Advanced Computer Programming

[Lecture 05]

Saeed Reza Kheradpisheh

kheradpisheh@ut.ac.ir

Department of Computer Science
Shahid Beheshti University
Spring 1397-98

METHODS

A method **packages a computation** consisting of multiple steps into a form that can be easily understood and **reused**.

Methods as Black Boxes

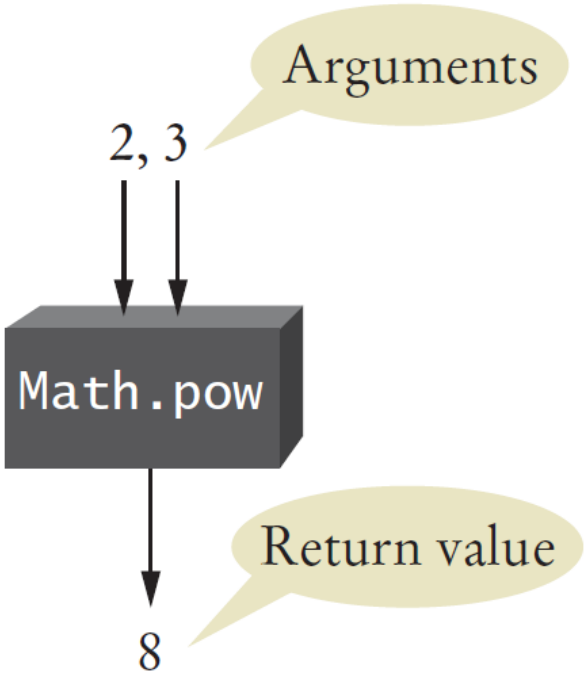
Definition

A **method** is a sequence of instructions with a name.

Methods you've seen

- `Math.pow`
  Contains instructions to compute a power x^y.
- `main`
  Contains instructions which the program starts and continues with executing them.

Methods as Black Boxes



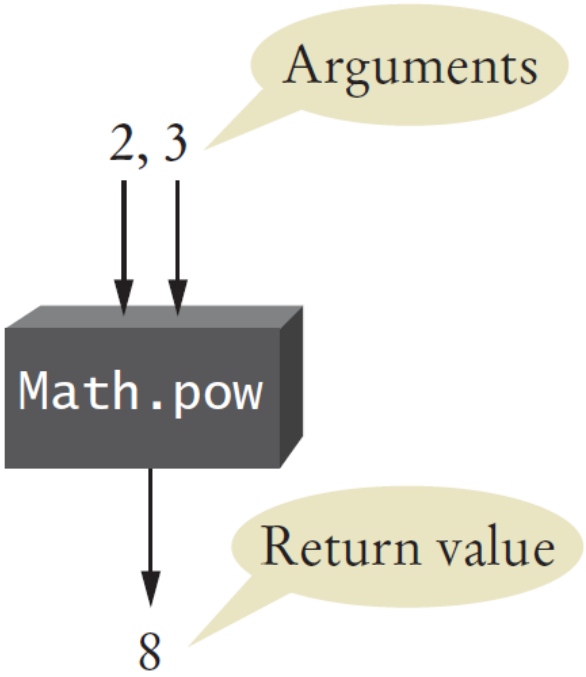
Definition

Arguments are inputs to a method, and supplied when the method is called.

Definition

The **return value** is the result that the method computes. The return value of a method is returned to the calling method.

Methods as Black Boxes



Definition

Arguments are inputs to a method, and supplied when the method is called.

Definition

The **return value** is the result that the method computes. The return value of a method is returned to the calling method.

Methods can receive multiple arguments (or zero), but they return at most one value.

Calling a Method

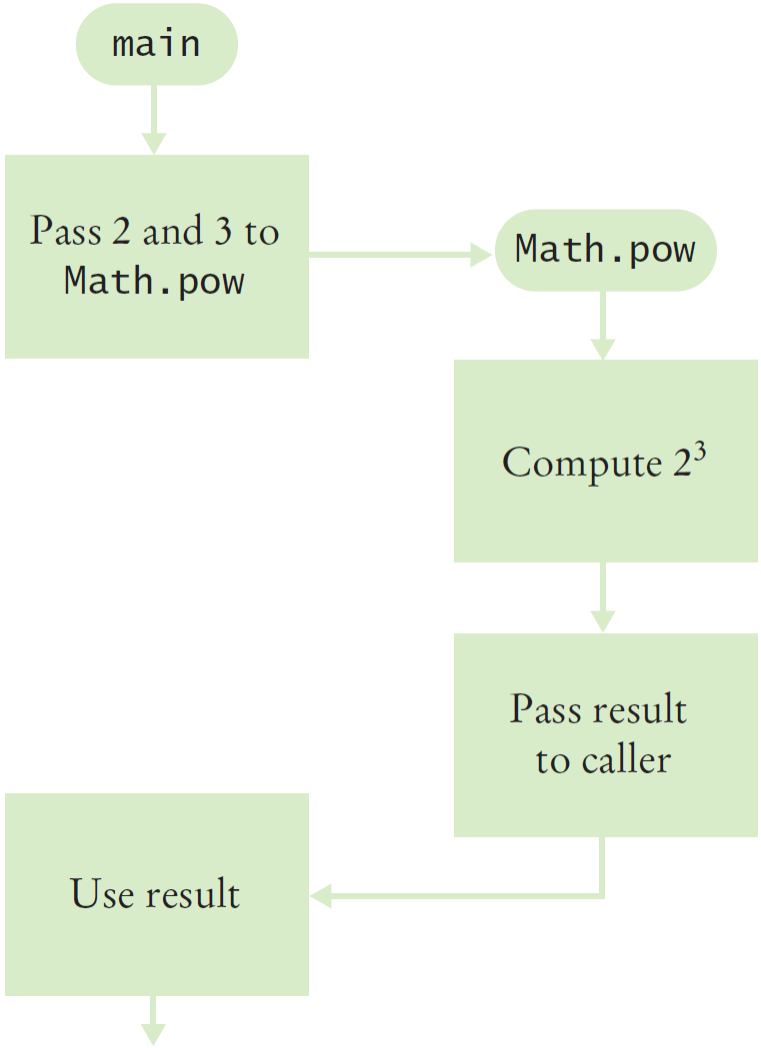
You call a method in order to execute its instructions.

```
public static void main(String[] args)
{
   double result = Math.pow(2, 3);
   . . .
}
```

Calling a Method

You call a method in order to execute its instructions.

```
public static void main(String[] args)
{
   double result = Math.pow(2, 3);
   . . .
}
```



Implementing a Method

Definition

When **declaring [header of] a method**, you provide a name for the method, a variable for each argument, and a type for the result.

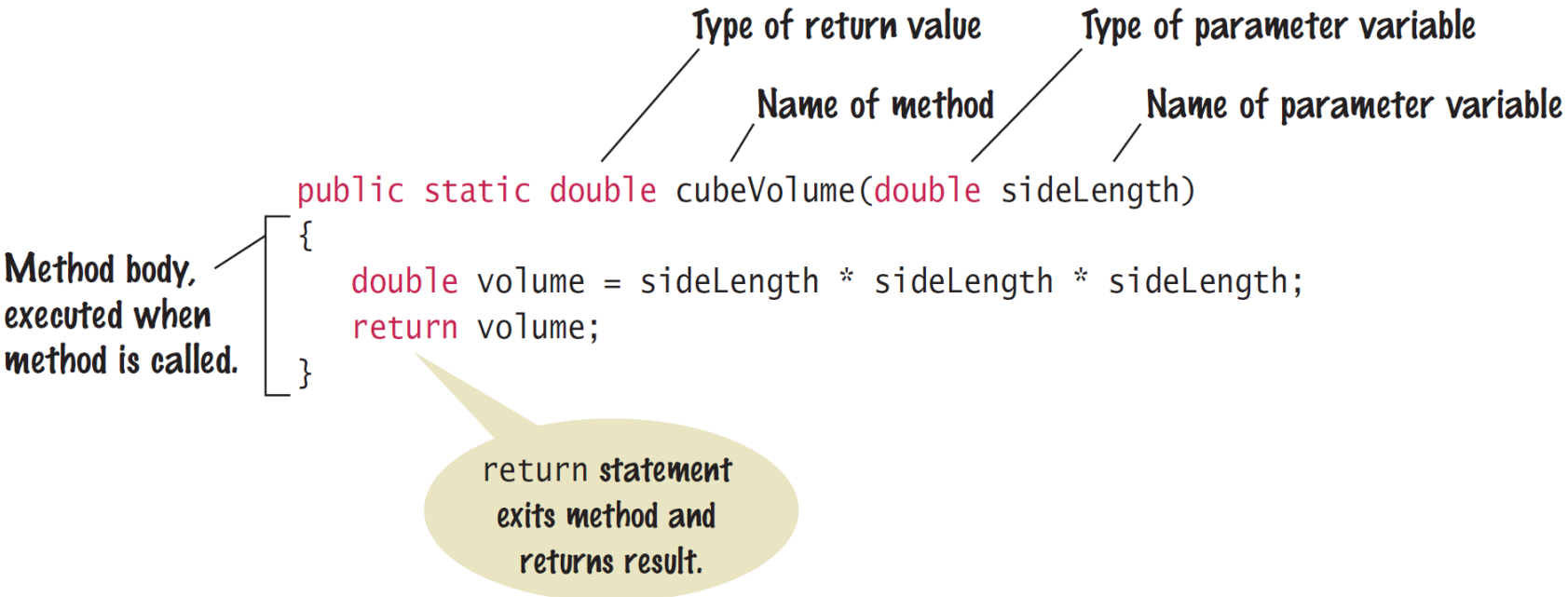
Implementing a Method

Definition

When **declaring [header of] a method**, you provide a name for the method, a variable for each argument, and a type for the result.

Syntax

```
public static returnType methodName(parameterType parameterName, . . . )
{
   method body
}
```



Example

```
1  /**
2     This program computes the volumes of two cubes.
3  */
4  public class Cubes
5  {
6     public static void main(String[] args)
7     {
8        double result1 = cubeVolume(2);
9        double result2 = cubeVolume(10);
10       System.out.println("A cube with side length 2 has volume " + result1);
11       System.out.println("A cube with side length 10 has volume " + result2);
12    }
13
14    /**
15       Computes the volume of a cube.
16       @param sideLength the side length of the cube
17       @return the volume
18    */
19    public static double cubeVolume(double sideLength)
20    {
21       double volume = sideLength * sideLength * sideLength;
22       return volume;
23    }
24 }
```

Parameters and Arguments

Parameters v.s. Arguments

Parameters are variables that hold the arguments supplied in the method call.

The values that are supplied to the method when it is called are the **arguments** of the call.

Each parameter variable is initialized with the corresponding argument.

Parameters and Arguments

1 Method call

```
double result1 = cubeVolume(2);
```

result1 =

sideLength =

2 Initializing method parameter variable

```
double result1 = cubeVolume(2);
```

result1 =

sideLength = 2

3 About to return to the caller

```
double volume = sideLength * sideLength * sideLength;
return volume;
```

result1 =

sideLength = 2

volume = 8

4 After method call

```
double result1 = cubeVolume(2);
```

result1 = 8

Tips and Errors

- Do Not Modify Parameter Variables.

```
public static int totalCents(int dollars, int cents)
{
   cents = dollars * 100 + cents; // Modifies parameter variable
   return cents;
}
```

Tips and Errors

- Do Not Modify Parameter Variables.

```
public static int totalCents(int dollars, int cents)
{
   cents = dollars * 100 + cents; // Modifies parameter variable
   return cents;
}
```

- You Cannot Modify Arguments.

```
public static int addTax(double price, double rate)
{
   double tax = price * rate / 100;
   price = price + tax; // Has no effect outside the method
   return tax;
}
```

Now consider

```
double total = 10;
addTax(total, 7.5); // Does not modify total
```

Return Values

Usage

The **`return` statement** terminates a method call and yields the method result.

When the return statement is processed, the method exits immediately.

```
public static double cubeVolume(double sideLength)
{
   if (sideLength < 0) { return 0; }
   // Handle the regular case
   . . .
}
```

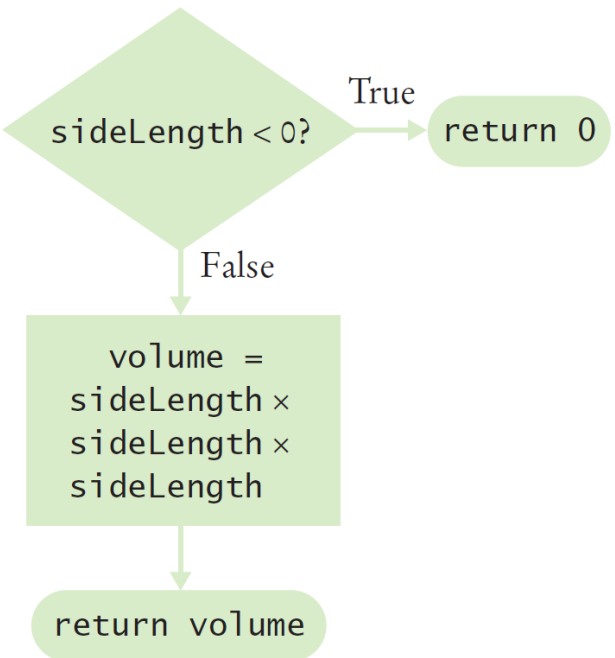
Return Values

Usage

The **`return` statement** terminates a method call and yields the method result.

When the return statement is processed, the method exits immediately.

```
public static double cubeVolume(double sideLength)
{
   if (sideLength < 0) { return 0; }
   // Handle the regular case
   . . .
}
```



Error

Every branch of a method needs to return a value.

```
public static double cubeVolume(double sideLength)
{
   if (sideLength >= 0)
   {
      return sideLength * sideLength * sideLength;
   } // Error—no return value if sideLength < 0
}
```

Error

Every branch of a method needs to return a value.

```
public static double cubeVolume(double sideLength)
{
   if (sideLength >= 0)
   {
      return sideLength * sideLength * sideLength;
   } // Error—no return value if sideLength < 0
}
```

```
public static double cubeVolume(double sideLength)
{
   if (sideLength >= 0)
   {
      return sideLength * sideLength * sideLength;
   }
   else
   {
      return 0;
   }
}
```

Implementing a Method: Guidelines

1. Describe what the method should do.
2. Determine the method's "inputs".
3. Determine the types of the parameter variables and the return value.
4. Write pseudocode for obtaining the desired result.
5. Implement the method body.
6. Test your method.

Exercise (`Password.java`)

Create a method that generates random passwords of a given length containing lower case letters.

Exercise (`Password.java`)

Create a method that generates random passwords of a given length containing lower case letters.

Exercise (`Password.java`)

Improve previous passwords by generating passwords with at least one digit and one special character.

Methods Without Return Values

Usage

Use a return type of **`void`** to indicate that a method does not return a value.

The `return` statement still works here! It just terminates the method.

A typical example: print a string in a box, like this:

```
-------
!Hello!
-------
```

A method for this task can be declared as follows:

```
public static void boxString(String contents)
```

Methods Without Return Values

Usage

Use a return type of **`void`** to indicate that a method does not return a value.

The `return` statement still works here! It just terminates the method.

A typical example: print a string in a box, like this:

```
-------
!Hello!
-------
```

A method for this task can be declared as follows:

```
public static void boxString(String contents)
```

Exercise (`BoxPrint.java`)

Implement the method `boxString`.

Problem Solving

- Reusable Methods
  - When you write nearly identical code or pseudocode multiple times, either in the same program or in separate programs, consider introducing a method.
  - Design your methods to be reusable. Supply parameter variables for the values that can vary when the method is reused.
- Divide and Conquer
  - Decompose a difficult task into some simpler sub-tasks.

Problem Solving

Exercise (`FiboBin.java`)

We call an integer number n a FiboBinary number if there is an integer i such that:
$n = fib(i) + bin(fib(i))$,
where $fib(i)$ is the i'th Fibonacci number and $bin(x)$ is the number of ones in the binary representation of x.
Implement a method that checks whether a number is FiboBinary or not.

Variable Scope

Definition

The **scope** of a variable is the part of the program in which you can access it.

Variable Scope

Definition

The **scope** of a variable is the part of the program in which you can access it.

Examples: body blocks of the followings define a scope

- Method
- Decisions
  - `if`
  - `else if`
  - `else`
  - `switch`
- Loops
  - `for`
  - `while`
  - `do while`

Variable Scope

Definition

The **scope** of a variable is the part of the program in which you can access it.

Examples: body blocks of the followings define a scope

- Method
- Decisions
  - `if`
  - `else if`
  - `else`
  - `switch`
- Loops
  - `for`
  - `while`
  - `do while`

Two variables can have the same name, provided that their scopes do not overlap.

Recursive Methods

Definition

A **recursive method** is a method that calls itself.

A recursive computation solves a problem by using the solution of the same problem with simpler inputs.

Recursive Methods

Definition

A **recursive method** is a method that calls itself.

A recursive computation solves a problem by using the solution of the same problem with simpler inputs.

A typical example: print triangle patterns like this:

```
[]
[][]
[][][]
[][][][]
```

Recursive Methods

```
public static void printTriangle(int sideLength)
{
   printTriangle(sideLength - 1);
   for (int i = 0; i < sideLength; i++)
   {
      System.out.print("[]");
   }
   System.out.println();
}
```

Recursive Methods

```
public static void printTriangle(int sideLength)
{
   printTriangle(sideLength - 1);
   for (int i = 0; i < sideLength; i++)
   {
      System.out.print("[]");
   }
   System.out.println();
}
```

For a recursion to terminate, there must be special cases for the simplest inputs.

```
public static void printTriangle(int sideLength)
{
   if (sideLength < 1) { return; }
   printTriangle(sideLength - 1);
   for (int i = 0; i < sideLength; i++)
   {
      System.out.print("[]");
   }
   System.out.println();
}
```

Recursive Methods

Here is what happens when we print a triangle with side length 4:

- The call `printTriangle(4)` calls `printTriangle(3)`.
  - The call `printTriangle(3)` calls `printTriangle(2)`.
    - The call `printTriangle(2)` calls `printTriangle(1)`.
      - The call `printTriangle(1)` calls `printTriangle(0)`.
        - The call `printTriangle(0)` returns, doing nothing.
      - The call `printTriangle(1)` prints `[]`.
    - The call `printTriangle(2)` prints `[][]`.
  - The call `printTriangle(3)` prints `[][][]`.
- The call `printTriangle(4)` prints `[][][][]`.

Exercise (`Brackets.java`)

Create a recursive method that takes n and prints the following pattern which has the length of $2n$.

```
[[[[[]]]]]
```