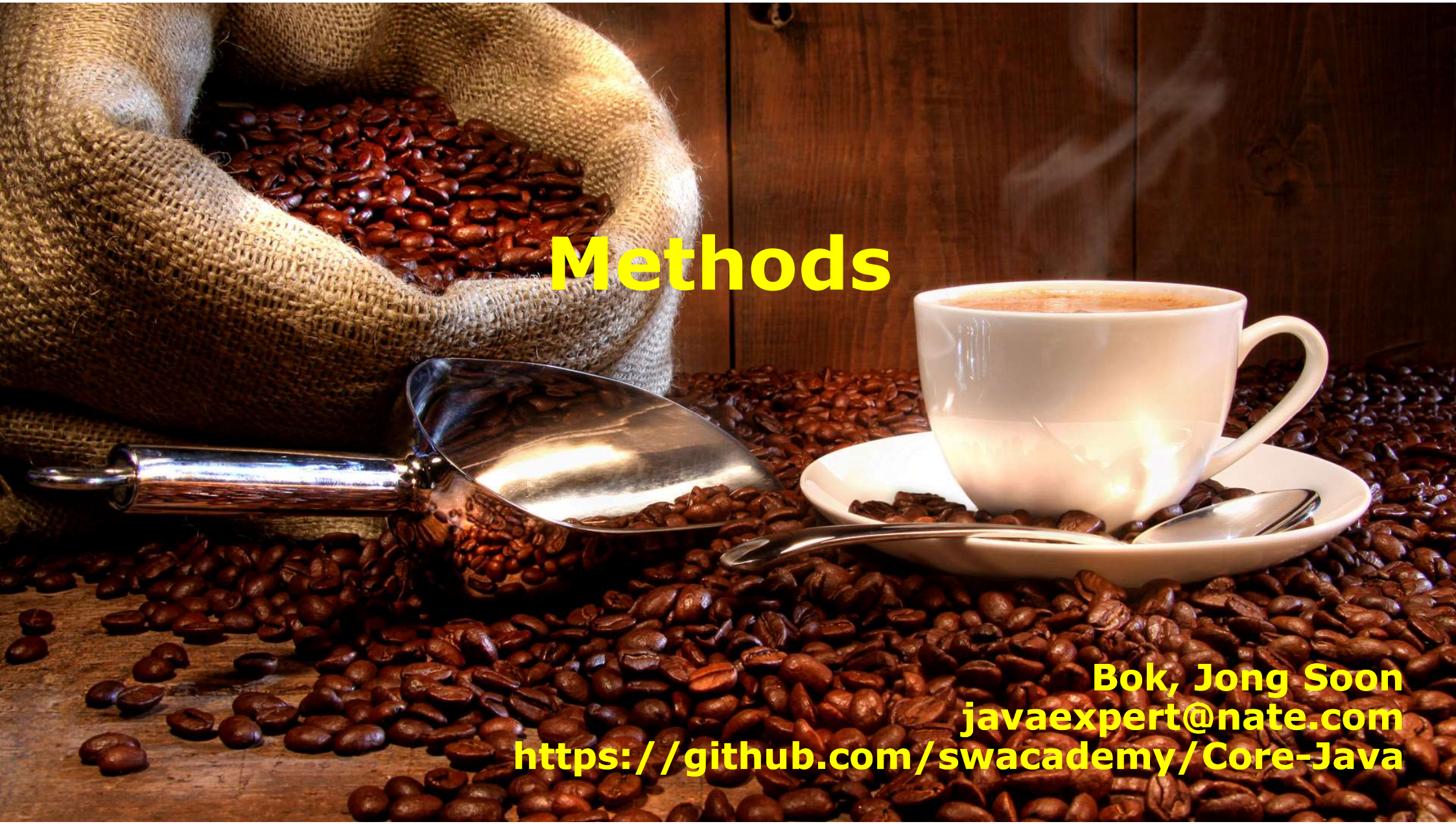
Methods
Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Core-Java

Methods

- Enable you to separate statements into code blocks.
- Can be called whenever appropriate.
- Can “invoke” each other.
- Can call themselves(recursion)

All programs use methods. Applications begin with the method called **`main`**.

Why Are Methods Necessary?

- If you do not use methods you are not using object orientation.
- Methods make programs more readable and more maintainable.
- Methods make development and maintenance quicker.
- Methods are central to reusable software.
- Methods avoid duplication.

Methods Syntax

```
[modifiers] return_type method_name ([parameters])
  { method_body }
```

- **`modifiers`** - Java keywords that can be used to modify the way methods are stored or how they run.
- **`return_type`** - Methods used to calculate a value or query an object for a variable.
- **`method_name`** – Identifier that will be used to call the method.
- **`parameters`** - Sequences of statements that perform a task.
- **`method_body`** - Methods that perform a task.

Calling Methods

- After you define a method, you can:
  - Call a method from within the same class
    - Use method's name followed by a parameter list in parentheses
  - Call a method that is in a different class
    - You must indicate to the compiler which class contains the method to call
  - Use nested calls
    - Methods can call methods, which can call other methods, and so on
- Calling method resumes at point after it called the worker method

Using the `return` Statement

- Immediate return
- Return with a conditional statement

```
static void exampleMethod( ){
    int su;
    //...
    System.out.println("Hello");
    if (su < 10)
        return;
    System.out.println("World");
}
```

Returning Values

- Declare the method with non-void type
- Add a return statement with an expression
  - Sets the return value
  - Returns to caller
- Non-void methods must return a value

```
static int twoPlusTwo( ) {
    int a,b;
    a = 2;
    b = 2;
    return a + b;
}
```

```
int x;
x = twoPlusTwo( );
System.out.println(x);
```

Receiving Return Values

- The return value is returned to the same place in your code where you called it from
- You can combine the call and the use of the return value in one line, when using the return value.

```
int value = t.add(2, 4);
```

Declaring and Calling Parameters

- Declaring parameters
  - Place between parentheses after method name
  - Define type and name for each parameter
- Calling methods with parameters
  - Supply a value for each parameter

```
static void methodWithParameters(int n, String y)
{ ... }

methodWithParameters(2, "Hello, world");
```

Call by Value (Pass by Value)

- Default mechanism for passing parameters:
  - Parameter value is copied
  - Variable can be changed inside the method
  - Has no effect on value outside the method
  - Parameter must be of the same type or compatible type

```
static void addOne(int x){
      x++; // Increment x
}
public static void main(String [] args){
      int k = 6;
      addOne(k);
      System.out.println(k);
       // Display the value 6, not 7
}
```

Call by Reference in Java

- What are reference parameters?
  - A reference to memory location
- Using reference parameters
  - Match types and variable values
  - Changes made in the method affect the caller
  - Assign parameter value before calling the method

```
static void addOne(Test test){
       test.su++; // Increment test.su
}
public static void main(String [] args){
       Test t = new Test();
       addOne(t);
       System.out.println(t.su); // Display the value 6
}

class Test{
       int su = 5;
}
```

Varargs (Variable Arguments)

- This facility eliminates the need for manually boxing up argument lists into an array when invoking methods that accept variable-length argument lists.

```
2  public class VarargsDemo {
3      public static void main(String[] args) {
4          // TODO Auto-generated method stub
5          new VarargsDemo().test(5,6,7,8,9);
6      }
7      void test(int ... array){
8          for(int i=0; i < array.length; i++){
9              System.out.printf("array[%d] = %d\n", i, array[i]);
10         }
11     }
12 }
```

```
array[0] = 5
array[1] = 6
array[2] = 7
array[3] = 8
array[4] = 9
```

Using Recursive Methods

- A method can call itself
  - Directly
  - Indirectly
- Useful for solving certain problems

```
void printTest(int n){
        System.out.print(n + "\t");
        if(n == 3) return;
        else   printTest(n + 1);
}
```

Using Recursive Methods (Cont.)

```
void display(int n){      //1부터 3까지 출력하기
    System.out.printf("%d\t", n);
    if(n == 3 ) return;
    else display(n+1);
}
```

```
int fibonacci(int n){    //피보나치수열 계산하기
    if( n <= 2 ) return 1;
    else return fibonacci(n - 2) + fibonacci( n - 1);
}
```

```
int factoria(int n) {   //factoria 값 구하기
    if( n == 0) return 1;
    else return n * factoria( n-1 );
}
```

```
int hap (int n) {   // 1부터 n까지의 합 구하기
    int dab;
    if ( n == 1 ) return 1;
    else dab = n + hap(n-1);
    return dab;
}
```

Method Overloading

- Methods that share a name in a class
  - Distinguished by examining parameter lists

```
class OverloadingExample {
    static int add(int a, int b){
        return a + b;
    }
    static int add(int a, int b, int c) {
        return a + b + c;
    }
    public static void main(String [] args){
        System.out.println(add(1,2) + add(1,2,3));
    }
}
```

Method Signatures

- Method signatures must be unique within a class
- Signature definition

Forms Signature Definition	No Effect on Signature
■ Name of method ■ Parameter type	■ Name of parameter ■ Return type of method

`static` Methods

- Must be preceded with the name of their class and the dot operator("`.`")
- When to use `static`.
- Example of a method: `Math.random()`.
- Why `main()` is `static`.