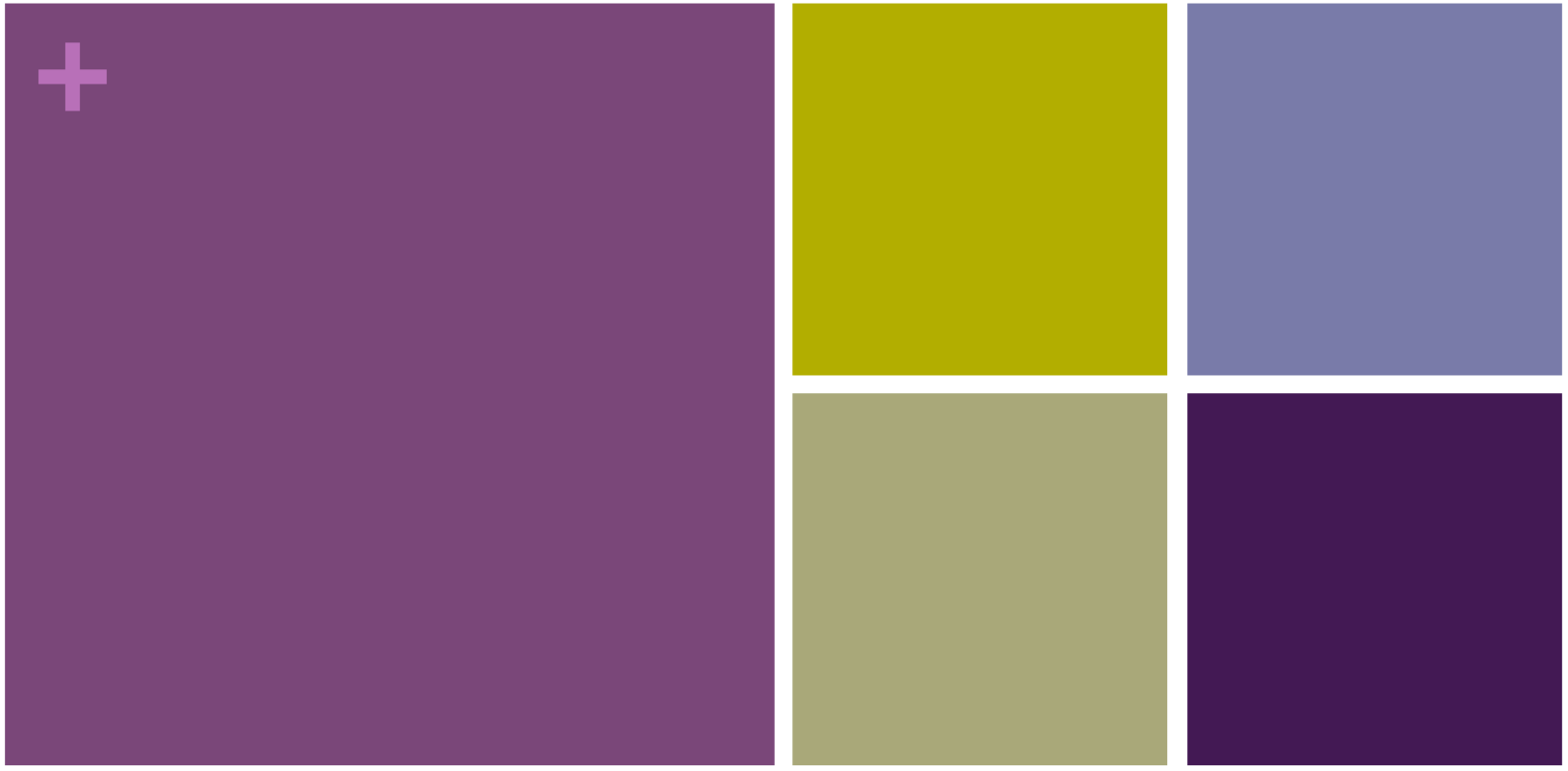


Methods

Introduction to Computer Science CSCI-UA.0101

Lecture 7

Agenda - Day 7

- Review Methods
- Memory Management: Stack memory
- The Scope of Variables
- Modularizing Code
- Overloading Methods
- Stepwise Refinement

Introduction to Methods

- A method is a collection of statements that have been organized and packaged together to perform a specific operation.
- Methods can be used to define code that can be easily re-used in your programs and in programs written by others.
- In other languages we would refer to a method as a "function" – in Java we use the term "method," but the two are quite similar in design.
- Note: we have been using methods since the first day of class

```
System.out.println("hello");
Math.pow(5,2);
Math.random();

public static void main(String[] args)
```

Defining a Method

- Methods in Java must exist within a class definition block.
- Methods cannot be nested inside of one another.
- Methods exist at the same level as other methods within a class (such as the main method).
- Methods begin with a header which describes to Java how the method is supposed to operate. Here's the syntax:

```
modifier returnTypeValue methodName (parameters) {
      // statement(s)
}
```

Example:

```
public static int addTwo(int a, int b) {
      // statement(s)
}
```

Defining a Method

- Refer to the following method header syntax:

```
modifier returnTypeValue methodName (parameters) {
       //  statement(s)
}
```

- The **modifier** for all methods we will write at this point in the semester will be "public static", just like our main method. This will change when we begin to explore object oriented programming.

- The **returnTypeValue** defines what kind of data the method will return upon its successful completion. This can be any of the data types we have discussed so far (int, double char, etc). If a method does not return any value you need to set its returnTypeValue as "void". If you will be returning a value from a method you MUST use the "return" command somewhere in the method to return an object of the correct type upon completion of the method.

Defining a Method

- Refer to the following method header syntax:

```
modifier returnTypeValue methodName (parameters) {
        // statement(s)
}
```

- The **methodName** for a method is a name of your choosing. Be sure to conform to the Java standards for identifiers (no spaces, no leading numbers, etc)

- The **parameters** for a method are placed within the parenthesis after the methodName. These are values that you will be passing into the method when you invoke or call it. Methods can accept zero or more parameters. For example:

```
// no parameters
public static void method1 () { }

// 1 parameter
public static void method2 (int a) { }

// 2 parameters
public static void method3 (int a, int b) { }
```

Calling a Method

- A method definition is used to define what you want a method to do.
- In order to run the statements inside your method you must invoke or "call" the method from another part of your program.
- If your method does not return a value you can simply type its name along with any parameters as a statement. For example:

```
// call the awesome method which does not have
// a return value
awesome();
```

- If your method does return a value you can treat the method as though it was that value. For example:

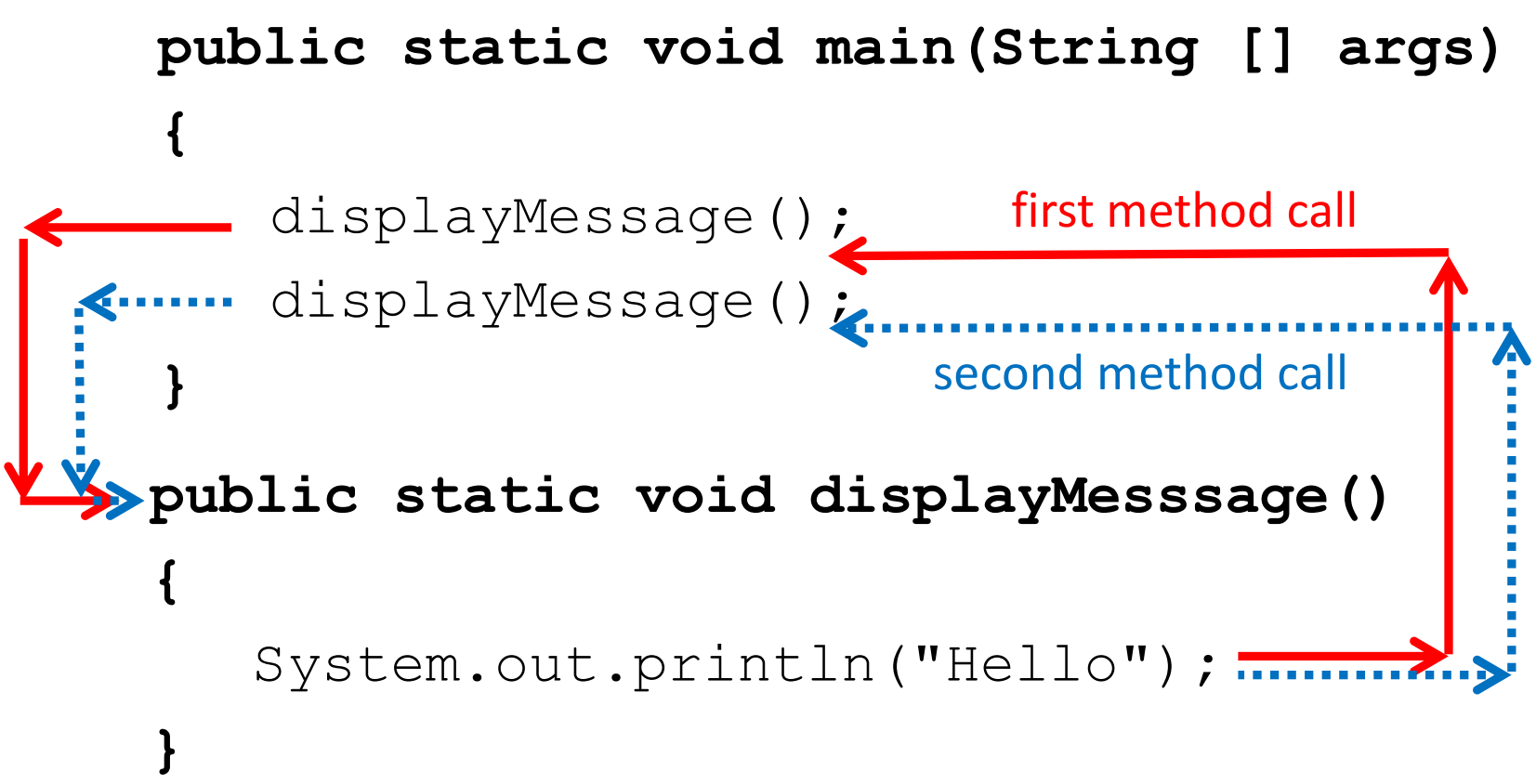
```
// call our method and store the result
int answer1 = addTwoNumbers(5, 7);

// call our method and use the result immediately
int answer2 = 100 + addTwoNumbers(50, 51);
```

Calling a Method

- When a program calls a method the control of the program is transferred to the method.
- Control is returned to the caller of the method when the method reaches the end of its block of execution, or when the "return" statement is called from within the method. Note that methods that return a vaule MUST have contain a "reachable" return statement.

Control flow in program execution



Void Methods

- A void method is a method that does not return a value to its caller.

- Void methods must be called as a statement and cannot be treated as data type (i.e. void methods do not appear in the right hand side of an assignment operator.)

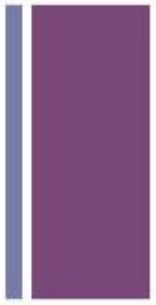
  Example:

  displayMessage();

Passing Parameters by Value

- The power of methods lie in their ability to accept parameters – or arguments – from a caller and then operate on this data.
- When designing methods you need to articulate the type of data you will be sending that method along with the order in which it can be expected. This is commonly referred to as "parameter-order association"

Passing Arguments to a Method



- Values that are sent into a method are called arguments.

```
System.out.println("Hello");
number = Integer.parseInt(str);
```

- The data type of an argument in a method call must correspond to the variable declaration in the parentheses of the method declaration. The parameter is the variable that holds the value being passed into a method.
- By using parameter variables in your method declarations, you can design your own methods that accept data this way.

Passing a value to **displayValue()**

```
displayValue(5);  //called from main
```

The argument 5 is copied into the parameter variable **num**.

```
public static void displayValue(int num)
{
  System.out.println("The value is " + num);
}
```

The method will display **The value is 5**

Passing Multiple Arguments

The argument 5 is copied into the **num1** parameter.

The argument 10 is copied into the **num2** parameter.

NOTE: Order matters!

```
showSum(5, 10); //called from main

public static void showSum(double num1, double num2)
{
    double sum;     //to hold the sum
    sum = num1 + num2;
    System.out.println("The sum is " + sum);
}
```

Arguments are Passed by Value

- In Java, all arguments of the primitive data types are *passed by value*, which means that only a copy of an argument's value is passed into a parameter variable.

- A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.

- If a parameter variable is changed inside a method, it has no affect on the original argument.

Passing Parameters by Value

```
public static void printMe(String message, int times)
{
      for (int i = 0; i < times; i++)
      {
            System.out.println(message);
      }
}
```

Valid or invalid method call?

```
printMe("hello", 4);
printMe(4, "hello");
printMe();
printMe("hello");
```

Passing Parameters by Value

- When you call a method with an argument the value of the argument is passed into the method.
- This means that the variable being sent to the method by the caller is unaffected by any changes made in the method.
- The variables being sent occupy their own memory addresses, and their values are copied to a new memory address and assigned new identifiers during the method call.

```
public static void main(String[] args)
{
	int y = 0;
	System.out.println("Before method: " + y);
	inc(y);
	System.out.println("After method: " + y);
}

public static void inc(int x)
{
	x++;
	System.out.println("Inside method: " + x);
}
```

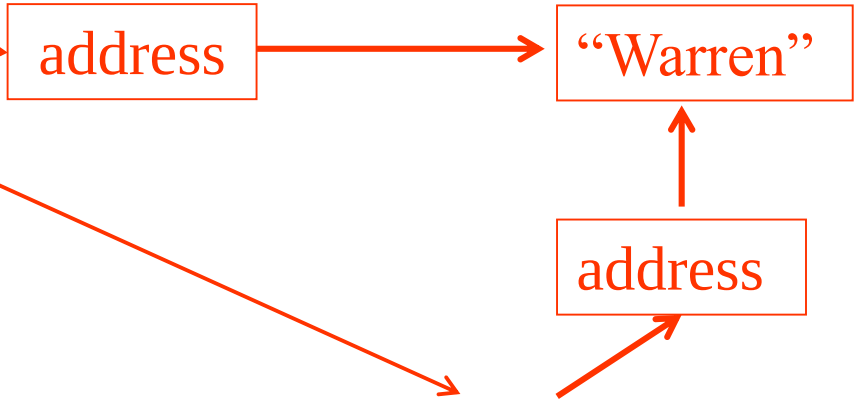
Passing Object References to a Method

- Recall that a class type variable does not hold the actual data item that is associated with it, but holds the memory address of the object. A variable associated with an object is called a reference variable.

- When an object such as a `String` is passed as an argument, it is actually a reference to the object that is passed.

Passing a Reference as an Argument

```
String name = "Warren";
showLength(name);
```



-The address of the object is copied into the **str** parameter.

-**name** and **str** reference the string object "Warren"

```
public static void showLength(String str)
{
  System.out.println(str + " is " + str.length()+
                                    " characters long.");
}
```

Value-returning Methods

- Value-returning methods return a value to the calling statement.

- Value-returning methods must include a "return" statement in the definition of the method.
- If a method returns a value, a call to the method is usually treated as a value (i.e. it appears on the right hand side of an assignment operator.

  Example:

  int larger = max(3, 4);

Returning a Value from a Method

- Data can be passed into a method by way of the parameter variables. Data may also be returned from a method, back to the statement that called it.

```
int num = Integer.parseInt("700");
```

- The string “700” is passed into the `parseInt` method.
- The `int` value 700 is returned from the method and assigned to the `num` variable.

Defining a Value-Returning Method

```
public static int sum(int num1, int num2)
{
  int result;
  result = num1 + num2;
  return result;
}
```

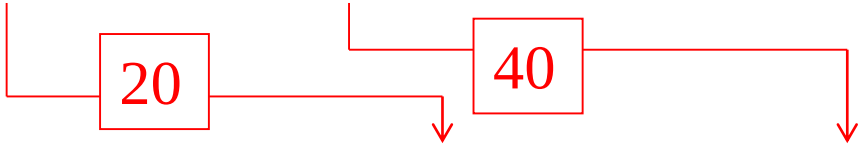
return type

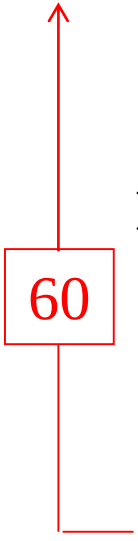
This expression must be of the same data type as the return type

The `return` statement causes the method to end execution and it returns a value back to the statement that called the method.

Calling a Value-Returning Method

```
int value1 = 20, value2 = 40;
total = sum(value1, value2); //from main
```






```
public static int sum(int num1, int num2)
{
  int result;
  result = num1 + num2;
  return result;
}
```

Q4: Read/Analyze the following code

```
public class MyClass
{
   public static void main(String [] args)
   {
        int value = 20;
        if(isValid(value))
                System.out.println("The value is within range");
        else
                System.out.println("The value is out of range");
   }
   public static boolean isValid(int number)
   {
       boolean status;
       if(number >= 1 && number <= 100)
          status = true;
       else
          status = false;
       return status;
    }
}
```

1) Name the methods used in the code?
2) Is local variable `number` known to main?
3) Which is the type of the value returned by isValid?
4) Which is the output produced by the code?

Returning a Reference to a `String`

```
customerName = fullName("John", "Martin");
```

```
public static String fullName(String first, String last)
{
        String name;
        name = first + " " + last;
        return name;
}
```

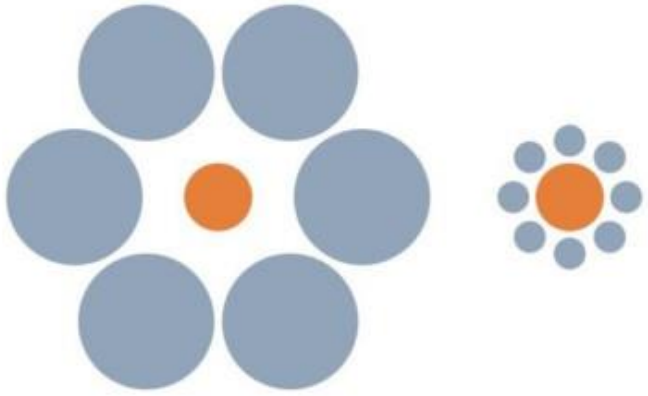
address

"John Martin"

Local variable `name` holds the reference to the object. The return statement sends a copy of the reference back to the call statement and it is stored in `customerName`.

Programming Challenge (value-returning method)

- Write a method called "max" that accepts two integers as parameters.
- Return the larger of the two integers to the caller of the method.
- **CalcMax.java**



Programming Challenge

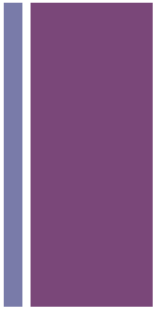
- Write a method called "sumRange" that accepts two integers as parameters.
- Sum up the numbers in the range provided and return the result to the caller.
- **Range.java**

Summary: Problem Solving with Methods

- A large, complex problem can be solved by breaking it down in methods.
- The process of breaking a problem down into smaller pieces is called *functional decomposition (divide and conquer)*.
- To use (call) a method you need to write a statement in **`main`** that calls the method.
- To define a method you need to specify the *header* (arguments, return value) and the *body* of the method (what the method does or computes).

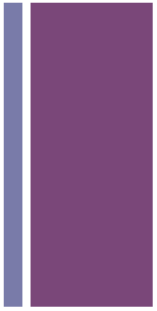
Memory Management: Call Stack Memory

The Stack



- Every time a method is called Java creates an "activation record" that stores information about the method call, such as the name of the method and any variables that are involved in the call.
- The activation record is placed into an area of memory called the "stack". The caller's activation record is also in memory on the stack.
- The stack operates as a "first in, last out" construct. This means that activation records are placed on top of the stack when a method is called. When the method call terminates the activation record is removed and control is passed back to the caller's activation record.

The Stack



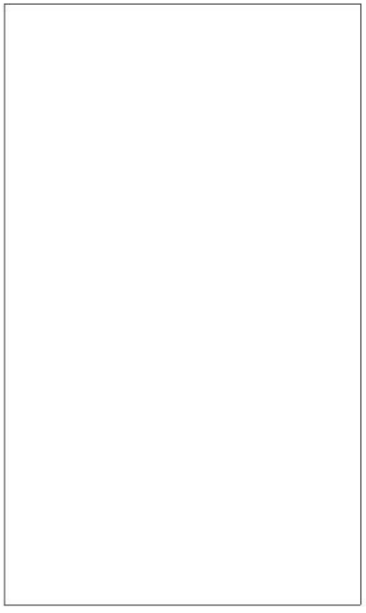
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

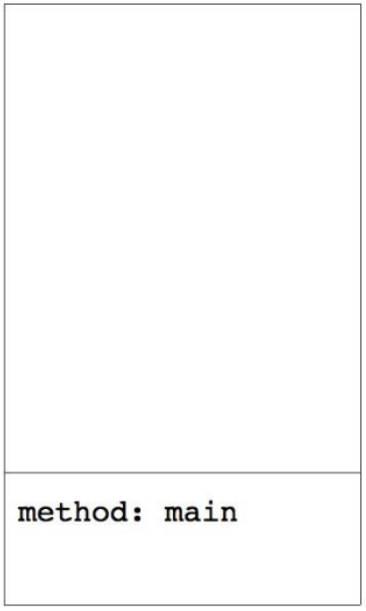
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

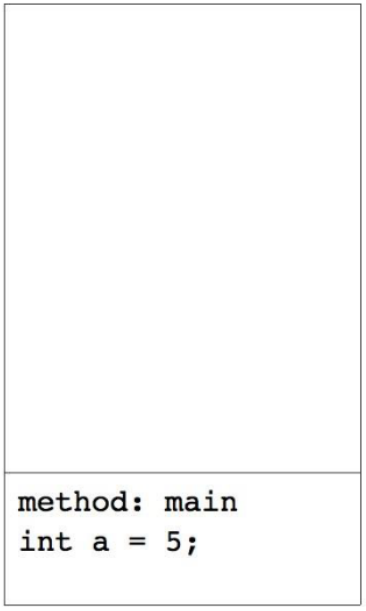
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

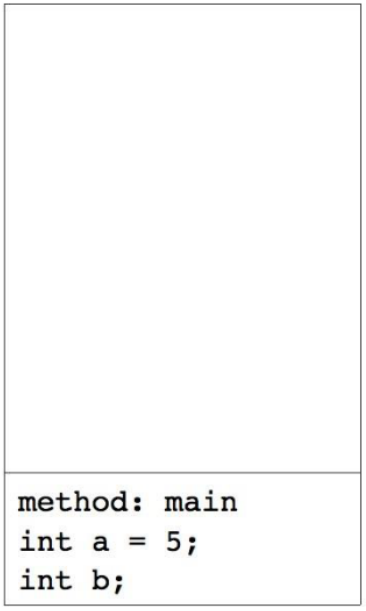
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

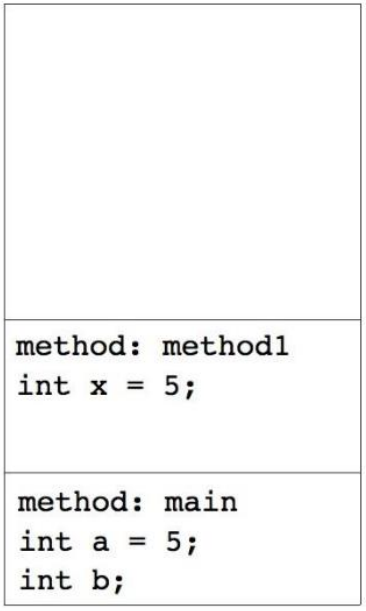
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

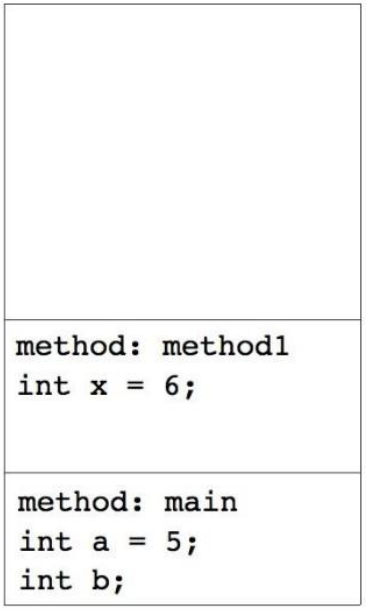
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

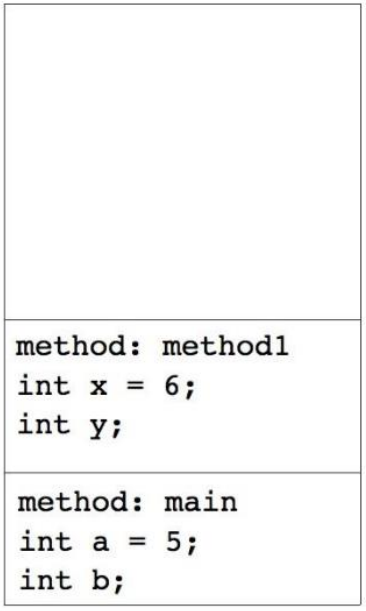
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

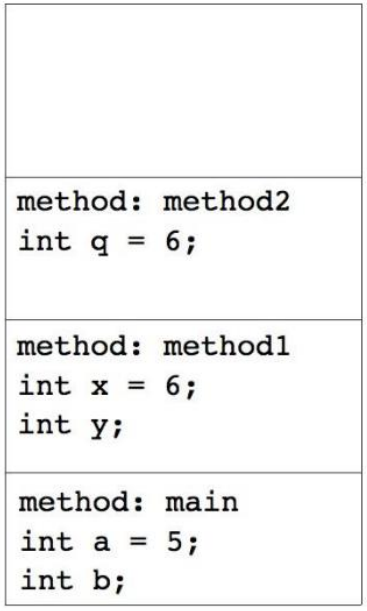
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

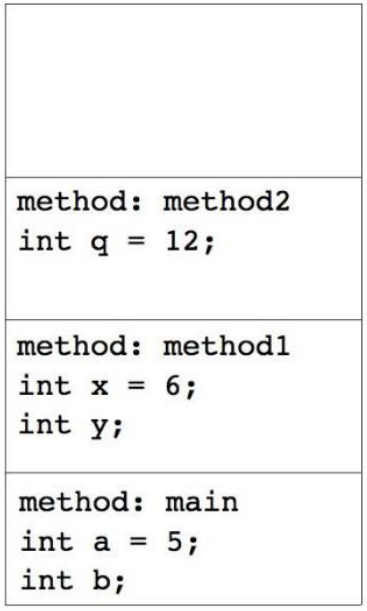
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

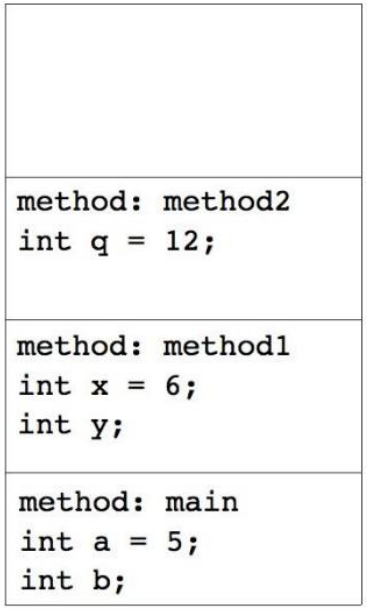
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

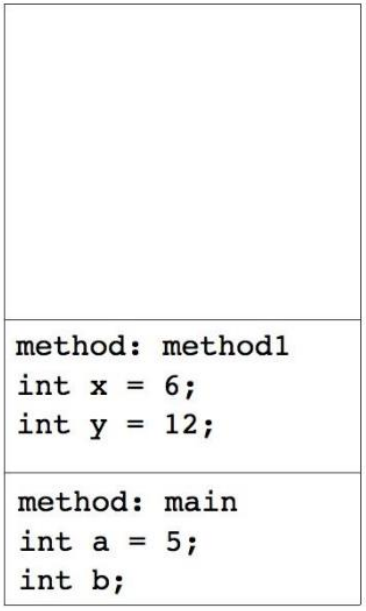
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

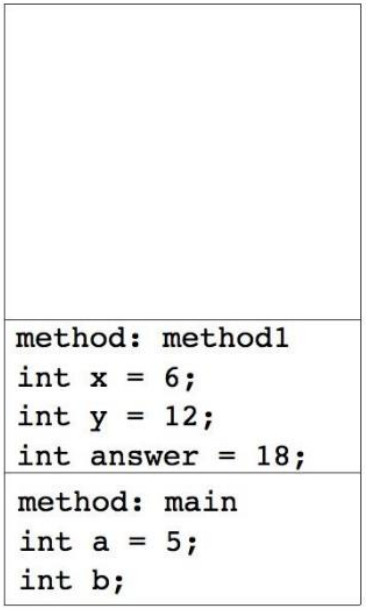
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

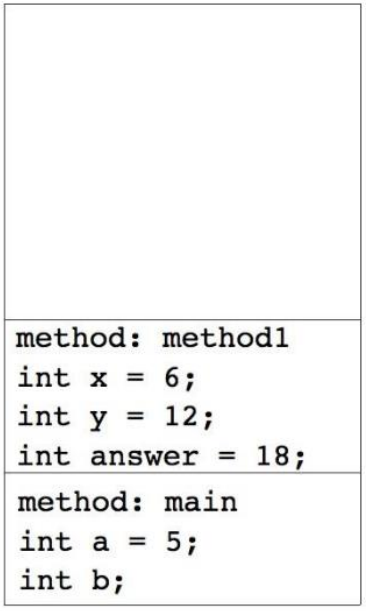
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

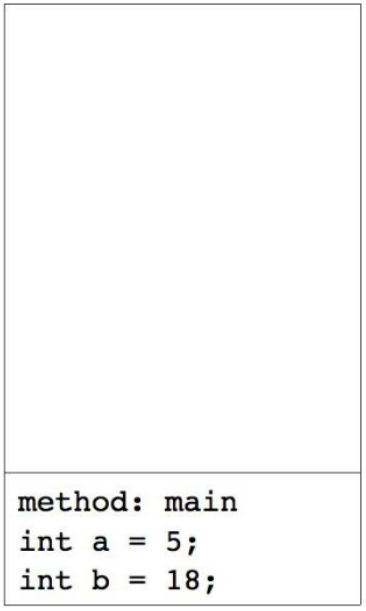
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



The Stack

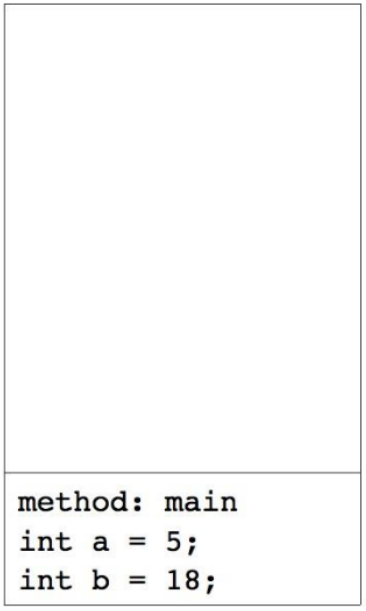
Source Code

```
public static void main(String[] args)
{
    int a = 5;
    int b = method1(a);
    System.out.println(b);
}

public static int method1(int x)
{
    x += 1;
    int y = method2(x);
    int answer = x + y;
    return answer;
}

public static int method2(int q)
{
    q *= 2;
    return q;
}
```

The Stack



Variable Scope

Variable Scope

- A variable's "scope" is the region of the program where the variable can be referenced by your code.
- In Java, variables defined inside of a method are considered "local" to that method and are inaccessible outside of the method.
- In addition, variables declared inside a block are local to that block and cannot be referenced outside of that block (but the can be referenced by blocks nested inside of the block)
- Variables defined in the initialization of a "for" loop are local to the entire loop, but are inaccessible outside the block of the loop.

Variable Scope (Examples)

- **VariableScope1.java**
- **ClassMethodVariableScope.java**

Modularizing Code

Modularizing Code

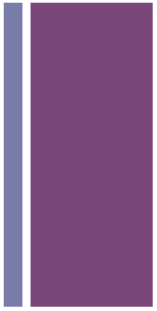
- Methods are often used to reduce redundant code and to facilitate code reuse by other programs.
- For example:

```
for (int i = 0; i < 100; i++)
{
      System.out.println("hello");
}
```

Can be modularized into:

```
public static void m0(int times)
{
      for (int j = 0; j < times; j++)
      {
            System.out.println("hello");
      }
}
```

Modularizing Code



- When modularizing your code you should attempt to identify which aspects of your code can be "encapsulated"
- This allows you to isolate problems in your logic and break down difficult tasks into smaller, more discrete unites
- This technique also allows you to re-use your code in other programs without having to re-write or copy code from program to program.

Programming Challenge

- Write a program to produce a sales report modularizing your code:

  1) ask the user to enter the daily sales

  2) calculate the average daily sales

  3) display the total and average

- Each step should be handled by a different method.

- **Sales.java**

Overloading Methods

Overloading Methods

- When calling a method you must part careful attention to the parameters you are sending. Any mismatch – either in number, data type or parameter ordering – can raise an exception and cause your program to crash. For example:

```
public static void main (String[] args)
{
     double x = max(1.25, 5.75);
}

public static int max(int a, int b)
{
     if (a < b) { return b; }
     else { return a; }
}
```

Overloading Methods

- Java supports a technique called "overloading" that lets you define multiple versions of the same method that can be invoked using different parameter sets.
- When overloading a method you simply define a different method for each version you wish to create, making sure that each version uses the same name but different parameter sets. Java will figure out at compile time which method to invoke based on the data provided. Example:

```
public static int addTwo (int a, int b)

public static double addTwo (double a, double b)
```

Programming Challenge

- Write a method that calculates the maximum value of 2 integers.
- Then write an overloaded version of this method that calculates the maximum value of 2 doubles.
- Finally, write an overloaded version of this method that calculates the maximum value of 3 doubles.

MethodOverloading.java