Functions (II)

Parameter Passing

- **Pass by value**: the computer evaluates the argument expression and **places a copy** of its value into the memory address referenced by the parameter. (C, C++, Java)
- **Pass by reference**: the computer evaluates the argument expression and **places a reference** to the result into the memory address referenced by the parameters. (C++)

- Show Swap.java, ask the output for swap(int, int) [won't swap a and b]
- ask the output for swap(Integer, Integer) [won't swap a and b]
- ask the output for resetVal (Item item) [reset the item val]
- Why? [Java is pass by value]

```
public class Swap {
     public static void swap (int a, int b) {
          int t = a;
          a = b;
          b = t;
     }

    // If change the reference to refer to another loction,
    // any changes to the reference are not reflected
    // back in the main function
    public static void swap (Integer a, Integer b) {
        Integer t = a;
        a = b;
        b = a;
    }

    // If change the member of an object referred by the reference
    // the changes are reflected back, as even it's a copy of
    // the object's reference, it refers to the same object
    public static void resetVal (Item item) {
        item.val = 0;
    }

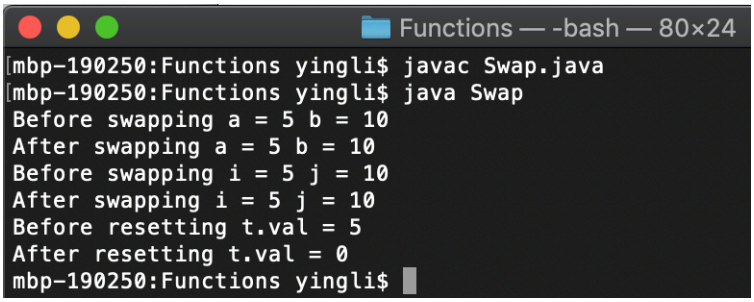
     public static void main (String[] args) {
        int a = 5;
        int b = 10;
        System.out.println("Before swapping a = " + a + " b = " + b);
        swap(a, b);
        System.out.println("After swapping a = " + a + " b = " + b);

        Integer i = 5;
        Integer j = 10;
        System.out.println("Before swapping i = " + i + " j = " + j);
        swap(i, j);
        System.out.println("After swapping i = " + i + " j = " + j);

        Item t = new Item(5);
        System.out.println("Before resetting t.val = " + t.val);
        resetVal(t);
        System.out.println("After resetting t.val = " + t.val);
     }
}
```

```
class Item {
    int val;

    public Item (int v) {
        val = v;
    }
}
```



- Now we know that Java is pass-by-value. How about C?
  - Using swap function is a good way To check whether a language uses pass-by-value or pass-by-reference. The the change happened inside the swap function cannot reflect to the calling function, the language is pass-by-value.
  - Show swapf.c, and ask the output and the parameter passing of C. [pass by value]

```
#include <stdio.h>

int swap_f (int x, int y) {
     int t;
     t = x;
     x = y;
     y = t;

     return 0;
}

int main () {
     int a = 3, b = 5;
     printf("Before swap a = %d, b = %d \n", a, b);
     swap_f(a, b);
     printf("After swap a = %d, b = %d \n", a, b);

    return 0;
}
```

- Although C uses passing by value, we have to use pointer to simulate passing by reference [show swap.c].

```c
#include <stdio.h>

int swap (int *x, int *y) {
     int t;
     t  = *x;
     *x = *y;
     *y = t;

     return 0;
}

int main () {
     int a = 3, b = 5;
     printf("Before swap a = %d, b = %d \n", a, b);
     swap(&a, &b);
     printf("After swap a = %d, b = %d \n", a, b);

    return 0;
}
```

- However, passing pointers as values is not equivalent to pass-by-reference. We can tell the difference after knowing what is pass-by-reference in C++.

- C++ allows passing by reference. Show swap.cc.

```cpp
#include <iostream>
using namespace std;

int swap (int &a, int &b) {
     int tmp = a;
     a = b;
     b = tmp;

     return 0;
}

int main () {
     int a = 3, b = 5;
     cout << "Before swap a = " << a << ", b = " << b << "\n";
     swap(a, b);
     cout << "After swap a = " << a << ", b = " << b << "\n";

    return 0;
}
```

- The above swap function passes arguments as references. Using pass-by-reference, *the only thing the swap function can do is update the values in the memory referred by the references*. It's *syntactically incorrect to update the references* inside the swap function. This is to say it's invalid to do anything like &a = &b in the above swap function.

- Note: passing by reference is not equivalent to passing pointer by value.
  - If a function is pass-by-reference, it's unlikely to modify the reference itself in the function be the value referred by the reference.

- If a function is pass-by-value and the value is a pointer, it's possible to reassign another address to the pointer, and the changes on the value referred by the new address won't impact the value stored in the original address.

- Show ppbv.cc and ask the output.

```
#include <iostream>
using namespace std;

// difference between pass by reference and
// passing a pointer by value.
void f( int &a ) {
    a = 3;
}

void g( int *a ) {
    *a = 4;
    int *y = (int*)malloc( sizeof(int) );
    *y = 5;
    a = y;
    *a = 7;
}

int main () {
  int a = 1;
  cout << "Before f, a = " << a << endl;
  f( a );
  cout << "After f, a = " << a << endl;
  cout << "Before g, a = " << a << endl;
  g( &a );
  cout << "After g, a = " << a << endl;

  return 0;
}
```

```
Before f, a = 1
After f, a = 3
Before g, a = 3
After g, a = 4
```

- Another thing you may want to pay attention to pass-by-reference and pass-by-value and the value is a pointer is the aliasing, which means reference to the same entity by different names.

```
#include <iostream>
using namespace std;

int open_to_problems (int &a, int &b) {
     a = a + 1;
     b = b + 1;

     return 0;
}

int main () {
  int x = 10, y = 20;

  cout << "Before open_to_problems, x = " << x << ", y = " << y << endl;
  open_to_problems( x, y );
  cout << "After open_to_problems, x = " << x << ", y = " << y << endl;

  cout << "Before open_to_problems, x = " << x << endl;
  open_to_problems( x, x );
  cout << "After open_to_problems, x = " << x << endl;

  return 0;
}
```

```
Before open_to_problems, x = 10, y = 20
After open_to_problems, x = 11, y = 21
Before open_to_problems, x = 11
After open_to_problems, x = 13
```