

Flow of execution

Flow of execution

The *flow of execution* is the order in which program statements are executed.

Unless specified otherwise, program execution always begins from the first line. Statements are executed one at a time, top down.

Flow of execution

The *flow of execution* is the order in which program statements are executed.

Most programming languages support three *control structures* to control the flow of execution.

- Decision structures
- Repetition structures
- Functions/methods

Methods

Methods

A *method* is a subprogram—a small program inside a program—with a name.

- Can be used anywhere anytime by calling its name.
- Most useful when we want to solve the same basic problem multiple times (e.g., rounding a real number to an integer).

Why methods?

- Make easier to read and debug.
- Make programs shorter by eliminating redundancy; if you make a change, you only have to make it in one place.
- Can be used for other purposes in the future...

Encapsulation

- Wrapping a piece of code up in a method is called *encapsulation*.
- To *encapsulate* means to enclose something in a capsule.

Interface

- When a piece of code is encapsulated in a method, users do not need to know how it is implemented inside.
- Users only need to know the *interface*: the name, list of parameters and what is being computed (but not how).
- For example, to invoke `Math.sqrt(2)`, users do not need to know how to compute square roots.

The flow of execution

The *flow of execution* is the order in which program statements are executed.

Unless specified otherwise, statements are executed one at a time from the first line, top down.

Method definitions create method objects only, and they are not executed until called.

The flow of execution

A method call creates a detour in the flow of execution.

- When a method is called, instead of going to the next line, the flow jumps to the first line of the method.
- When it reaches the last line of the method, the flow returns to where it left off.

The flow of execution

```
public static void happy() {  
    System.out.println("Happy birthday to you!");  
}  
  
public static void sing(String anyone) {  
    happy();  
    happy();  
    System.out.println("Happy birthday, dear " + anyone  
        + ".");  
    happy();  
}  
  
public static void main(String[] args) {  
    sing("Dan");  
    System.out.println();  
    sing("Ann");  
}
```

The flow of execution

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear Dan.

Happy birthday to you!

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear Ann.

Happy birthday to you!

Parameters and arguments

A variable inside a method whose value is specified by a caller is called a *parameter*.

A value being passed by a caller to a parameter is called an *argument*.

A parameter is always a *variable*.

An argument is always a *value*.

In Java, an argument's *type* must match with the corresponding parameter's type.

Parameters and arguments

Consider this very simple method.

```
public static void moo(int n) {  
    System.out.println(n);  
    System.out.println(n);  
}
```

What does this do? This method simply prints the value of `n` twice.

Parameters and arguments

What does this do?

```
moo(1823);
```

Output:

```
1823
```

```
1823
```

Parameters and arguments

What does this do?

```
int m = 1823;
```

```
moo(m + 3);
```

Output:

1826

1826

Parameters and arguments

If a method has n parameters, a caller must pass n arguments.

Arguments are assigned to parameters according to the order alone.

Parameters and arguments

Consider this very simple method.

```
public static void noo(int m, int n) {  
    System.out.println("1st value: " + m);  
    System.out.println("2nd value: " + n);  
}
```

What does this do?

This method simply prints the values of the parameters `m` and `n`.

Parameters and arguments

What does this do?

```
noo (3, 4) ;
```

Output:

```
1st value: 3
```

```
2nd value: 4
```

Parameters and arguments

What does this do?

```
int m = 3;  
int n = 4;  
noo(m, n);
```

Output:

```
1st value: 3  
2nd value: 4
```

Parameters and arguments

What does this do?

```
int m = 3;  
int n = 4;  
noo(n, m);
```

Output:

```
1st value: 4  
2nd value: 3
```

Variables and parameters

Variables and parameters defined inside a method are *local*: they can only be used inside the method.

Value-returning methods

Value-returning methods

So far all methods we designed output the results of computation via print statements:

```
public static void moo(<parameters>) {  
    <statements>  
    System.out.println(...);  
}
```

However, methods can also *return* (and thus represent) values.

Value-returning methods

This method `maxOfTwo` *returns* the maximum of two given integers:

```
public static int maxOfTwo(int m, int n) {  
    if (m > n) return m;  
    else return n;  
}
```

Here, the maximum value will be *returned* to the caller, where `int` specifies its type.

Value-returning methods

When we call a method that returns a value, a method call is an *expression* representing a *value* being returned.

Example. `maxOfTwo(3, 4)` is an **int** expression representing an **int** value returned by the method `maxOfTwo()` with given arguments 3 and 4 (that is, 4).

Value-returning methods

What would this output?

```
public static int maxOfTwo (int m, int n) {  
    if (m > n) return m;  
    else return n;  
}
```

```
System.out.println(maxOfTwo(3, 4));
```

Output: 4

Value-returning methods

What would this output?

```
public static int maxOfTwo(int m, int n) {  
    if (m > n) return m;  
    else return n;  
}
```

```
System.out.println(maxOfTwo(3, 4)  
    + maxOfTwo(5, 6));
```

Output: 10

Value-returning methods

What would this output?

```
public static int maxOfTwo(int m, int n) {  
    if (m > n) return m;  
    else return n;  
}
```

```
System.out.println(maxOfTwo(3, (maxOfTwo(4, 5))));
```

Output: 5

Value-returning methods

This method `isEven` returns a boolean value: **true** or **false**.

```
public static boolean isEven(int n) {  
    if (n % 2 == 0) return true;  
    else return false;  
}
```

Here, **boolean** indicates the type of a value returned.

Value-returning methods

This method `isEven` returns a boolean value: **true** or **false**.

```
public static boolean isEven(int n) {  
    return (n % 2 == 0);  
}
```

Here, **boolean** indicates the type of a value returned.

Value-returning methods

What would this output?

```
public static boolean isEven(int n) {  
    return (n % 2 == 0);  
}
```

```
System.out.print(isEven(1823));
```

Output: **false**

Value-returning methods

What would this output?

```
public static int maxOfTwo(int m, int n) {  
    if (m > n) return m;  
    else return n;  
}
```

```
public static boolean isEven(int n) {  
    return (n % 2 == 0);  
}
```

```
System.out.print(isEven(maxOfTwo(3, 4)));
```

Output: **true**

Value-returning methods

What would this output?

```
public static boolean isEven(int n) {  
    return (n % 2 == 0);  
}
```

```
Scanner s = new Scanner(System.in);  
System.out.print("Enter a number: ");  
int x = s.nextInt();  
if (isEven(x))  
    System.out.println("You entered an even number.");  
else  
    System.out.println("You entered an odd number.");
```

Incremental development

Example. A method to return the average of three numbers.

Step 1. Start with correct syntax.

```
public static double average(int l, int m, int n) {  
    return 0.0;  
}
```

```
System.out.println(average(1, 2, 4)); // prints 0
```

Incremental development

Step 2. Compute the sum and print it.

```
public static double average(int l, int m, int n) {  
    int sum = l + m + n;  
    System.out.println("sum is: " + sum);  
    return 0.0;  
}
```

```
System.out.println(average(1, 2, 4)); // is sum correct?
```

Incremental development

Step 3. Compute the average and print it.

```
public static double average(int l, int m, int n) {  
    int sum = l + m + n;  
    System.out.println("sum is: " + sum);  
    double ave = sum/2.0;  
    System.out.println("average is: " + ave);  
    return 0.0;  
}
```

```
System.out.println(average(1, 2, 4)); // is it correct?
```

Incremental development

Step 4. Fix a bug.

```
public static double average(int l, int m, int n) {  
    int sum = l + m + n;  
    System.out.println("sum is: " + sum);  
    double ave = sum/3.0;  
    System.out.println("average is: " + ave);  
    return 0.0;  
}
```

```
System.out.println(average(1, 2, 4)); // it is correct!
```

Incremental development

Step 5. Return the average.

```
public static double average(int l, int m, int n) {  
    int sum = l + m + n;  
    // System.out.println("sum is: " + sum);  
    double ave = sum/3.0;  
    // System.out.println("average is: " + ave);  
    return ave;  
}
```

```
System.out.println(average(1, 2, 4)); // it is correct!
```

Incremental development

Step 6. Clean up and document.

```
/**  
 * Returns the average of three given integers.  
 *  
 * @param l - an integer  
 * @param m - an integer  
 * @param n - an integer  
 * @return the average of l, m and n.  
 */  
public static double average(int l, int m, int n) {  
    int sum = l + m + n;  
    double ave = sum/3.0;  
    return ave;  
}
```

Incremental development

- Add code in small increments.
- Run your partial solution.
- Test and verify before adding more code.
- Use local variables to print partial results.
- Remove scaffolding and clean up code.
- Document your code.