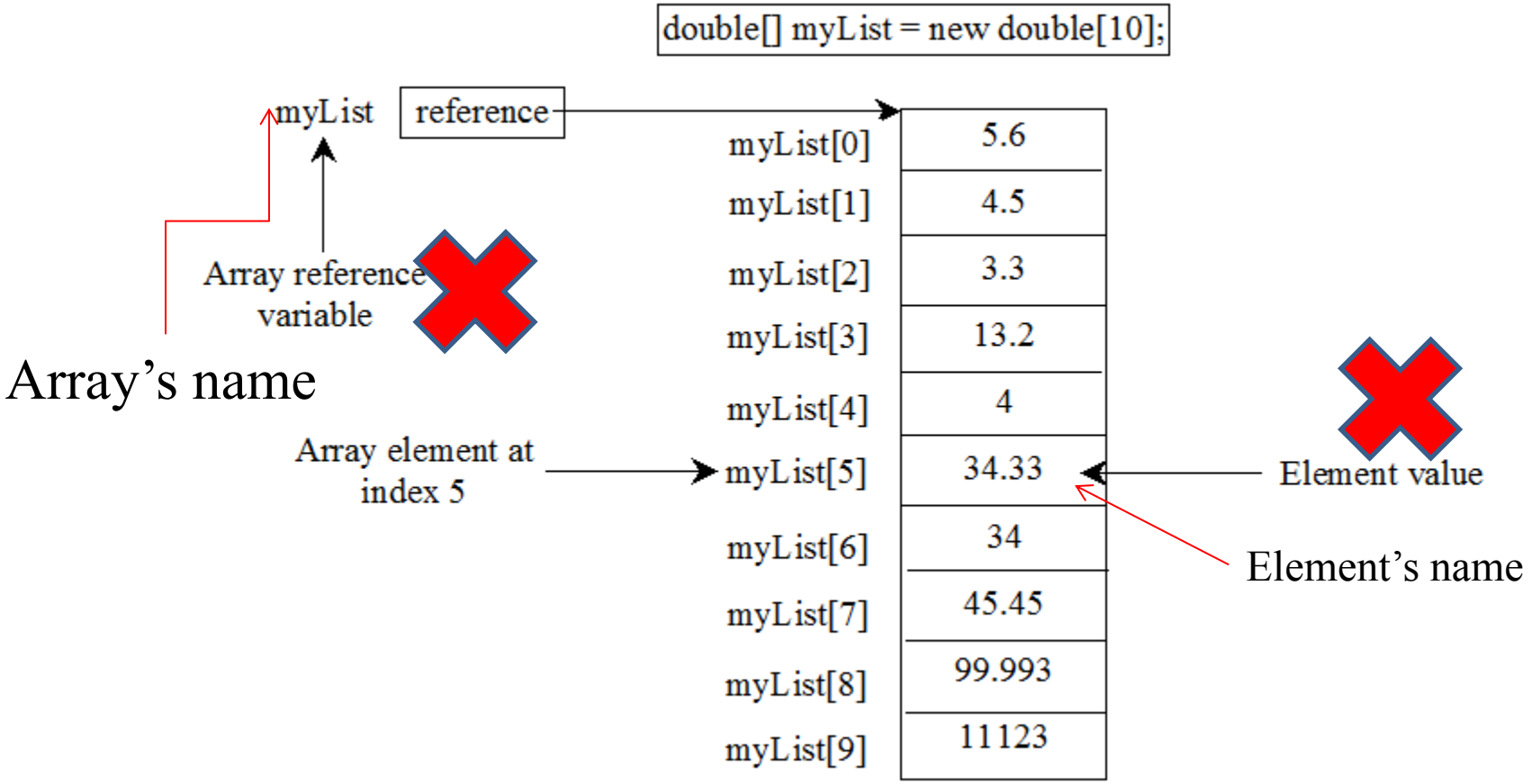
Arrays

Introducing Arrays



The Length of an Array

Once an array is created, its size is fixed. It cannot be changed. You can find its size using

```
arrayRefVar.length
```

For example,

```
myList.length returns 10
```

Enhanced for Loop (for-each loop)

For example, the following code displays all elements in the array myList:

```
for (double value: myList)
  System.out.println(value);
```

In general, the syntax is

```
for (elementType value: arrayRefVar) {
  // Process the value
}
```

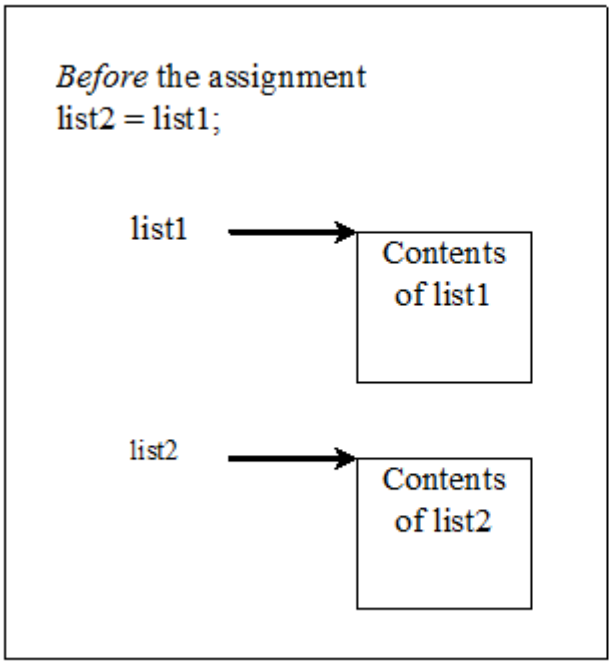
You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

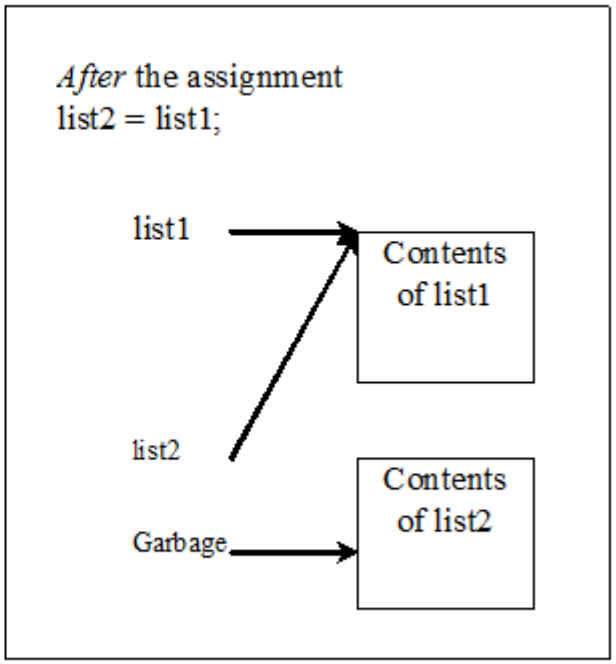
Copying Arrays

You need to duplicate an array or a part of an array.

In such cases you could attempt to use the assignment statement (=)

list2 = list1;





Passing Arrays to Methods

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

Invoke the method

```
int[] list = {3, 1, 2, 6, 4, 2};
printArray(list);
```

Invoke the method

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Anonymous array

Pass By Value

Java uses *pass by value* to pass arguments to a method. There are important differences between passing a value of variables of primitive data types and passing arrays.

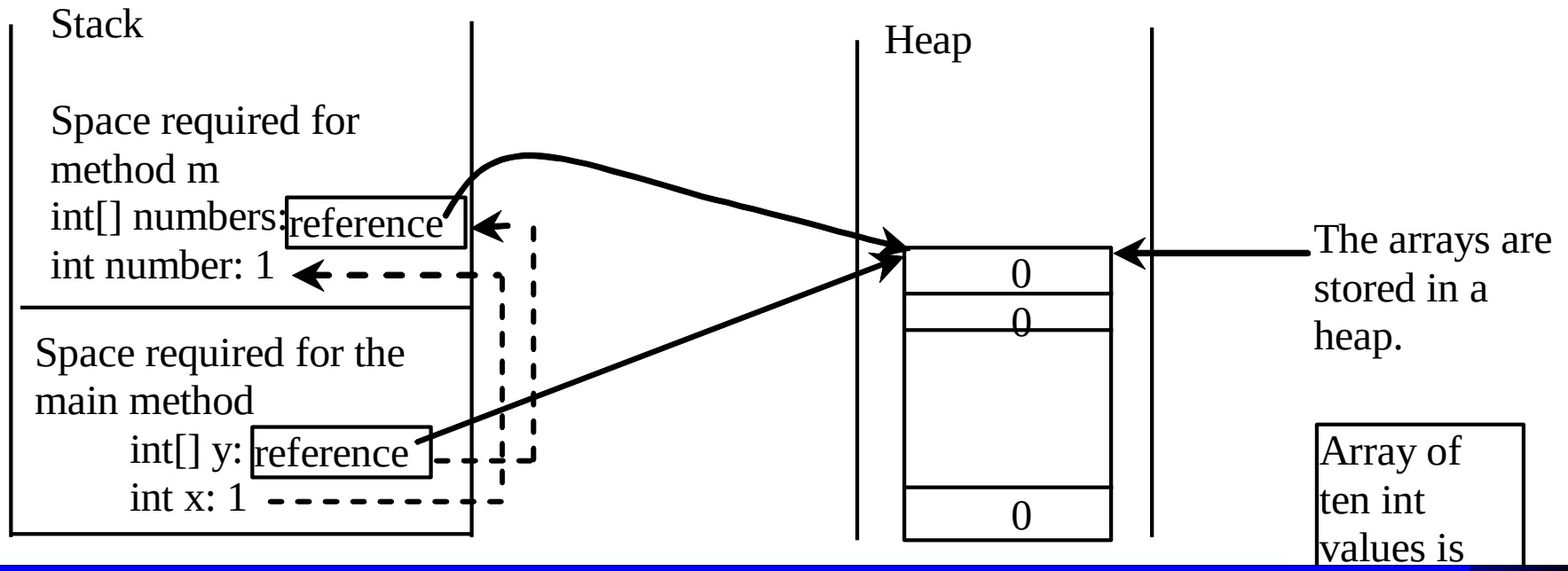
- For a parameter of a primitive type value, the actual value is passed. Changing the value of the local parameter inside the method does not affect the value of the variable outside the method.

- For a parameter of an array type, the name of the parameter contains a “reference” to an array; this reference is passed to the method. Any changes to the array that occur inside the method body will affect the original array that was passed as the argument.

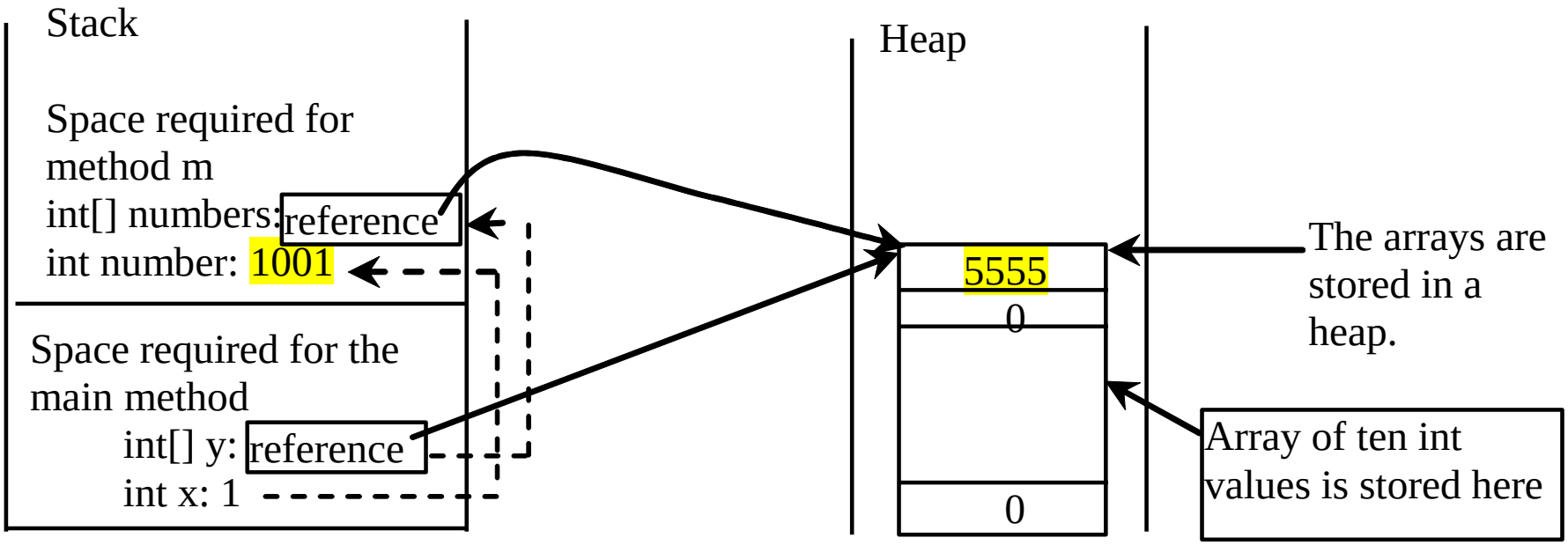
Simple Example

```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```

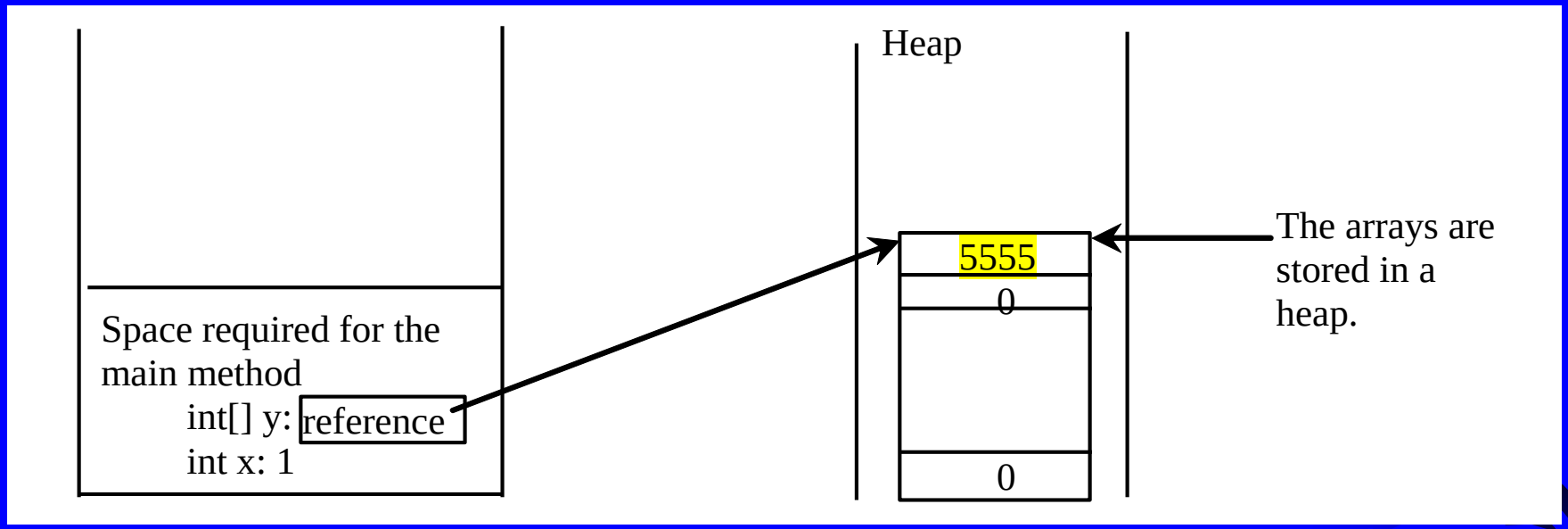
Call Stack


Stack
Space required for method m
int[] numbers: reference
int number: 1
Space required for the main method
int[] y: reference
int x: 1
Heap
0
0
0
The arrays are stored in a heap.
Array of ten int values is

Call Stack

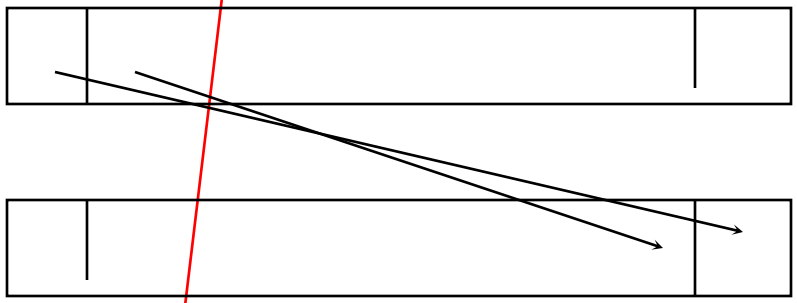

Stack
Space required for method m
int[] numbers: reference
int number: 1001
Space required for the main method
int[] y: reference
int x: 1
Heap
5555
0
0
The arrays are stored in a heap.
Array of ten int values is stored here

Heap


Heap
Space required for the
main method
int[] y: reference
int x: 1
5555
0
0
The arrays are
stored in a
heap.

Returning an Array from a Method

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1;
       i < list.length; i++, j--) {
    result[j] = list[i];
  }

  return result;
}
```



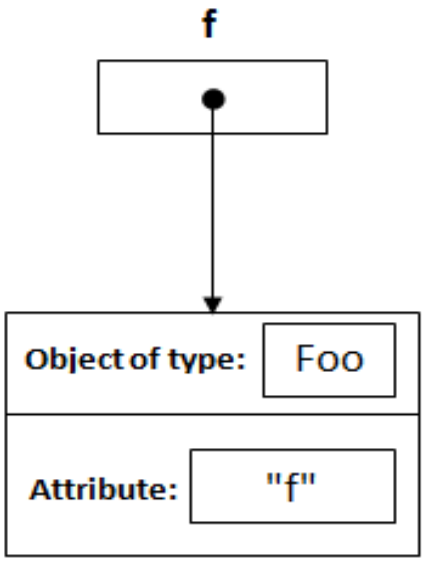

```
int[] list2 = reverse(new int[]{1, 2, 3, 4, 5, 6});
```

Parameter Passing One More Time

```
Public class Main{
        public static void main(String[] args){
                Foo f = new Foo(“f”);
                changeReference(f); modifyReference(f)}

        public static void changeReference(Foo a){
                Foo b = new Foo(“b”); a = b}

        public static void modifyReference(Foo c){
                c.setAttribute(“c”);}
}
```

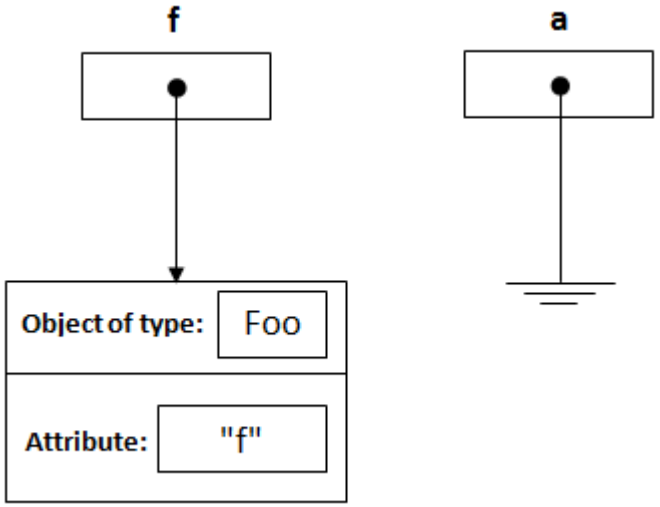
Declaring a reference named f of type Foo and assign it to a new object of type Foo with an attribute "f".

```
Foo f = new Foo("f");
```



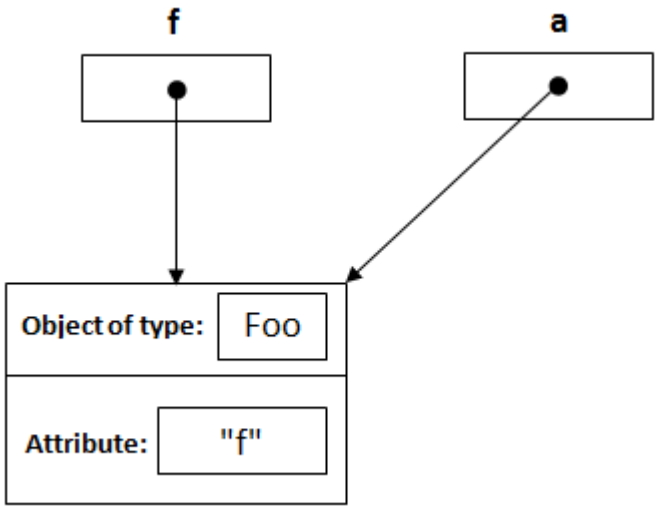
From the method side, a reference of type Foo with a name a is declared and it's initially assigned to null.

public static void changeReference(Foo a)



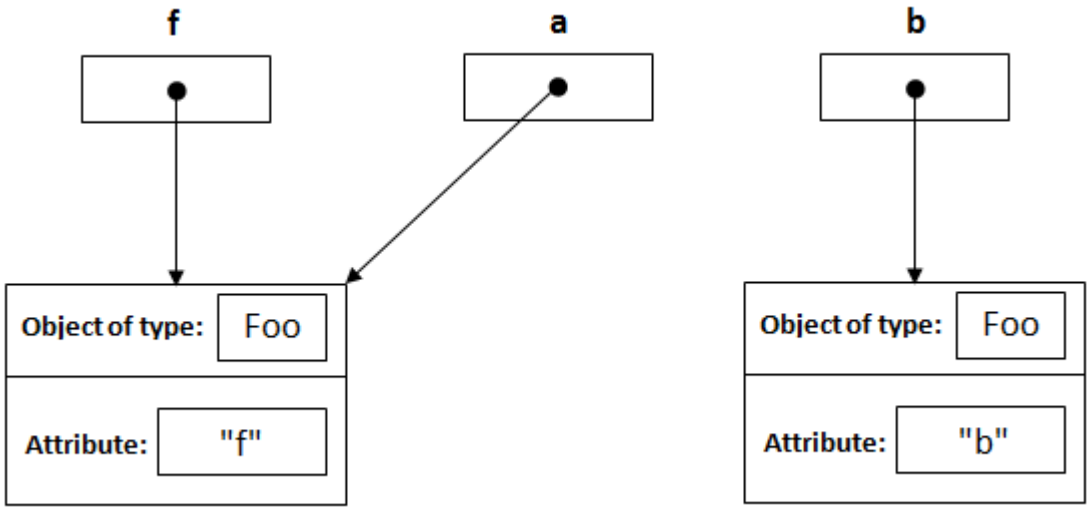
As you call the method changeReference, the reference a will be assigned to the object which is passed as an argument.

changeReference(f);

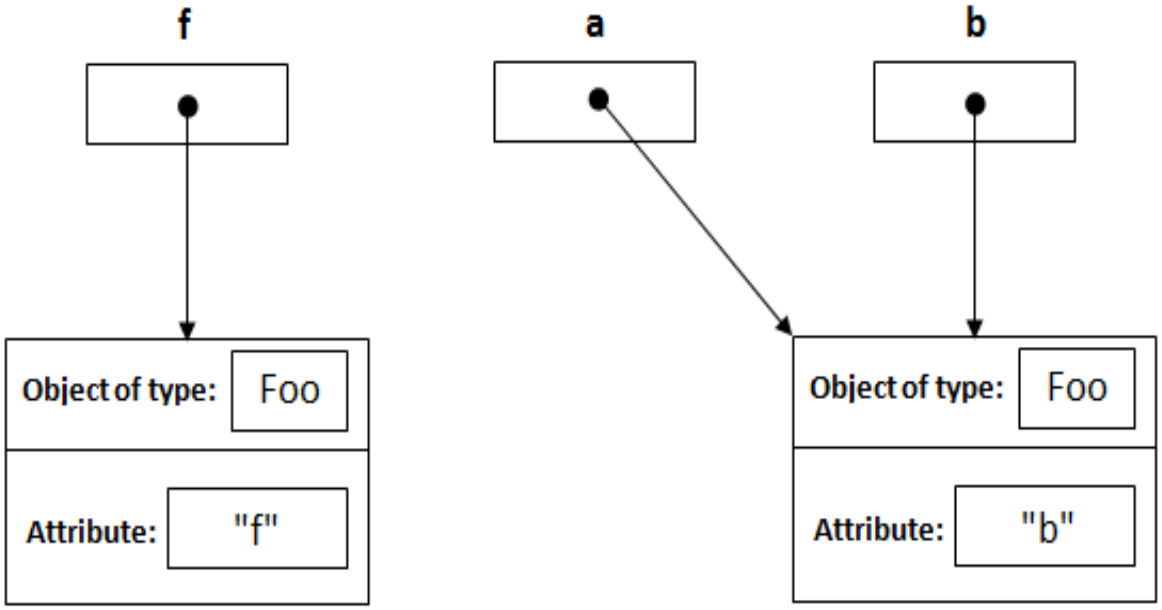


Declaring a reference named b of type Foo and assign it to a new object of type Foo with an attribute "b".

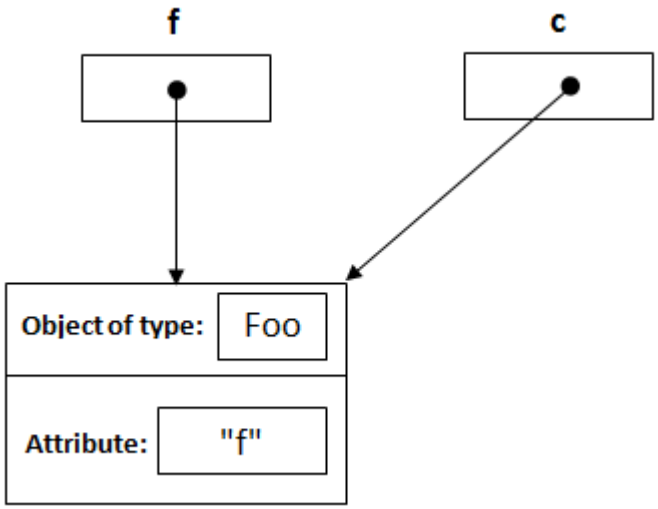
```
Foo b = new Foo("b");
```



a = b is re-assigning the reference a NOT f to the object whose its attribute is "b".



As you call modifyReference(Foo c) method, a reference c is created and assigned to the object with attribute "f".



Obviously we have skipped the equivalent of step two for the sake of brevity.

c.setAttribute("c") will change the attribute of the object that reference c points to, and it's same object that reference f points to.