

Methods - Arguments - Parameters

Do now

Imagine you are coding a complex algorithm that requires many lines of code.

What could be some strategies to keep your code clean, simple, organized and easy to read?



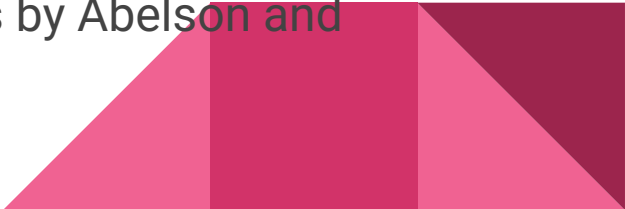
Organizing code

Organizing your code is not about communication with the computer.

Organizing your code is to make people understand the code, so it can become easy to maintain and evolve it.

“Programs should be written for people to read, and only incidentally for machines to execute.”

— Structure and Interpretation of Computer Programs by Abelson and Sussman



Organizing code

- Organize code in classes
- Raise errors when we know a certain event cannot happen in our program (example: a variable cannot have a specific value or meet a certain condition)
- Split code into smaller methods. This will help to:
 - Easily maintain your code
 - Facilitates testing and debugging
 - Fix issues
 - Prevent duplicate code. Reuse code



Organizing code

Avoid hard coding: Hard coding means to write data (values) directly in your code.

Add comments to your code explaining the code especially the parts that are not obvious.

Choose names that make sense for classes, methods and variables.



Argument vs. Parameter

Argument: It is a value passed to a method when the method is called. An argument when passed with a function replaces with those variables which were used during the function definition and the function is then executed with these values.

```
public static int multiply(int a, int b){  
    return a * b;  
}
```

```
// When you invoke the method the variables  
// x and y are arguments
```

```
int product = multiply(x, y);
```

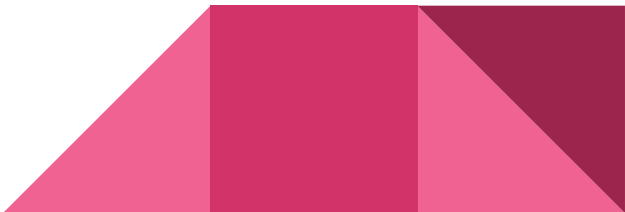


Argument vs. Parameter

Parameter

A parameter is a variable used to define a particular value during a function definition. variables that are being used in the running of that function.

```
// the variables a and b are parameters
public static int multiply(int a, int b)
{
    return a * b;
}
```



Argument vs. Parameter


Argument	Parameter
When a function is called, the values that are passed in the call are called arguments.	The values which are written at the time of the function prototype and the definition of the function.
These are used in function call statement to send value from the calling function to the called function.	These are used in function header of the called function to receive the value from the arguments.
During the time of call each argument is always assigned to the parameter in the function definition.	Parameters are local variables which are assigned value of the arguments when the function is called
They are also called Actual Parameters	They are also called Formal Parameters

Method Signature in Java

Method Signature:

- It is defined as the **structure of a method** designed by the programmer.
- It is the combination of a **method's name and its parameter list**.

Warning:

- A class **cannot have two methods with the same signature** (compilation error is thrown).
 - Method signature **does not include the return type** of a method.
- 

Why do we need a Method Signature in Java?

Sometimes we will need methods that have the same name, but different parameter list.

Two parameter lists are different if either of the following conditions are satisfied:

- Different number of parameters in both parameter lists
- At least one parameter is of different type

This event of declaring two methods with same name but different parameter lists (or signature) is called **Method Overloading**.



Method Overloading Example

```
public static int sum(int x, int y){
```

```
    return x + y;
```

```
}
```

```
public static int sum(int x, int y, int z){
```

```
    return x + y + z;
```

```
}
```



Are these methods the same or different

`int myMethod(int x)` and `int myMethod(int x, int y)`

Different

`int myMethod(int x)` and `int myMethod(double x)`

Different

`int myMethod(int x)` and `int myMethod(int y)`

Same. Parameter name (identifier) does make a difference

`int myMethod(int x)` and `double myMethod(int y)`

Same. Return type does not make a difference



Passing variables as arguments

Java is always **pass-by-value**, meaning that a copy of the argument's value is passed into the method's formal parameter.



Passing immutable object references

```
void swap(int x, int y) {  
    int temp = x;  
    x = y;  
    y = temp;  
}
```

```
int a = 10;  
int b = 20;  
swap(a, b);
```

Calling the method swap with the arguments a and b, which are integer (primitive types and String objects are immutable). **Java will create a copy in memory of those values into the parameters x, y in method swap.**

What will be the values in a, b after executing the method?

a is still original a => 10
b is still original b => 20



Passing mutable object references

```
void swap2(int[] x, int[] y){  
    int[] temp = x;  
    x = y;  
    y = temp;  
}
```

```
int[] a = {1, 2, 3};  
int[] b = {4, 5, 6};
```

```
swap2(a, b)
```

Calling the method swap with the arguments a and b, which are arrays. We know we can modify an array values (mutable). **The value of a mutable object is its reference in memory. Java will create new variable that copy this reference in memory of the array. Original variables keep pointing to the same memory location.**

What will be the values in a, b after executing the method?

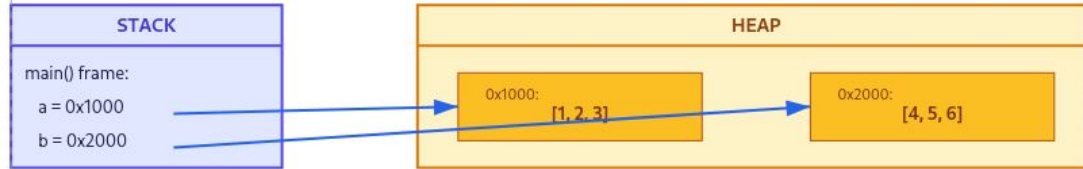
a is still pointing to => {1, 2, 3}

b is still pointing to => {4, 5, 6}

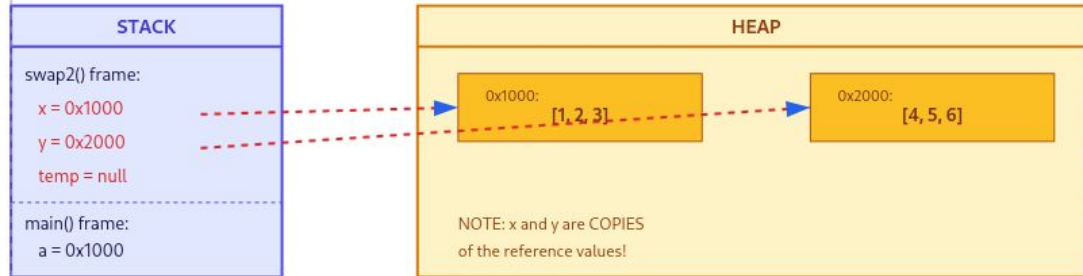
x is a copy of a's reference, and y is a copy of b's reference. Swapping x and y doesn't change where a and b point, only x and y inside the method are swapped.

Local x, y swapped ≠ main's a, b unchanged!

Step 1: Before swap2(a, b)



Step 2: Inside swap2() - Parameters created



Step 3: After "temp = x" and "x = y" and "y = temp"



Java passes references BY VALUE - you get a copy of the reference, not the reference itself!

Why swap2() Doesn't Swap Arrays in Java

Changing mutable object references

```
void change(int[] x, int[] y){  
    int[] temp = x;  
    x=y;  
    y=temp;  
    x[0] = 99;  
}
```

```
int[] a = {1, 2, 3};  
int[] b = {4, 5, 6};  
change(a, b);
```

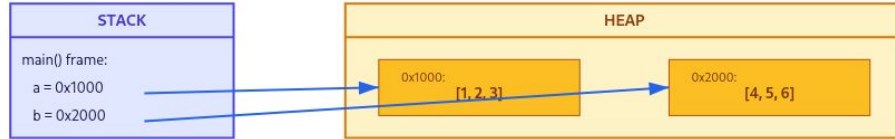
Calling the method change with the arguments a and b, which are arrays. We know we can modify an array values (mutable). **Java will create new variable that make reference to the memory location of the array. If a change is executed in one of the parameters, the array in the memory will be modified too.**

What will be the values in a, b after executing the method?

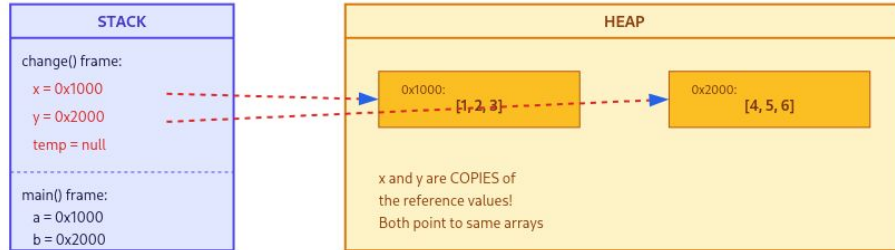
a is still pointing to => {1, 2, 3}
b is still pointing to same memory ref, but the array in this memory ref has been changed in the method => {99, 5, 6}

Next: x[0] = 99 will modify the heap!

Step 1: Before change(a, b)



Step 2: Inside change() - Parameters x and y created (copies of a and b)

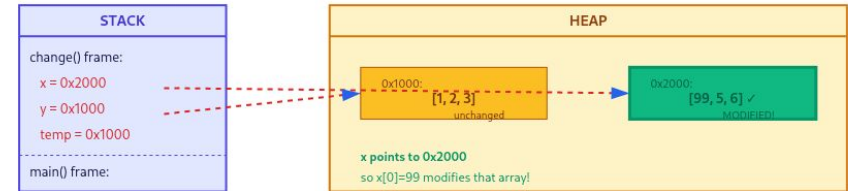


Step 3: After "temp = x" and "x = y" and "y = temp"

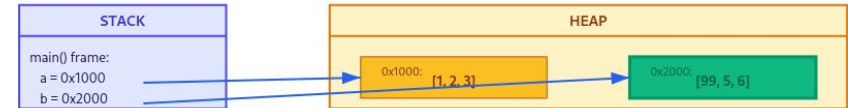


Part 2: Array Modification and Final Result

Step 4: After "x[0] = 99" executes



Step 5: After change() returns to main()



FINAL RESULT IN main():

a still references [1, 2, 3] | b now references [99, 5, 6]
The swap of x and y was local, but x[0]=99 modified the heap object that b points to!

Analogy

Imagine you have a house at "123 Main Street" and you write that address on a piece of paper.

Pass-by-Value (Java):

When you give someone directions to your house, you **write the address on a NEW piece of paper** and hand them that copy.

- **You keep your original paper** with the address
- **They get a copy** of the address
- Both papers say "123 Main Street"
- If they **cross out their paper and write a different address**, YOUR paper still says "123 Main Street"
- BUT if they **go to the house and paint it red**, the house IS red because both addresses point to the SAME house!



Passing by value summary

- Java always passes parameter variables by value.
- A mutable object's value can be changed when it is passed to a method.
- An immutable object's value cannot be changed, even if it is passed a new value.
- "Passing by value" refers to passing a copy of the value.

