

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...

Methods

Example. Suppose you want to write a program to print out these famous lyrics:

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear <someone>.

Happy birthday to you!

Methods

If you know who <someone> is, say, Dan, you would just write:

```
System.out.println("Happy birthday to you!");  
System.out.println("Happy birthday to you!");  
System.out.println("Happy birthday, dear Dan.");  
System.out.println("Happy birthday to you!");
```

But this code has a lot of redundancy!

Methods

To save this redundancy, we can define a method for printing "Happy birthday to you!" in the following way.

```
public static void happy() {  
    System.out.println("Happy birthday to you!");  
}
```

This method is named `happy()`.

Methods

To *call* or *invoke* this method `happy()`, all you need to do is:

```
happy();
```

Methods

The lyrics for Dan can be simplified as:

```
happy();  
happy();  
System.out.println("Happy birthday, dear Dan.");  
happy();
```

which will print

Methods

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear Dan.

Happy birthday to you!

Methods

We can also make this a method as well:

```
public static void singDan() {  
    happy();  
    happy();  
    System.out.println("Happy birthday, dear Dan.");  
    happy();  
}
```

Methods

We can also make this a method for Ann:

```
public static void singAnn() {  
    happy();  
    happy();  
    System.out.println("Happy birthday, dear Ann.");  
    happy();  
}
```

Methods

What does this do?

```
singDan();  
System.out.println();  
singAnn();
```

Methods

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!

Methods

I still see some redundancy here. Can we make a generic method to sing happy birthday to anyone and use that for Dan and Ann somehow?

Yes, using a *parameter*—a variable whose value is specified by the caller (or user).

Methods

Consider this method for anyone:

```
public static void sing(String anyone) {  
    happy();  
    happy();  
    System.out.println("Happy birthday, dear "  
        + anyone + ".");  
    happy();  
}
```

Here, the value of `anyone` is to be specified by the caller of this method.

Methods

Invoking `sing("Dan")` gives:

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear Dan.

Happy birthday to you!

Here, "Dan" is passed as an argument to `sing's` parameter `anyone`.

Methods

Invoking `sing("Ann")` gives:

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear Ann.

Happy birthday to you!

Here, `"Ann"` is passed as an argument to `sing's` parameter `anyone`.

Methods

Now, what does this do?

```
sing("Dan");  
System.out.println();  
sing("Ann");
```

Methods

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!

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.