Review on Methods
Method definition

public static int plus(int x, int y) {
    // Method body
}

return type

parameters
/* Variable x declared in the parameter list of the div method is totally
* independent of the variable x declared inside main.
*/

class Math {
    public static void main(String args[]) {
        int x = 2;
        int y = 4;

        System.out.println(div(y, x)); // answer is 2, not 0.5
    }
}

public static float div(int x, int y) {
    return x / y;
}
Scope of variables

class Math {
    public static void main(String args[]) {
        int num;
        initialize();

        /* The num here is still uninitialized! The num here is totally
         * independent from the num inside initialize(). They just
         * happen to have the same names.
        */
    }
}

public static void initialize() {
    int num = 0;
}
}
Return

• Assigns a value to the method. Ex.

    static float pi() {
        return 3.1415;
    }

    // somewhere
    float temp = pi();

• Can be any arbitrary expression as long as the resulting value will have the same type as the return type.
Return

- Also jumps out of the method and returns the control to the caller of the function

```c
static float pi() {
    return 3.1415;

    // The lines below can never be executed! The compiler will also complain about it.
    int x = 0;
    return x++;
}
```
static public int plus (int x, int y) {
    return x + y;
}

static public void main(String[] args) {
    int sum = plus(1, 2);
    // sum is now 3
}
Visualizing function calls

```java
static public int sum(int x, int y) {
    int plus = (int) x + y;
    return plus;
}
```
Void

- Special return type for methods that does not need to have a resulting value. Ex.

```java
static void printPI() {
    System.out.println(3.1415);
}
```

// somewhere
printPI();

- Return statement is implied at the end of the method body.
Void

- Methods with a void return type cannot be assigned to any expression. Ex.

  float pi = printPI(); // illegal!

  if (printPI()) { // illegal!
  ..
  }
Pass by Value

- This applies to all primitive Java datatypes - boolean, byte, char, short, int, long, float and double.

- The actual value is being passed to the method. Modifying the value inside the method won't affect the value outside. Ex.

```java
public static int increment(int x) {
    return x++;
}
```

// somewhere
int a = 0;
int b = increment(a); // b is 1, a is still 0
Pass by Reference

- This applies to all non-primitive types, like arrays and objects.
- The reference of the object is being passed to the method, and modifying the object inside the method will affect the object outside. Ex.

```java
static void setFirstElemToOne(int arr[]) {
    arr[0] = 1;
}

// somewhere
int zeroes[] = {0, 0};
setFirstElemToOne(zeroes);
// zeroes is now {1, 0}
```
Pass by Reference

• However, reassigning the parameter and modifying it will not affect the outside object since the variable is now referring to a different object. Ex.

```java
static void setFirstElemToOne(int arr[]) {
    arr = new int[2];
    arr[0] = 1;
}

// somewhere
int fives[] = {5, 5};
setFirstElemToOne(fives);
// fives is still {5, 5}
```