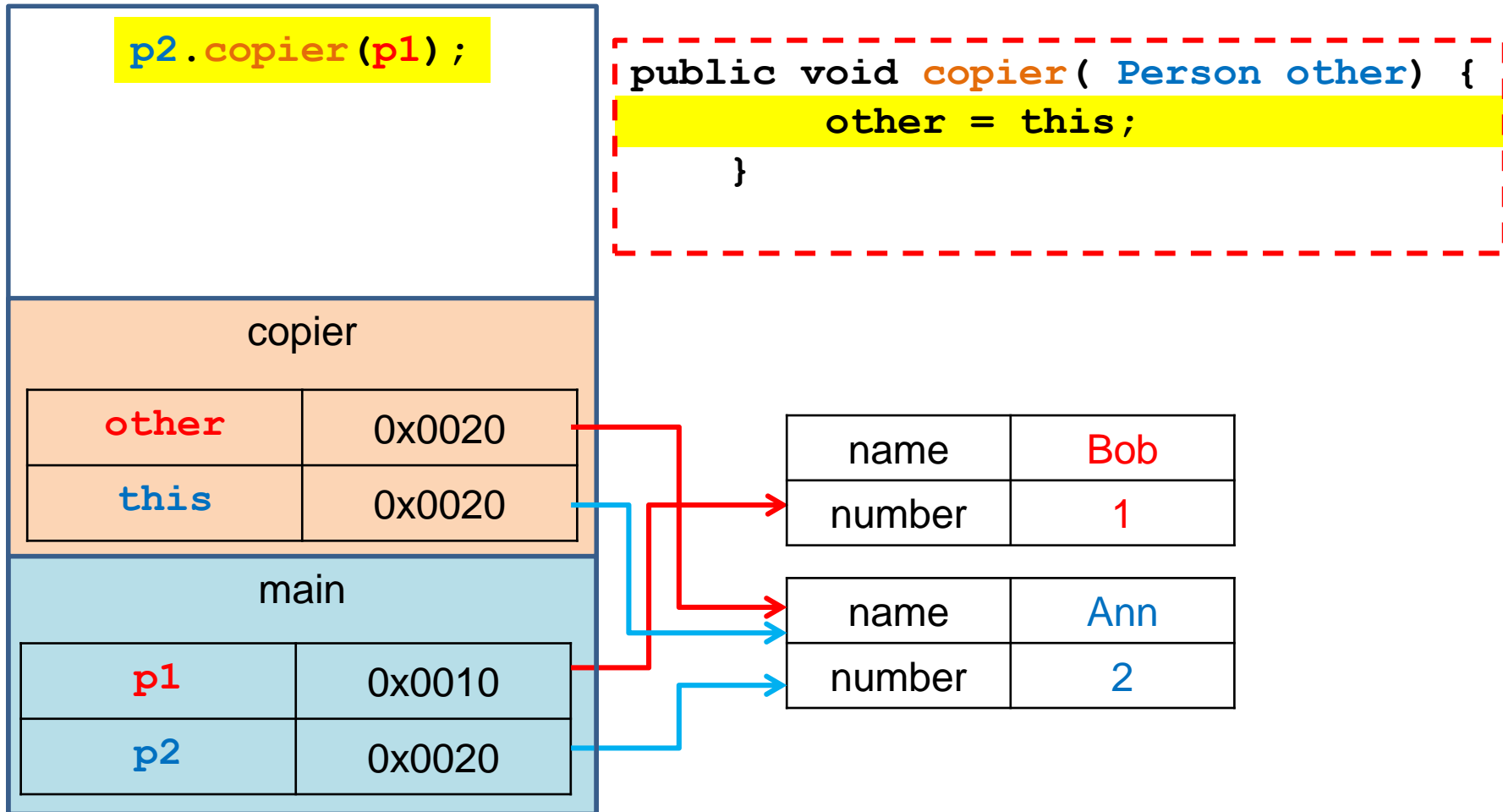
```
public class ClassParameterDemo  
{  
    public static void main(String[] args)  
    {  
        Person p1 = new Person("Bob", 1);  
        Person p2 = new Person("Ann", 2);  
        p2.copier(p1);  
        System.out.println(p1);  
    }  
}
```

Τι θα τυπώσει?

Εξέλιξη του προγράμματος

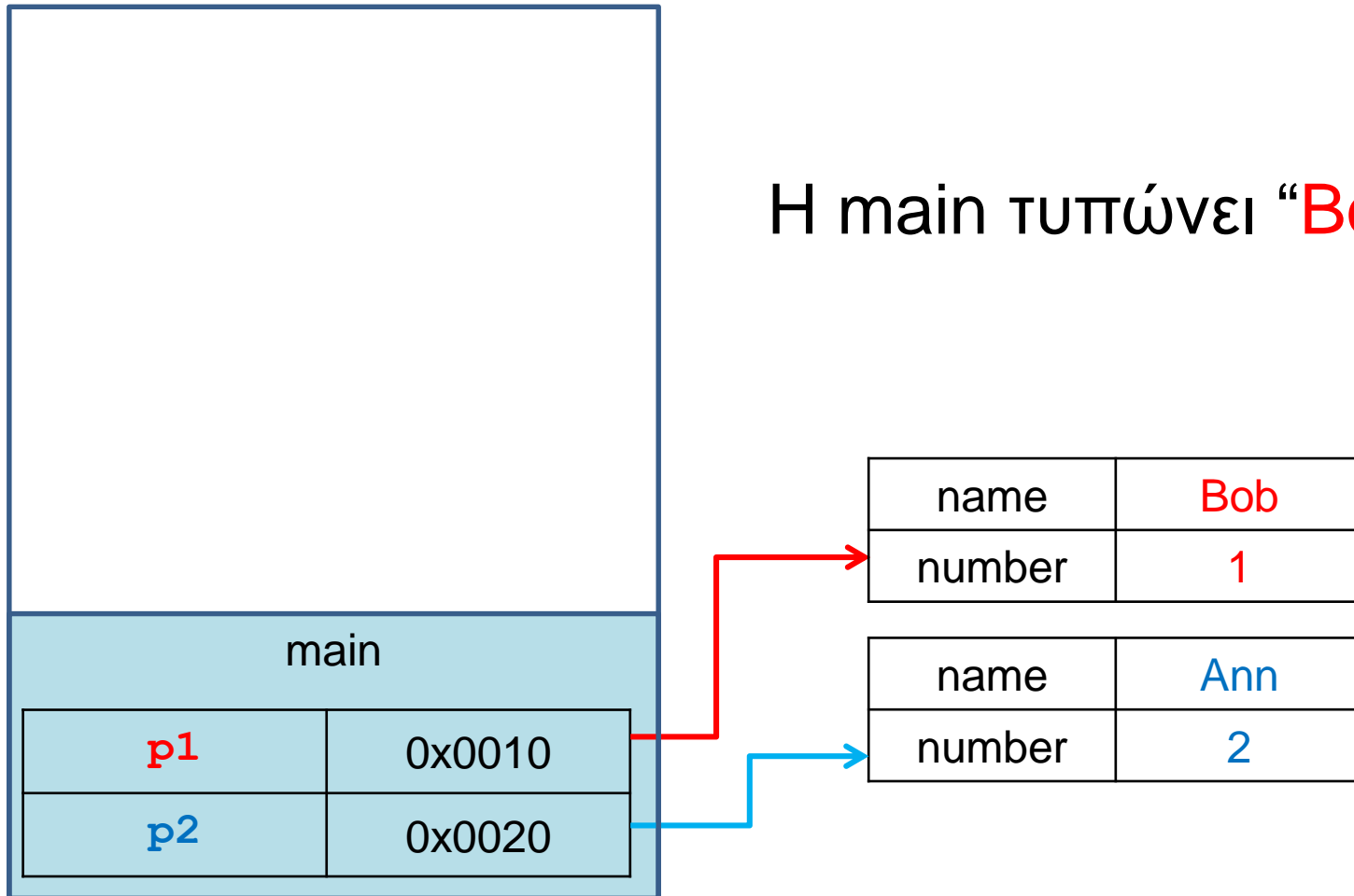


Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος

Η main τυπώνει “**Bob 1**”



Μια ακόμη υλοποίηση της copier

```
public void copier( Person other) {  
    other = new Person(this.name, this.number);  
}
```

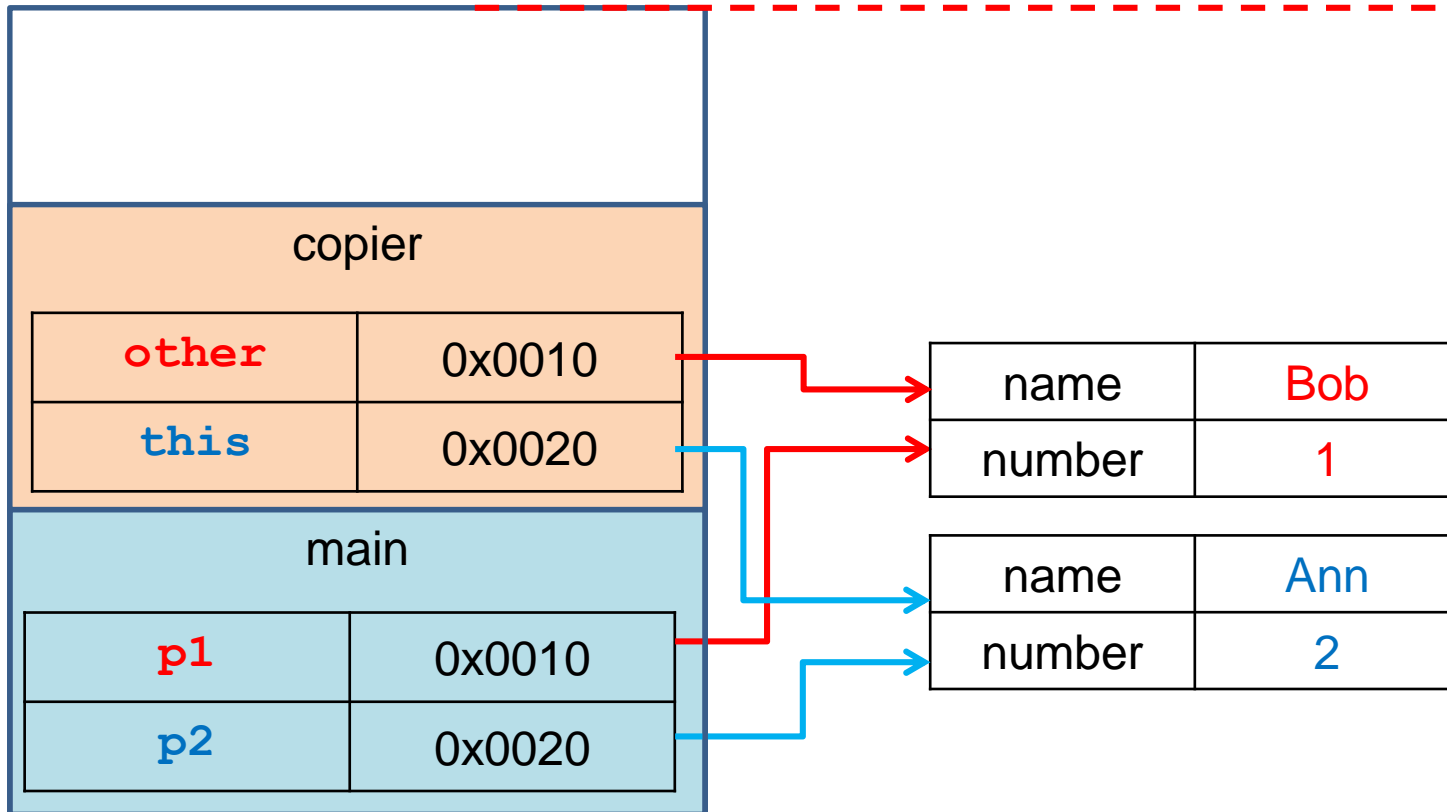
```
public class ClassParameterDemo  
{  
    public static void main(String[] args)  
    {  
        Person p1 = new Person("Bob", 1);  
        Person p2 = new Person("Ann", 2);  
        p2.copier(p1);  
        System.out.println(p1);  
    }  
}
```

Τι θα τυπώσει?

Εξέλιξη του προγράμματος

```
p2.copier(p1);
```

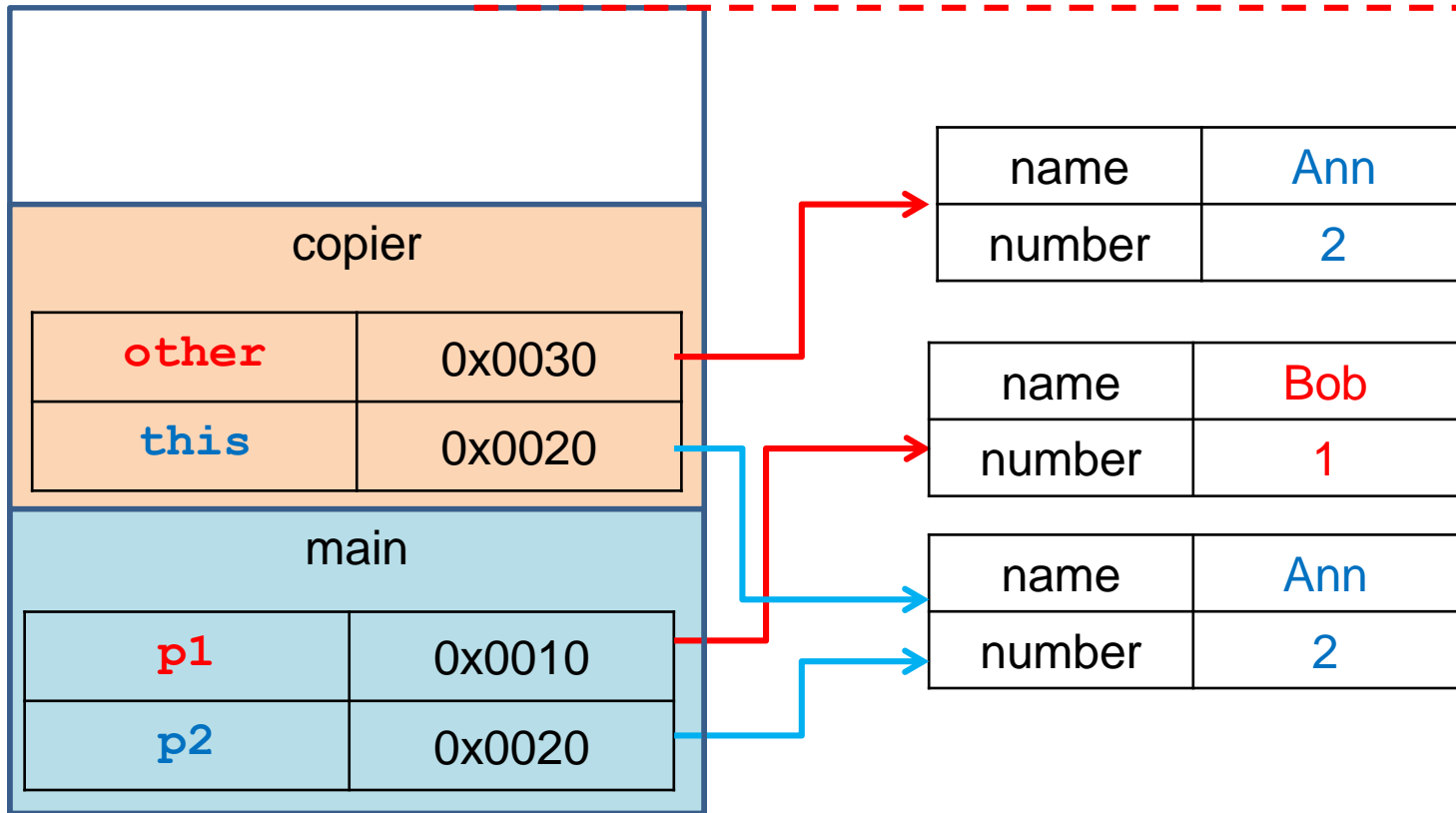
```
public void copier( Person other) {  
    other = new Person(this.name, this.number);  
}
```



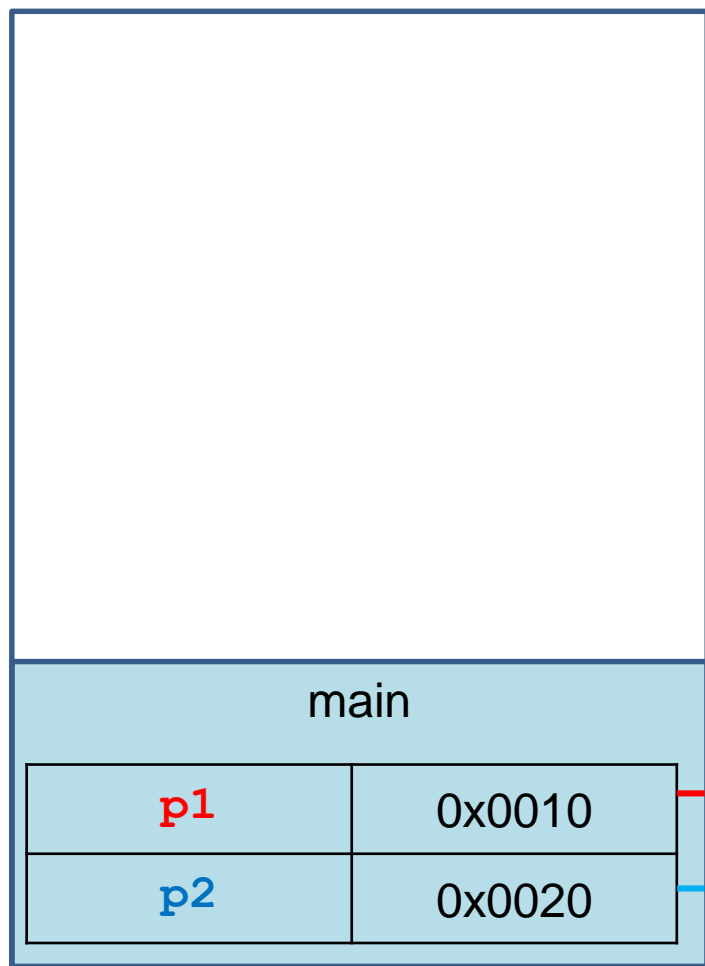
Εξέλιξη του προγράμματος

```
p2.copier(p1);
```

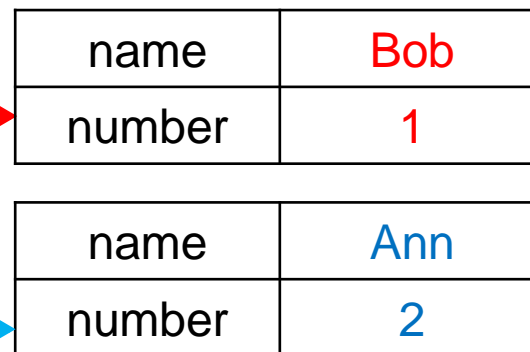
```
public void copier( Person other) {  
    other = new Person(this.name, this.number);  
}
```



Εξέλιξη του προγράμματος



Η `main` τυπώνει **“Bob 1”**



Αλλαγή παραμέτρων

- Στο πρόγραμμα που είδαμε η νέα τιμή του **other** **χάνεται** όταν επιστρέφουμε από την συνάρτηση και η **p1** παραμένει αμετάβλητη.
- Αυτό γιατί το πέρασμα των παραμέτρων γίνεται κατά τιμή, και η μεταβλητή **other** είναι **τοπική**. Ότι αλλαγή κάνουμε στην τιμή της θα έχει εμβέλεια μόνο μέσα στην **copier**.
 - Το νέο αντικείμενο που δημιουργήσαμε στην περίπτωση αυτή θα χαθεί άμα φύγουμε από τη μέθοδο εφόσον δεν υπάρχει κάποια αναφορά σε αυτό.
- Η αλλαγή στην **τιμή** της **other** είναι διαφορετική από την αλλαγή στα **περιεχόμενα** της διεύθυνσης στην οποία δείχνει η **other**
 - Οι αλλαγές στα περιεχόμενα αλλάζουν τον χώρο μνήμης στο σωρό (heap). Οι αλλαγές επηρεάζουν όλες τις αναφορές στο αντικείμενο.

```
class ArrayVar{
    public static void main(String[] args){
        int[] array = {1,2,3};
        int x = 5;

        increment(array);
        for (int i = 0; i < array.length; i ++){
            System.out.print(array[i] + " ");
        }
        System.out.println("");

        increment(x);
        System.out.println("x: " + x);

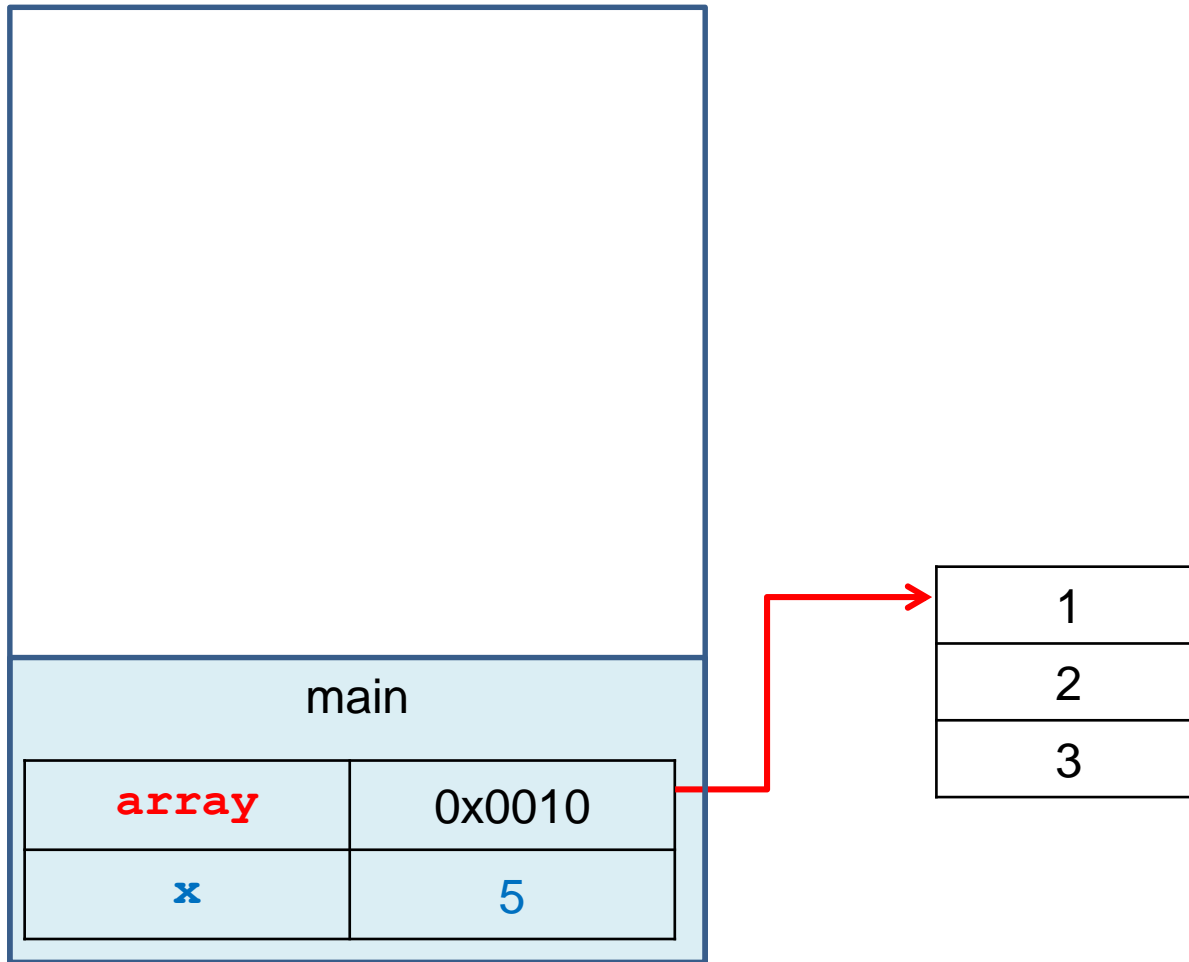
        increment(array[0]);
        System.out.println("array[0] = " + array[0]);
    }

    public static void increment(int[] array){
        for (int i = 0; i < array.length; i ++){
            array[i] ++;
            System.out.print(array[i] + " ");
        }
        System.out.println("");
    }

    public static void increment(int x)
    {
        x ++ ;
        System.out.println("x: " + x);
    }
}
```

Τι θα τυπώσει?

Πέρασμα παραμέτρων



Πέρασμα παραμέτρων

increment(array)

```
public static void increment(int[] array){  
    for (int i = 0; i < array.length; i ++){  
        array[i] ++;  
        System.out.print(array[i] + " ");  
    }  
    System.out.println("");  
}
```

increment

array

0x0010

main

array

0x0010

x

5

1

2

3

Πέρασμα παραμέτρων

increment(array)

```
public static void increment(int[] array){  
    for (int i = 0; i < array.length; i ++){  
        array[i] ++;  
        System.out.print(array[i] + " ");  
    }  
    System.out.println("");  
}
```

increment

array

0x0010

main

array

0x0010

x

5

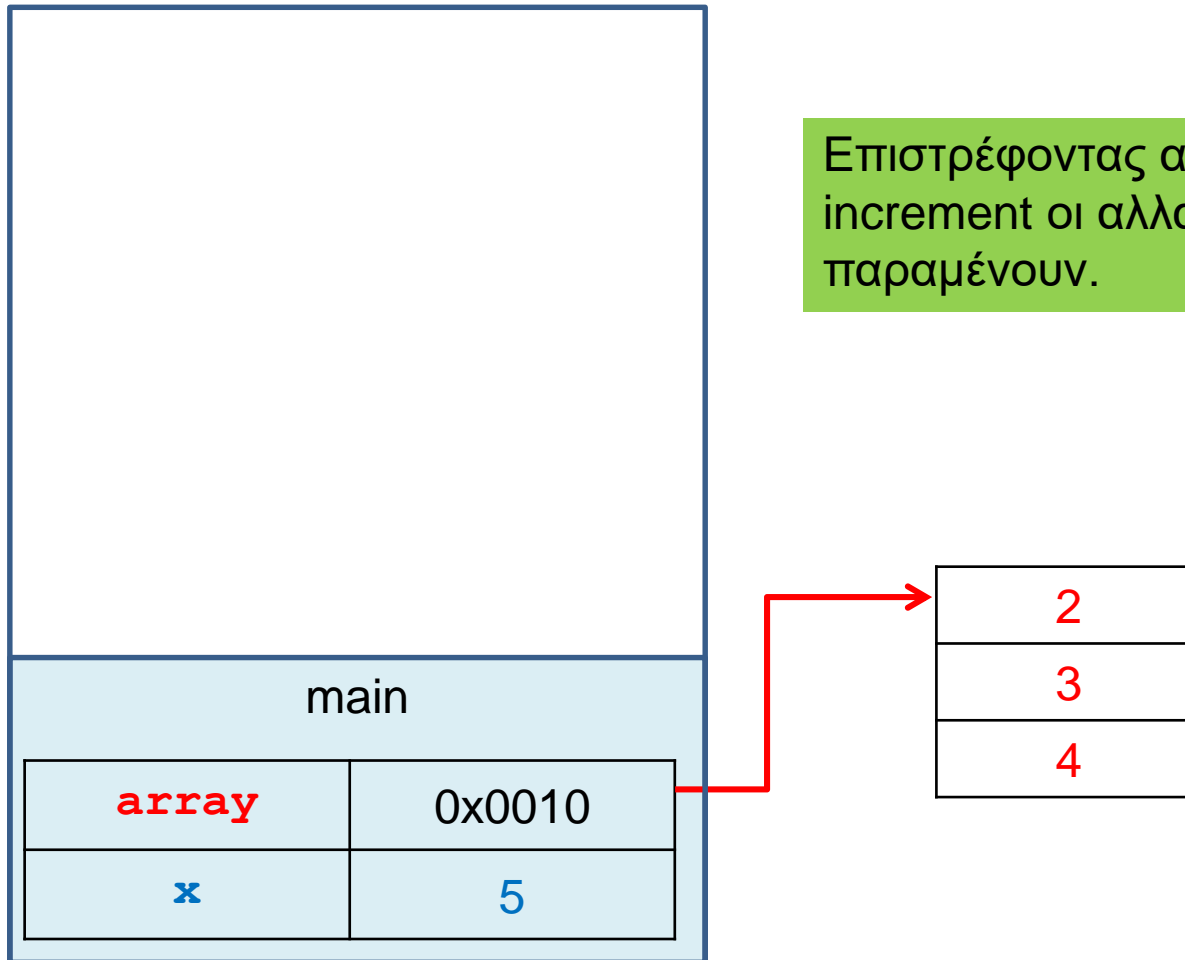
2

3

4

Πέρασμα παραμέτρων

Επιστρέφοντας από την μέθοδο `increment` οι αλλαγές στον πίνακα παραμένουν.



Πέρασμα παραμέτρων

increment(x)

increment

x

5

main

array

0x0010

x

5

```
public static void increment(int x) {  
    x ++;  
    System.out.println("x: " + x);  
}
```

2

3

4

Πέρασμα παραμέτρων

increment(x)

```
public static void increment(int x) {  
    x ++;  
    System.out.println("x: " + x);  
}
```

increment

x

6

main

array

0x0010

x

5

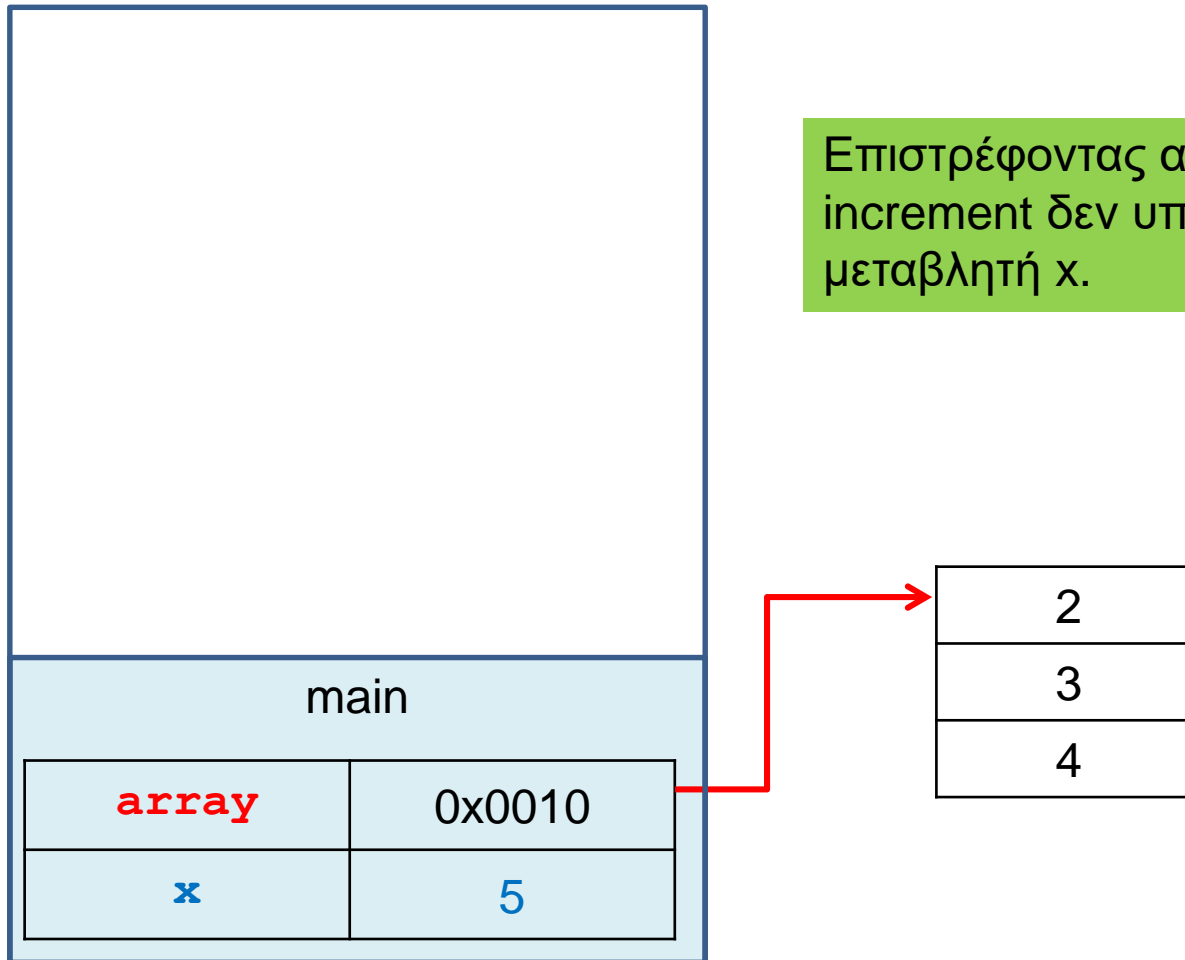
2

3

4

Πέρασμα παραμέτρων

Επιστρέφοντας από την μέθοδο `increment` δεν υπάρχουν αλλαγές στη μεταβλητή `x`.



Πέρασμα παραμέτρων

`increment(array[0])`

```
public static void increment(int x) {  
    x ++;  
    System.out.println("x: " + x);  
}
```

increment

`x`

2

main

`array`

0x0010

`x`

5

2

3

4

Πέρασμα παραμέτρων

increment(array[0])

```
public static void increment(int x) {  
    x ++;  
    System.out.println("x: " + x);  
}
```

increment

x

3

main

array

0x0010

x

5

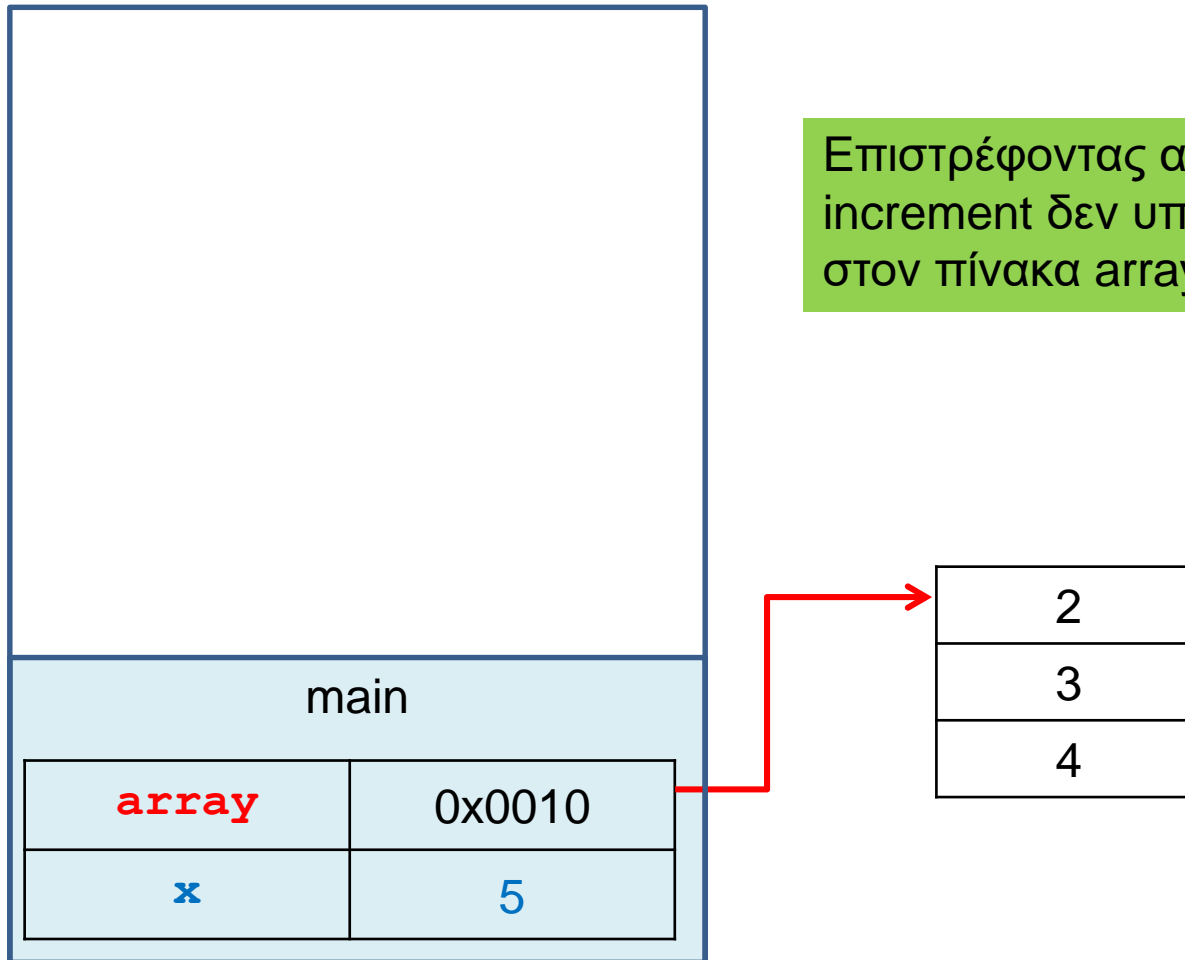
2

3

4

Πέρασμα παραμέτρων

Επιστρέφοντας από την μέθοδο `increment` δεν υπάρχουν αλλαγές στη στον πίνακα `array`.



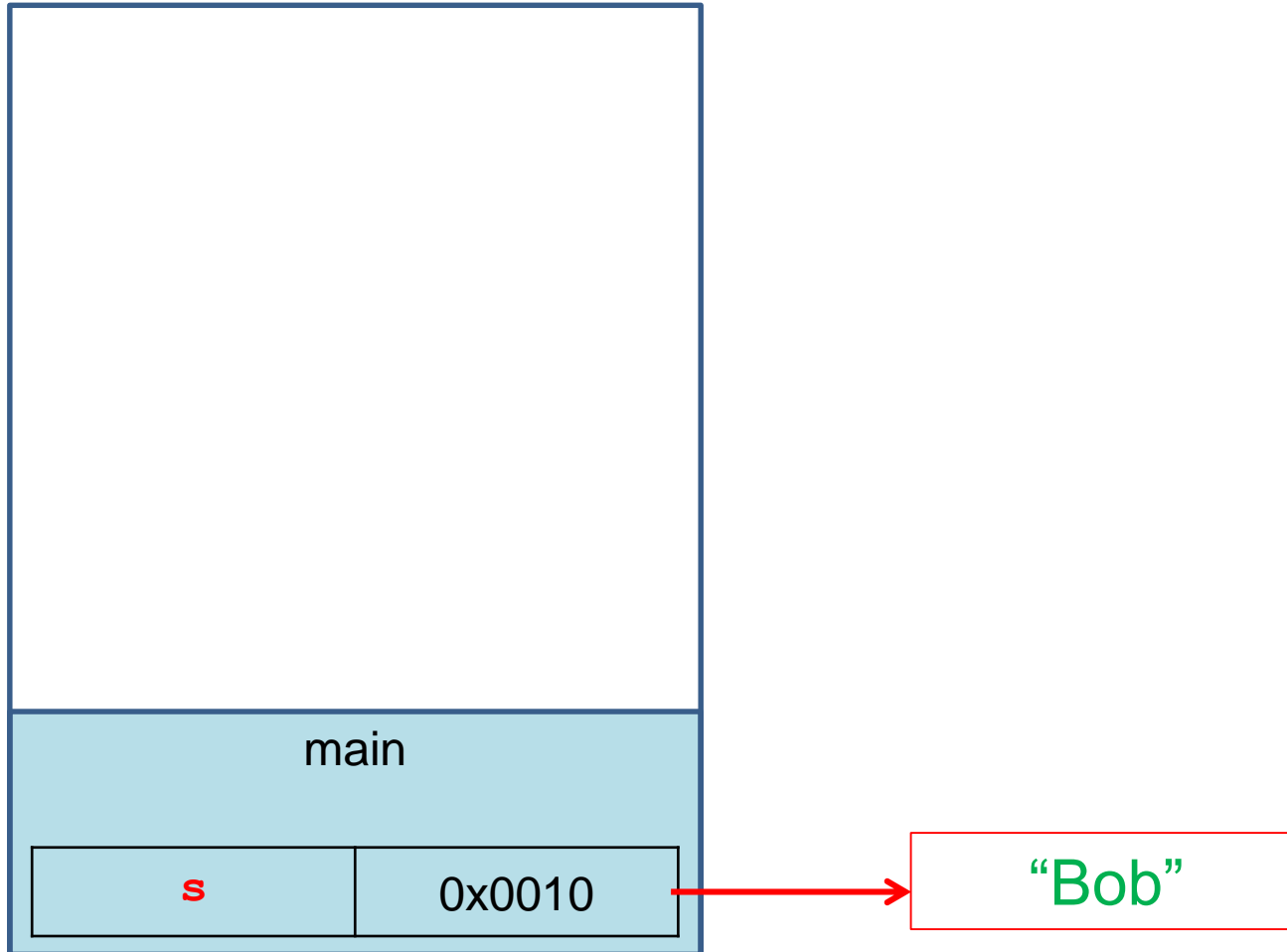
Άλλο ένα παράδειγμα

```
public class StringParameterDemo
{
    public static void main(String[] args)
    {
        String s = "Bob";
        changeString(s);
        System.out.println(s);
    }

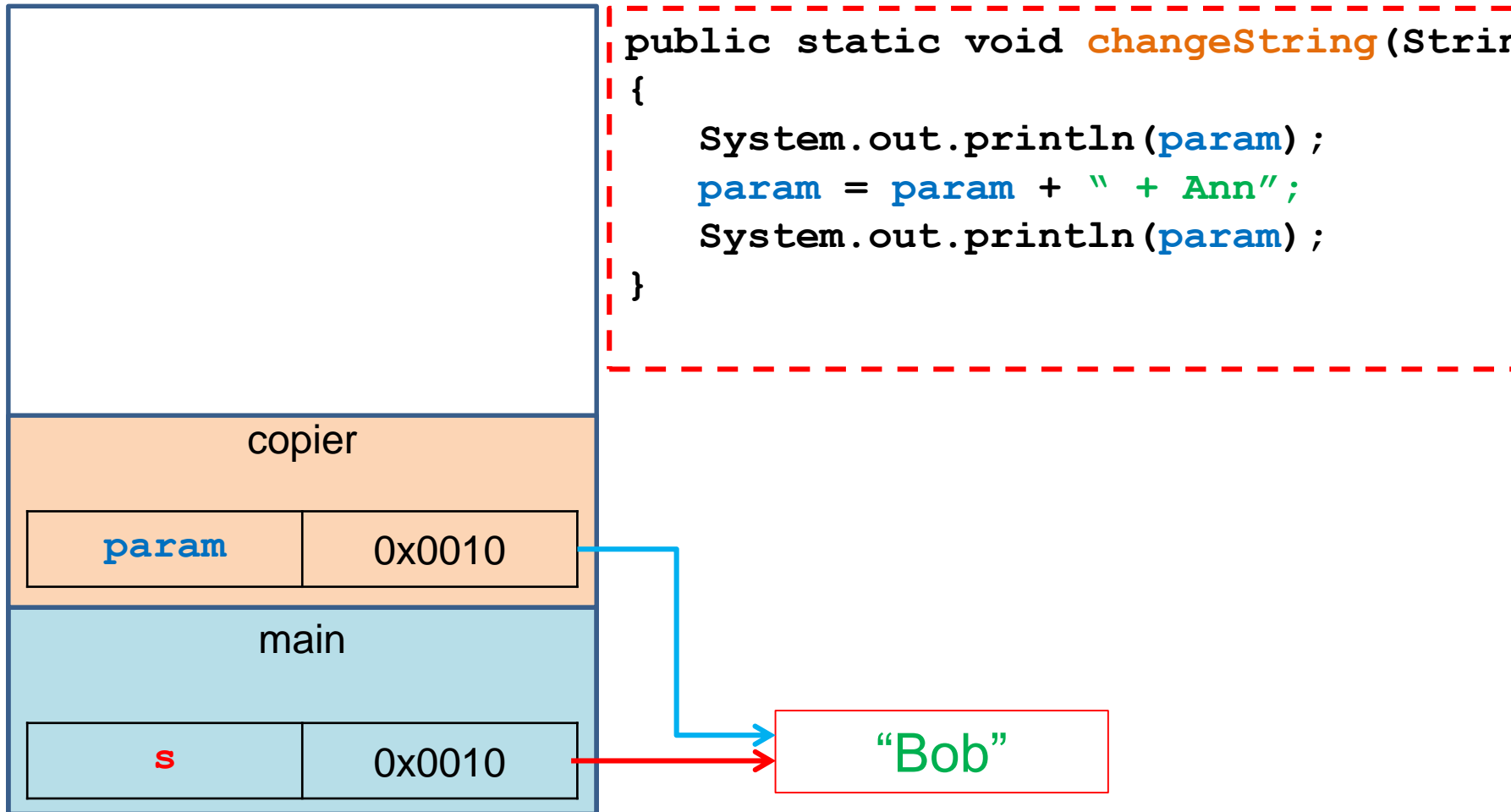
    public static void changeString(String param)
    {
        System.out.println(param);
        param = param + " + Ann";
        System.out.println(param);
    }
}
```

Τι θα τυπώσει?

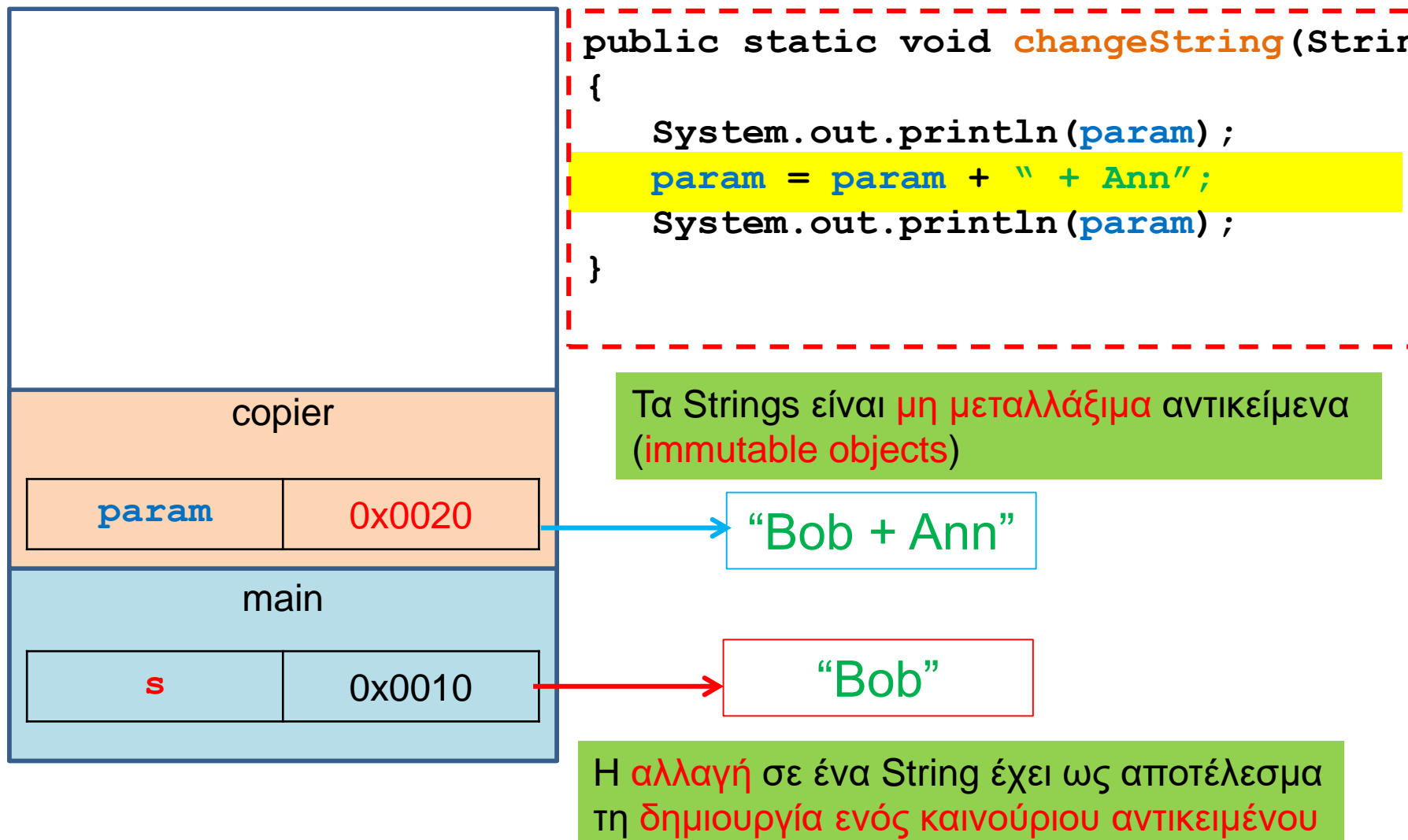
Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος



String Interning

- Στην Java για κάθε **string value** που εμφανίζεται δημιουργείται ένα **αντικείμενο**, το οποίο ονομάζεται **intern string**, και το οποίο κρατάει αυτή την τιμή.
- Για αυτό και οι αλφαριθμητικές σταθερές μπορούν να χρησιμοποιηθούν και σαν αντικείμενα. Π.χ. μπορούμε να καλέσουμε:

```
"java".length()
```

- Αυτό μπορεί να προκαλέσει μπερδέματα με ελέγχους ισότητας.

Ισότητα String

Τι θα εκτυπωθεί?

```
import java.util.Scanner;

class StringEquality{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);

        String x = input.next();
        String z = new String("java");
        String y = "java";

        System.out.println("1. " + (x == "java"));
        System.out.println("2. " + (y == "java"));
        System.out.println("3. " + (z == "java"));
        System.out.println("4. " + x.equals("java"));
        System.out.println("5. " + y.equals("java"));
        System.out.println("6. " + z.equals("java"));
    }
}
```

1. false

2. true

3. false

4. true

5. true

6. true

Για την σύγκριση Strings **ΠΑΝΤΑ** χρησιμοποιούμε την μέθοδο **equals**.

String Interning

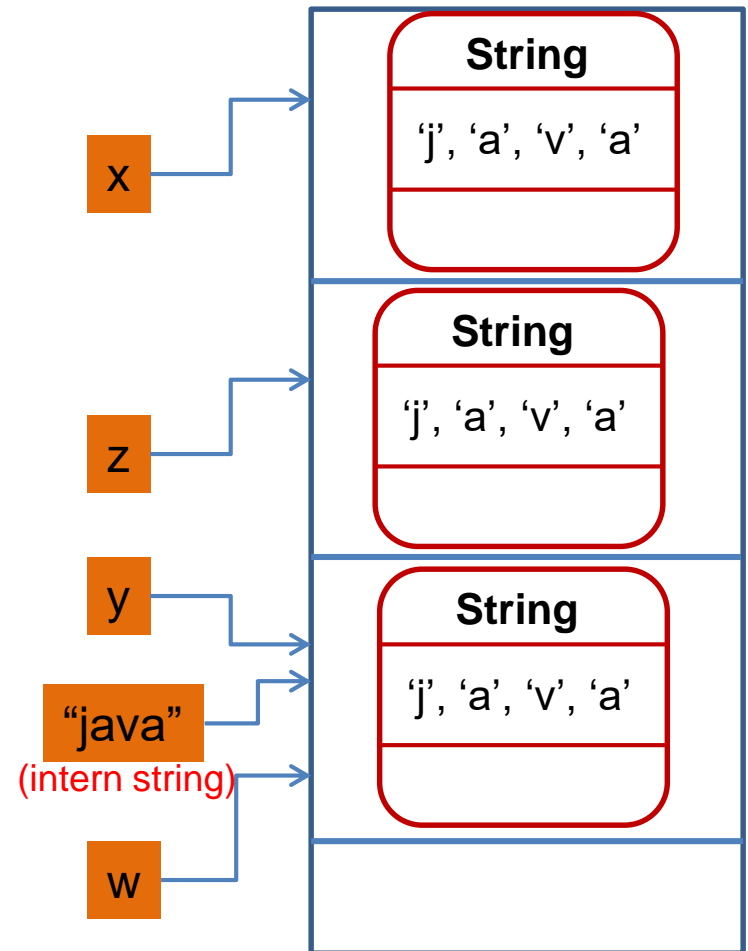
- Όταν γίνεται η εκχώρηση της τιμής "java" δημιουργείται ένα **intern string**, και το οποίο κρατάει αυτή την τιμή.

- Η εντολή

```
String y = "java";
```

κάνει το **y** να δείχνει στη θέση που είναι αποθηκευμένη η τιμή "java"

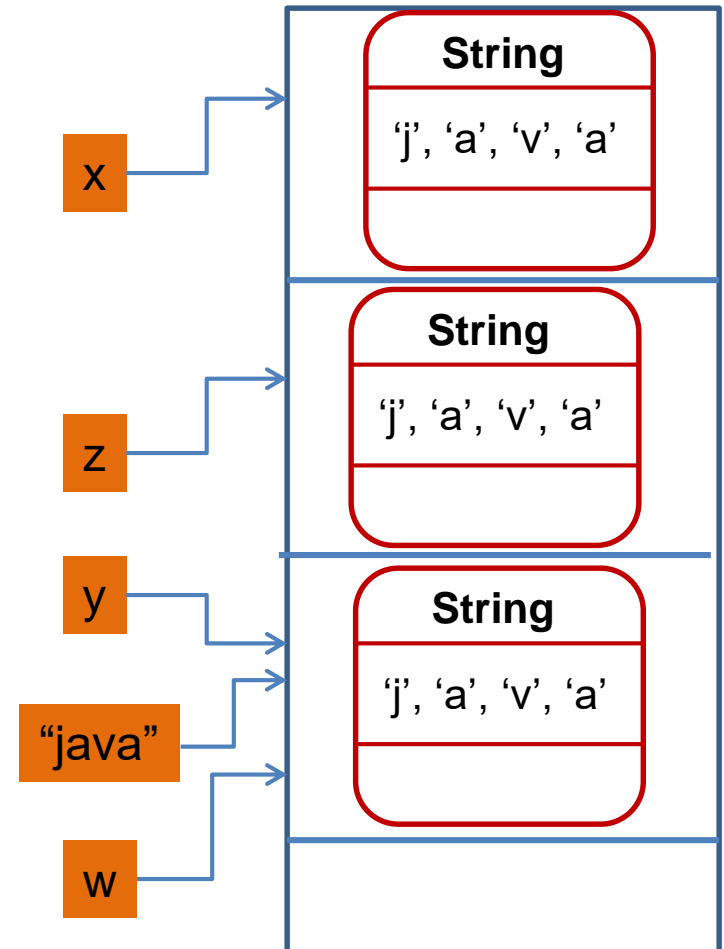
```
String x = input.next();  
String z = new String("java");  
String y = "java";  
String w = "java";
```



String Interning

```
String x = input.next();  
String z = new String("java");  
String y = "java";  
String w = "java";  
System.out.println((y == "java"));
```

- Ο τελεστής `==` μεταξύ δύο αντικειμένων εξετάζει αν πρόκειται για την ίδια θέση μνήμης.
- Γι αυτό `(y == "java")` επιστρέφει true.



Equals

- Έχουμε πει ότι όταν ελέγχουμε ισότητα μεταξύ αντικειμένων (π.χ., Strings) πρέπει να γίνεται μέσω της μεθόδου **equals** και όχι με το **==**
- Η συζήτηση με τις αναφορές εξηγεί γιατί η σύγκριση με **==** δε δουλεύει
- Η σύγκριση με **==** συγκρίνει αν δύο **αναφορές** είναι ίδιες και **όχι** αν **τα περιεχόμενα** των θέσεων μνήμης στις οποίες δείχνουν οι αναφορές είναι ίδια.

```
public class Person
{
    private String name;
    private int number;

    public Person(String initName, int initNumber){
        name = initName;
        number = initNumber;
    }

    public void set(String newName, int newNumber){
        name = newName;
        number = newNumber;
    }

    public boolean equals(Person other){
        return this.name.equals(other.name) && this.number == other.number;
    }

    public void copier(Person other) {
        other.name = name;
        other.number = number;
    }

    public String toString( ){
        return (name + " " + number);
    }
}
```

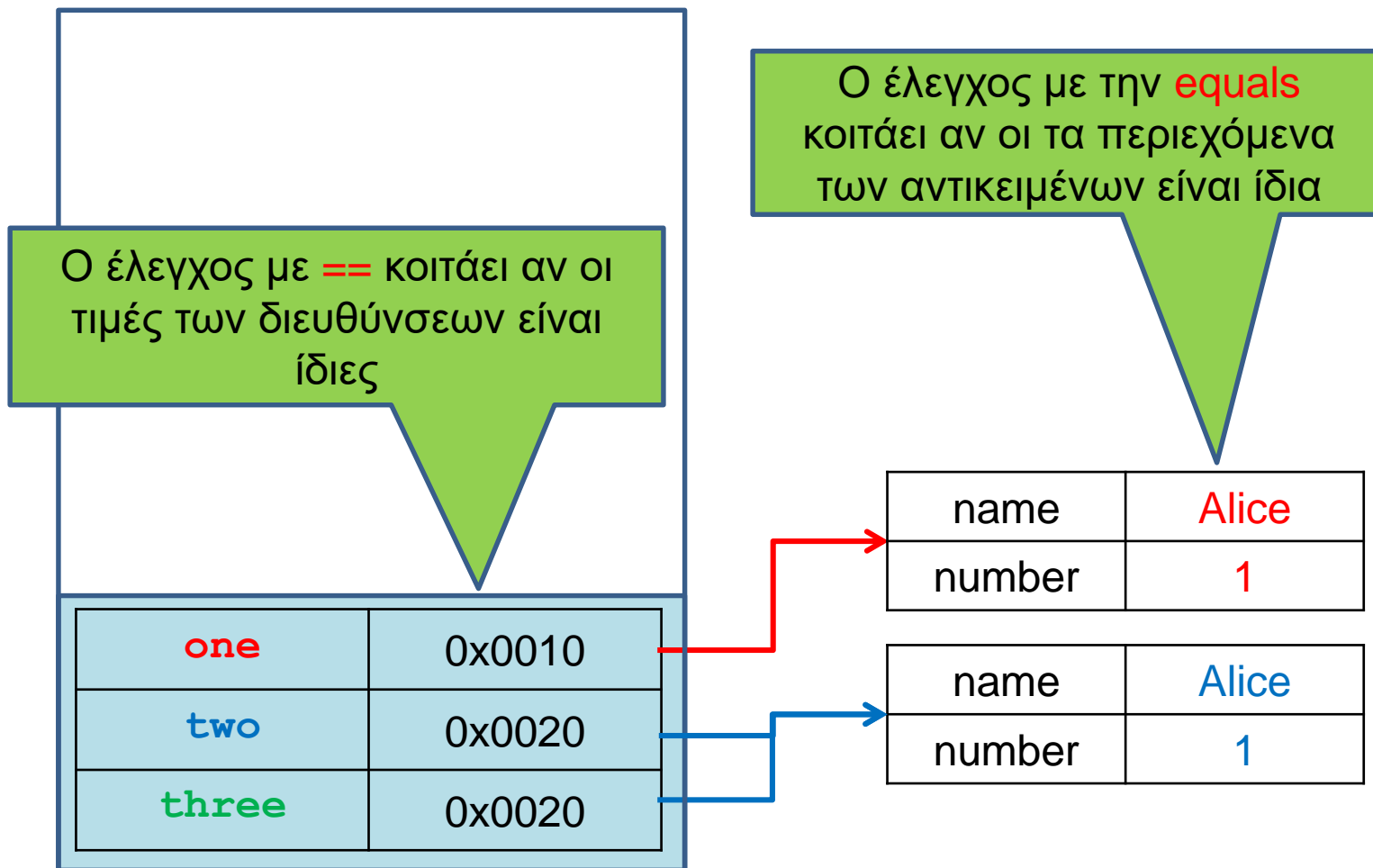
Παράδειγμα

- Τι θα τυπώσει ο παρακάτω κώδικας?

```
Person one = new Person("Alice", 1);
Person two = new Person("Alice", 1);
Person three = two;
System.out.println(one == two);
System.out.println(two == three);
System.out.println(one == three);
System.out.println(one.equals(two));
System.out.println(two.equals(three));
System.out.println(one.equals(three));
```

false
true
false
true
true
true

Εξήγηση




```
class StringClass
```

```
{  
    String s = "abc";
```

```
    public void changeObject(StringClass other) {
```

```
        if (this.s == other.s) {  
            System.out.println("Same");
```

```
        }else {  
            System.out.println("Different");
```

```
        }
```

```
        String local = new String("local");
```

```
        other.s = local;
```

```
        local = "local";
```

```
        s = local;
```

```
        if (this.s == other.s) {  
            System.out.println("Same");
```

```
        }else {  
            System.out.println("Different");
```

```
        }
```

```
    }
```

```
}
```

Η ανάθεση String σταθεράς έχει αποτέλεσμα την δημιουργία ενός intern string στο οποίο δείχνουν όλα τα strings στα οποία ανατίθεται η σταθερά.

Η ανάθεση String σταθεράς είναι διαφορετική από τη δημιουργία αντικειμένου με new

Η σταθερά δημιουργεί ένα νέο **intern String**

Τι θα τυπώσει?

```
class StringTest{
```

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

```
        StringClass obj1 = new StringClass();
```

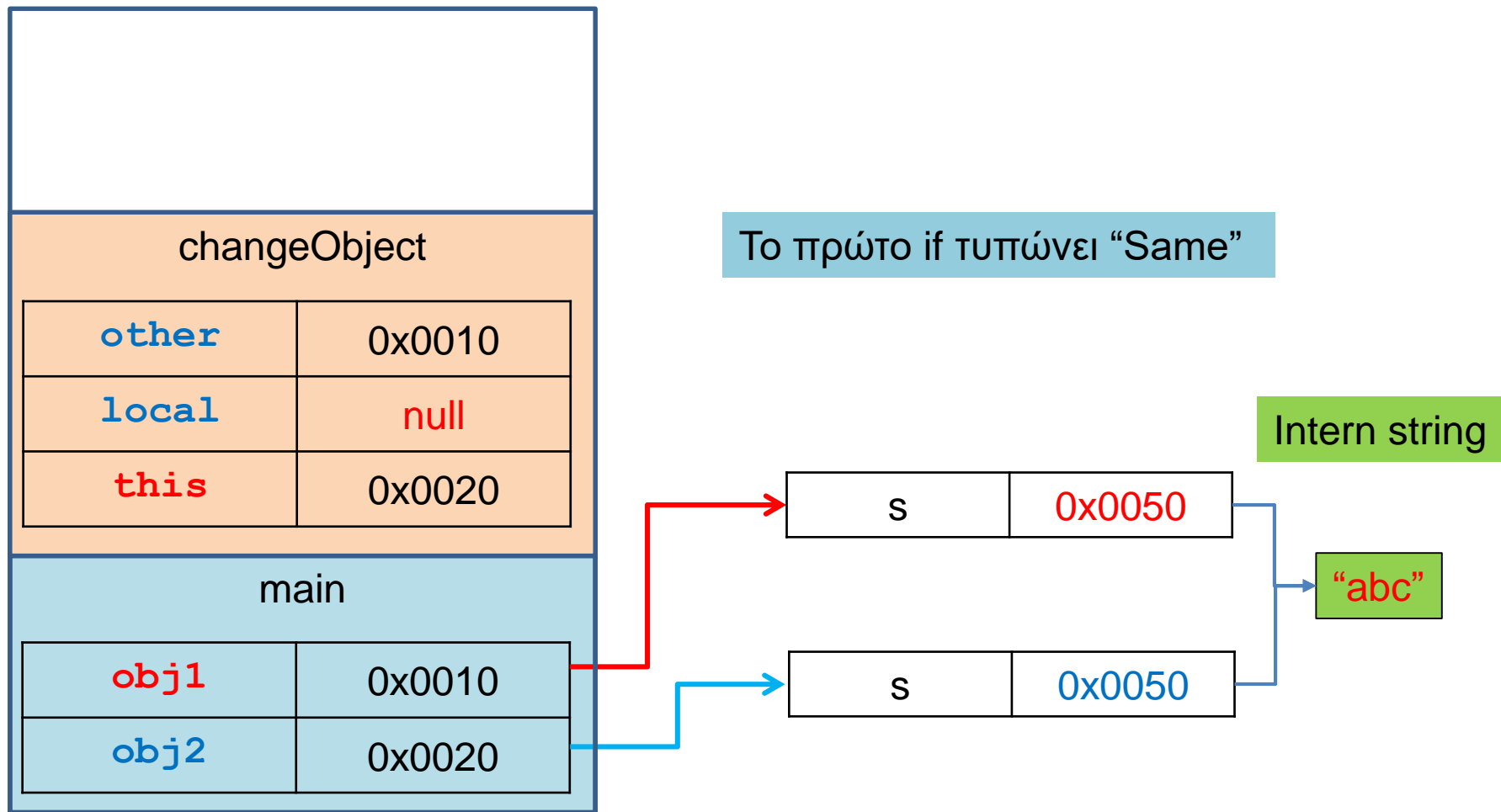
```
        StringClass obj2 = new StringClass();
```

```
        obj2.changeObject(obj1);
```

```
    }
```

```
}
```

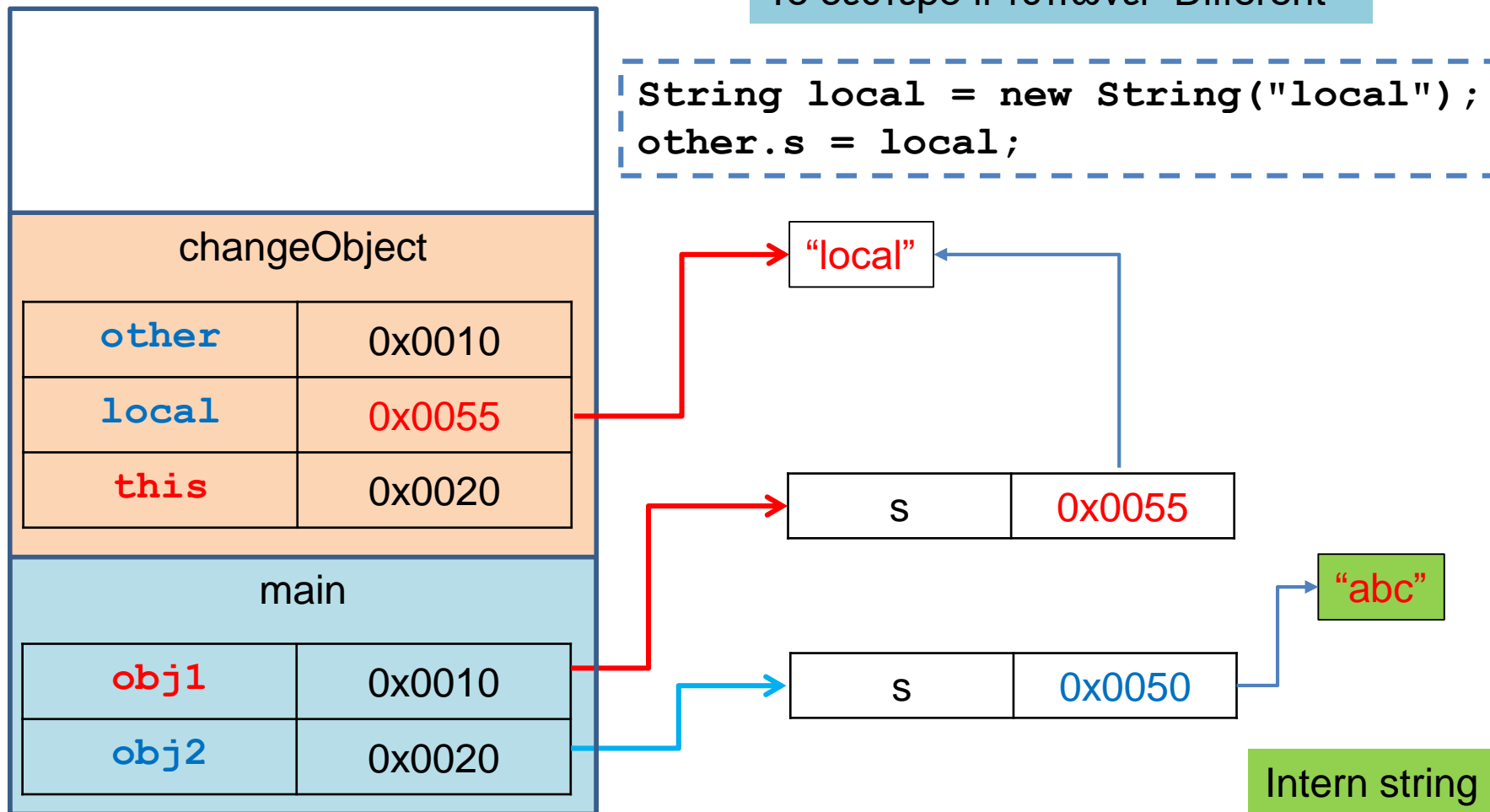
Εξέλιξη του προγράμματος



Εξέλιξη του προγράμματος

Το δεύτερο if τυπώνει "Different"

```
String local = new String("local");  
other.s = local;
```



Εξέλιξη του προγράμματος

Το δεύτερο if τυπώνει "Different"

```
local = "local";  
s = local;
```

