METHODS

Methods

- *Methods can be used to define reusable code and organize and simplify coding*
- A *method* is a collection of statements grouped together to perform an operation

-
  ```
  int sum = 0;
   for (int i = 1; i <= 10; i++)
   sum += i;
   System.out.println("Sum from 1 to 10 is " + sum);
   sum = 0;
   for (int i = 20; i <= 37; i++)
   sum += i;
   System.out.println("Sum from 20 to 37 is " + sum);
   sum = 0;
   for (int i = 35; i <= 49; i++)
   sum += i;
   System.out.println("Sum from 35 to 49 is " + sum);
  ```

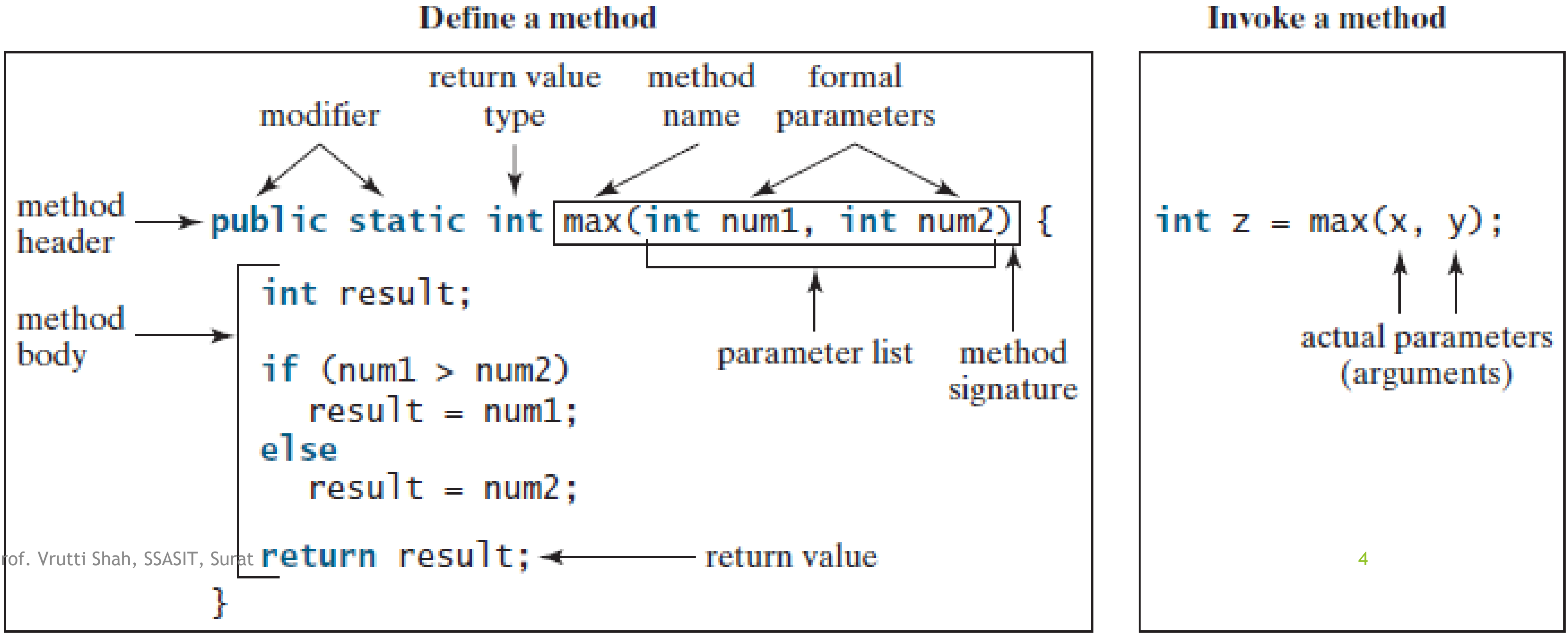
```
public static int sum(int i1, int i2) {
int result = 0;
for (int i = i1; i <= i2; i++)
result += i;
return result;
}

public static void main(String[] args) {
System.out.println("Sum from 1 to 10 is " + sum(1, 10));
System.out.println("Sum from 20 to 37 is " + sum(20, 37));
System.out.println("Sum from 35 to 49 is " + sum(35, 49));
}
```

Defining a Method

- *A method definition consists of its method name, parameters, return value type, and body*
- modifier returnValueType methodName(list of parameters) {

  // Method body;

  }



```
public static int max(int num1, int num2) {
    int result;

    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
}
```

```
int z = max(x, y);
```

Defining a Method (Contd…)

- *method header* specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method
- **returnValueType** is the data type of the value the method returns
- Some methods do not return a value, for that **returnValueType** is the keyword **void**
- If a method returns a value, it is called a *value-returning method*
- variables defined in the method header are known as *formal parameters*
- when a method is invoked, you pass a value to the parameter, this value is referred to as an *actual parameter*
- method name and the parameter list together constitute the *method signature*

Defining a Method (Contd…)

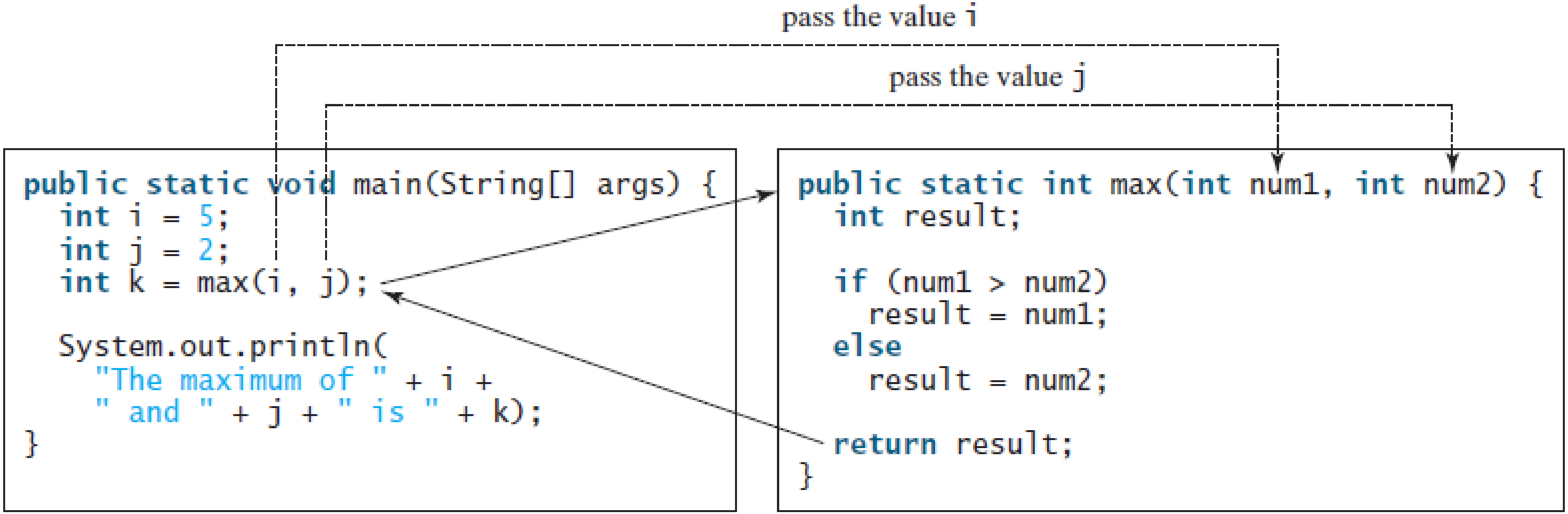
- method body contains a collection of statements that implement the method
- for a value-returning method to return a result, a return statement *required*
- method terminates when a return statement is executed

Calling a Method

- *Calling a method executes the code in the method*
- In a method definition, you define what the method is to do
- To execute the method, you have to *call* or *invoke* it
- If a method returns a value
  - **int** larger = max(**3**, **4**);
  - System.out.println(max(**3**, **4**));
- If a method returns **void**, a call to the method must be a statement
  - System.out.println("**Welcome to Java!**");

Calling a Method (contd…)

- When a program calls a method, program control is transferred to the called method
- A called method returns control to the caller when
  - its return statement is executed or
  - its method ending closing brace is reached



pass the value i

pass the value j

```java
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum of " + i +
    " and " + j + " is " + k);
}
```

```java
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Passing Arguments by Values

- *arguments are passed by value to parameters when invoking a method*
- *parameter order association*: When calling a method, provide arguments, which must be given in the same order as their respective parameters in the method signature
- arguments must match the parameters in *order, number,* and *compatible type,* as defined in the method signature
- *pass-by-value*: When you invoke a method with an argument, the value of the argument is passed to the parameter

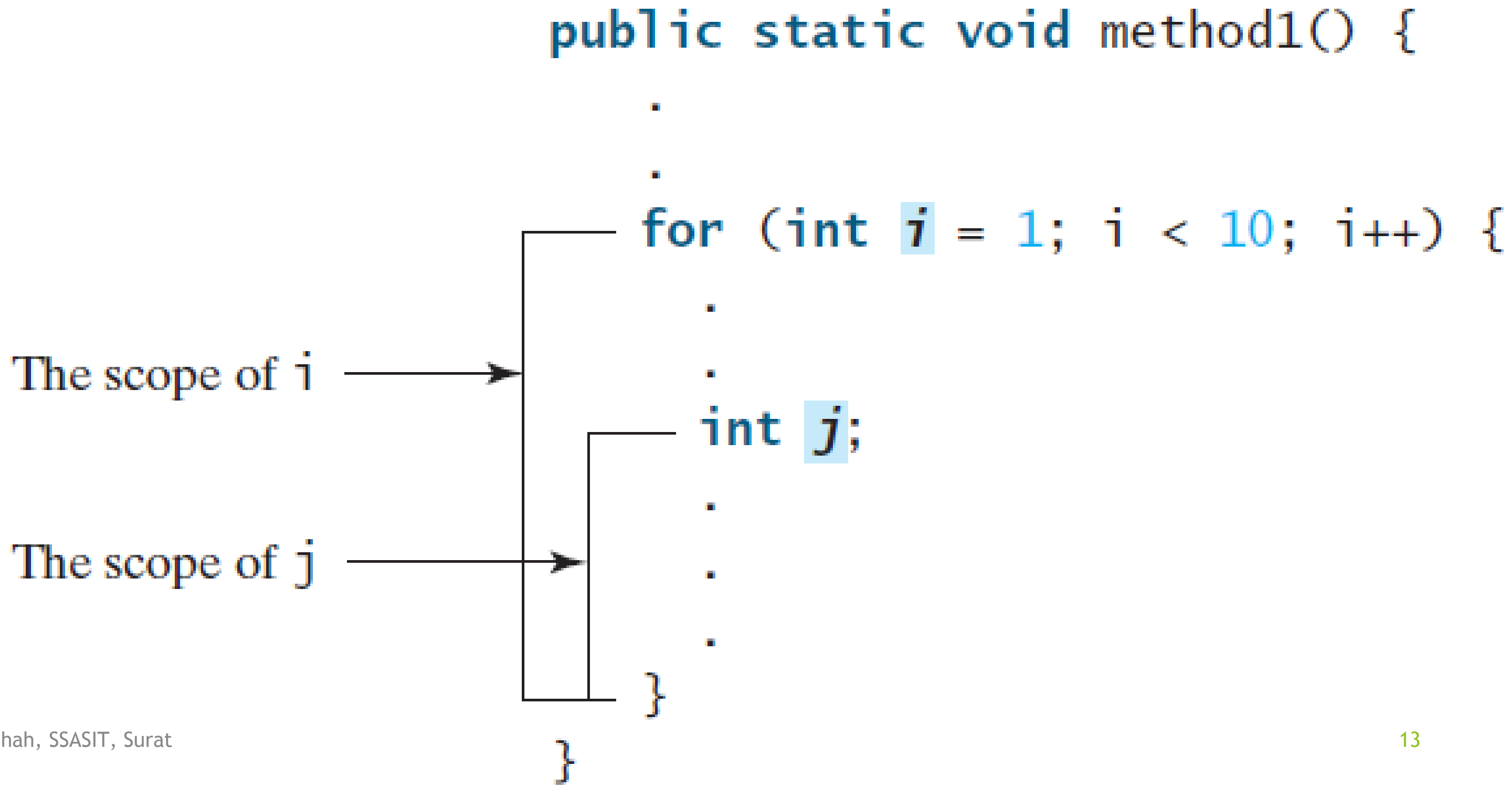
Overloading Methods

- *Overloading methods enables you to define the methods with the same name as long as their signatures are different*
- Methods that perform the same function with different types of parameters should be given the same name.
- Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types
- *ambiguous invocation*: resulting in a compile error

Scope of Variables

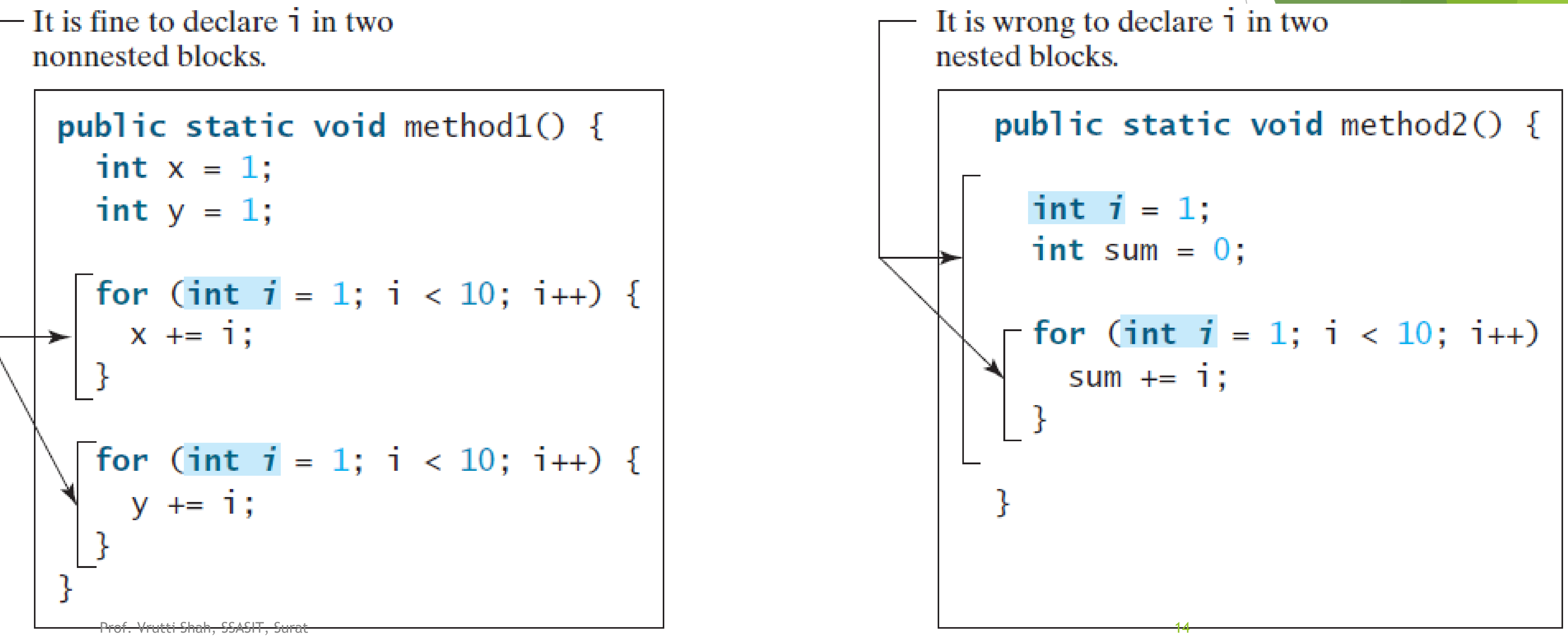
- *scope of a variable is the part of the program where the variable can be referenced.*
- variable defined inside a method is referred to as a *local variable*
- scope of a local variable starts from its declaration and continues to the end of the block
- local variable must be declared and assigned a value before it can be used

- variable declared in the initial-action part of a for-loop header has its scope in the entire loop
- variable declared inside a for-loop body has its scope limited in the loop body from its declaration to the end of the block



```
public static void method1() {
   .
   .
   for (int i = 1; i < 10; i++) {
      .
      .
      int j;
      .
      .
      .
   }
}
```

- You can declare a local variable with the same name in different blocks in a method
- you cannot declare a local variable twice in the same block or in nested blocks



It is fine to declare i in two nonnested blocks.

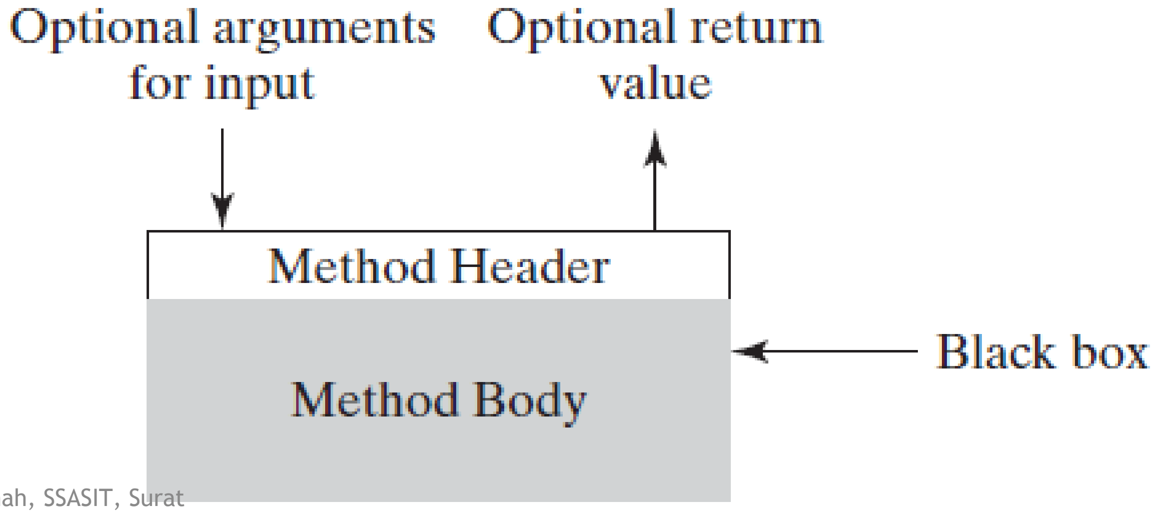
```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

It is wrong to declare i in two nested blocks.

```
public static void method2() {

  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++)
    sum += i;
  }

}
```

Method Abstraction and Stepwise Refinement

- *Method abstraction*: to separate the use of a method from its implementation
- client can use a method without knowing how it is implemented
- *Information hiding* or *encapsulation*: details of the implementation are encapsulated in the method and hidden from the client



- When writing a large program, you can use the *divide-and-conquer* strategy (*stepwise refinement*) : decompose it into subproblems
- subproblems can be further decomposed into smaller

Benefits of Stepwise Refinement

- Simpler Program
  - Rather than writing a long sequence of statements in one method, stepwise refinement breaks it into smaller methods
- Reusing Methods
  - promotes code reuse within a program
- Easier Developing, Debugging, and Testing
  - method can be developed, debugged, and tested individually
  - makes developing, debugging, and testing easier
- Better Facilitating Teamwork
  - subproblems can be assigned to different programmers
  - makes it easier for programmers to work in teams