

COP 3330: Object-Oriented Programming Summer 2011

Methods In Java – A Closer Look

Instructor : Dr. Mark Llewellyn
 markl@cs.ucf.edu
 HEC 236, 407-823-2790
 <http://www.cs.ucf.edu/courses/cop3330/sum2011>

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida



Methods In Java

- A method is a construct for grouping statements together to perform some function. By writing a method, you can write the code once for performing the function in a program and reuse it in many other programs.
- For example, when you need to find the maximum between two numbers. Whenever you need this function, you have to write the following code:

```
int num1, num2, result;  
if (num1 > num2)  
    result = num1;  
else  
    result = num2;
```

- By defining a method for this code, you do not have to repeatedly write the code.



Methods In Java

- We've already been using several methods in some of the sample programs we've seen. For example, we've used:

```
System.out.print
```

```
JOptionPane.showInputDialog
```

```
System.out.println
```

```
JOptionPane.showMessageDialog
```

```
Double.parseDouble
```

- All of these methods are defined in various Java libraries. You've also been creating your own methods based on UML class diagrams and are now familiar with the differences between class and instance methods.
- We now want to consider more complex problems and learn the concept of method abstraction.



Defining Methods In Java

- The syntax for defining a method in Java is:

```
modifier returnType methodName (parameter list)
{
    //method body
}
```



Defining Methods In Java

- The **method header** specifies the **modifiers**, **return value**, **method name**, and **parameters** of the method.
- A method may return a value. The `returnValueType` is the data type of the value the method returns.
- Some methods may perform their desired operations without returning a value. In this case, the `returnValueType` is the keyword **void**.
- A method that returns a value is called a **value-returning method**, and a method that does not return a value is called a **void method**.



Defining Methods In Java

- The variables defined in method header are known as **formal parameters** or simply **parameters**. A parameter is like a placeholder. When a method is invoked, a value is passed to the parameter. This value is referred to as an **actual parameter** or **argument**.
- The **parameter list** refers to the type, order, and number of the parameters of a method.
- The method name and the parameter list together constitute the **method signature**.
- It is possible for a method to have no parameters.



Defining Methods In Java

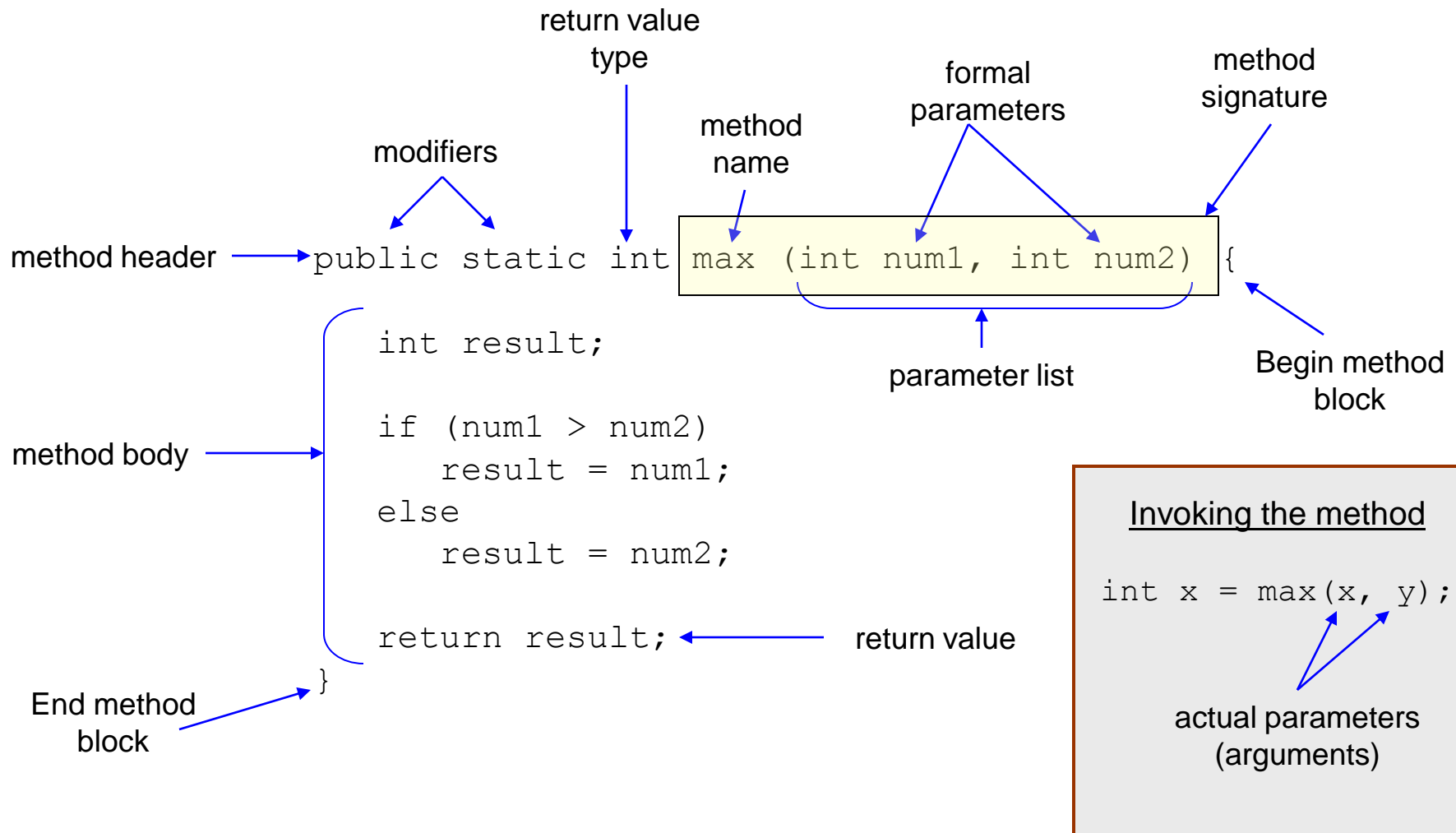
- The method body contains a collection of statements that define what the method does, i.e., its functionality.
- A return statement, using the keyword `return`, is required for a value-returning methods to return a result.
- A method terminates when a return statement is executed.

NOTE:

1. In other languages, methods are referred to as procedures and functions. A value-returning method is called a function, a `void` method is called a procedure.
2. You need to declare a separate data type for each parameter. For instance, `int num1, num2` should be replaced by `int num1, int num2`.
3. A return statement can be included in a `void` method and is used for terminating the method and returning control to the caller, the syntax is simply `return;`.



Defining Methods In Java



Invoking (Calling) Methods In Java

- By creating a method, you give a definition of what the method is to do. To use a method, you have to **call** or **invoke** it.
- There are two ways to invoke a method; the choice is based on whether the method returns a value or not.
- If the method returns a value, a call is usually treated as a value.
 - For example: `int larger = max(3, 4);`
calls `max(3, 4)` and assigns the result of the method to the variable `larger`.
 - Another example is: `System.out.println(max(3, 4));` which prints the
return value of the method call `max(3, 4)`.
- If the method returns `void`, a call to the method must be a statement.
 - For example, the method `println` returns `void` and the following call is a statement:
`System.out.println("Hello!");`



Invoking (Calling) Methods In Java

- It is also possible, although rare, that a value-returning method can be invoked as a statement in Java. In such a case, the caller simply ignores the return value.
- When a program invokes 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 when its method ending closing brace is reached.



```
/* TestMax class - simple method illustration
 *
 * Mark Llewellyn - 6/21/2011
 * No known bugs
 */
import java.util.Scanner;

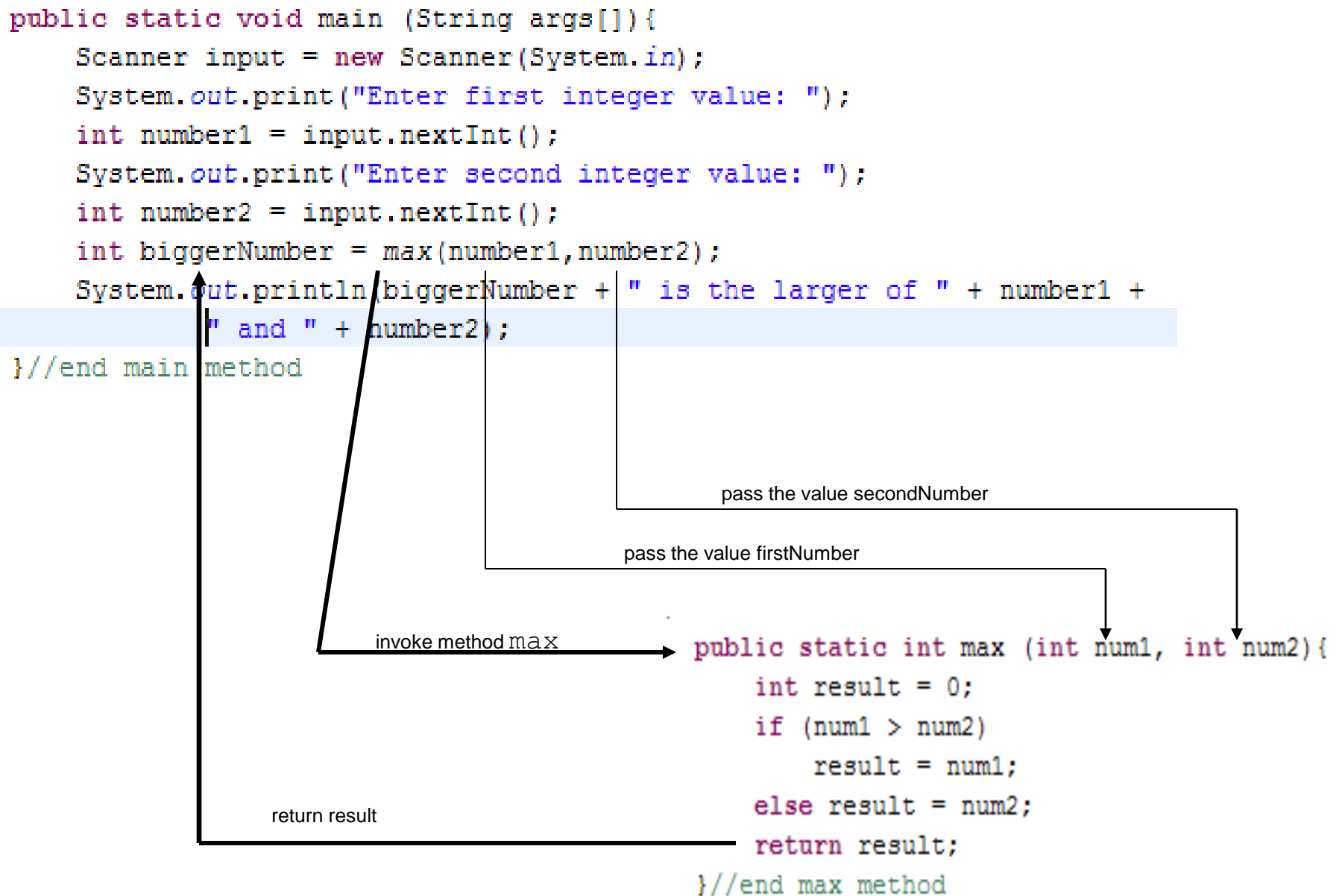
public class TestMax {
/**
 * @param args
 */
    public static int max (int num1, int num2){
        int result = 0;
        if (num1 > num2)
            result = num1;
        else result = num2;
        return result;
    }//end max method

    public static void main (String args[]){
        Scanner input = new Scanner(System.in);
        System.out.print("Enter first integer value: ");
        int number1 = input.nextInt();
        System.out.print("Enter second integer value: ");
        int number2 = input.nextInt();
        int biggerNumber = max(number1,number2);
        System.out.println(biggerNumber + " is the larger of " + number1
            " and " + number2);
    }//end main method
} //end TestMax class
```

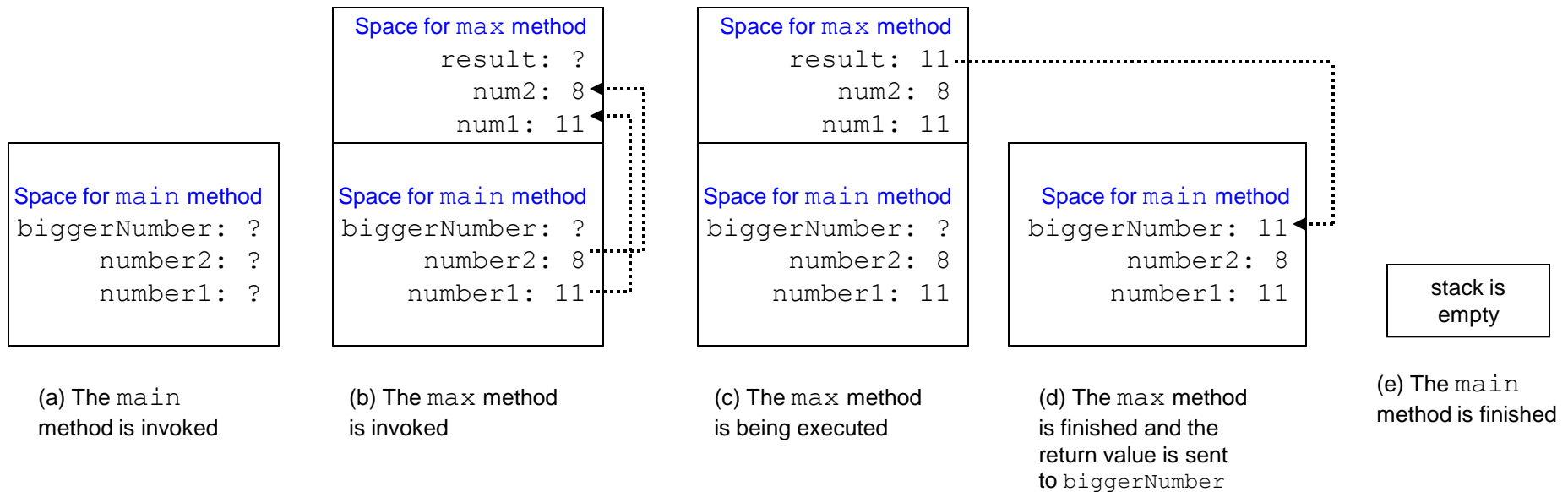
Example of a value-returning method

This method is a class method that requires two integer parameters and returns an integer value.





Call Stacks



```
/* TestVoidMethod - illustrate a void method in Java
 *
 * Mark Llewellyn - 6/21/2011
 * No known bugs
 */
import java.util.Scanner;

public class TestVoidMethod {

    public static void printGrade(double score) {
        if (score >= 90.0)
            System.out.println("Your grade is an A.");
        else if (score >= 80.0)
            System.out.println("Your grade is a B.");
        else if (score >= 70.0)
            System.out.println("Your grade is a C.");
        else if (score >= 60.0)
            System.out.println("Your grade is a D.");
        else System.out.println("Your grade is an F.");
        return;
    } //end printGrade method

    public static void main (String args[]) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the score: ");
        double score = input.nextDouble();
        TestVoidMethod.printGrade(score);
    } //end main method
} //end TestVoidMethod
```

Example of a void method

This method is a class method that requires one double type parameter and does not return a value. Notice that it does have a return statement.



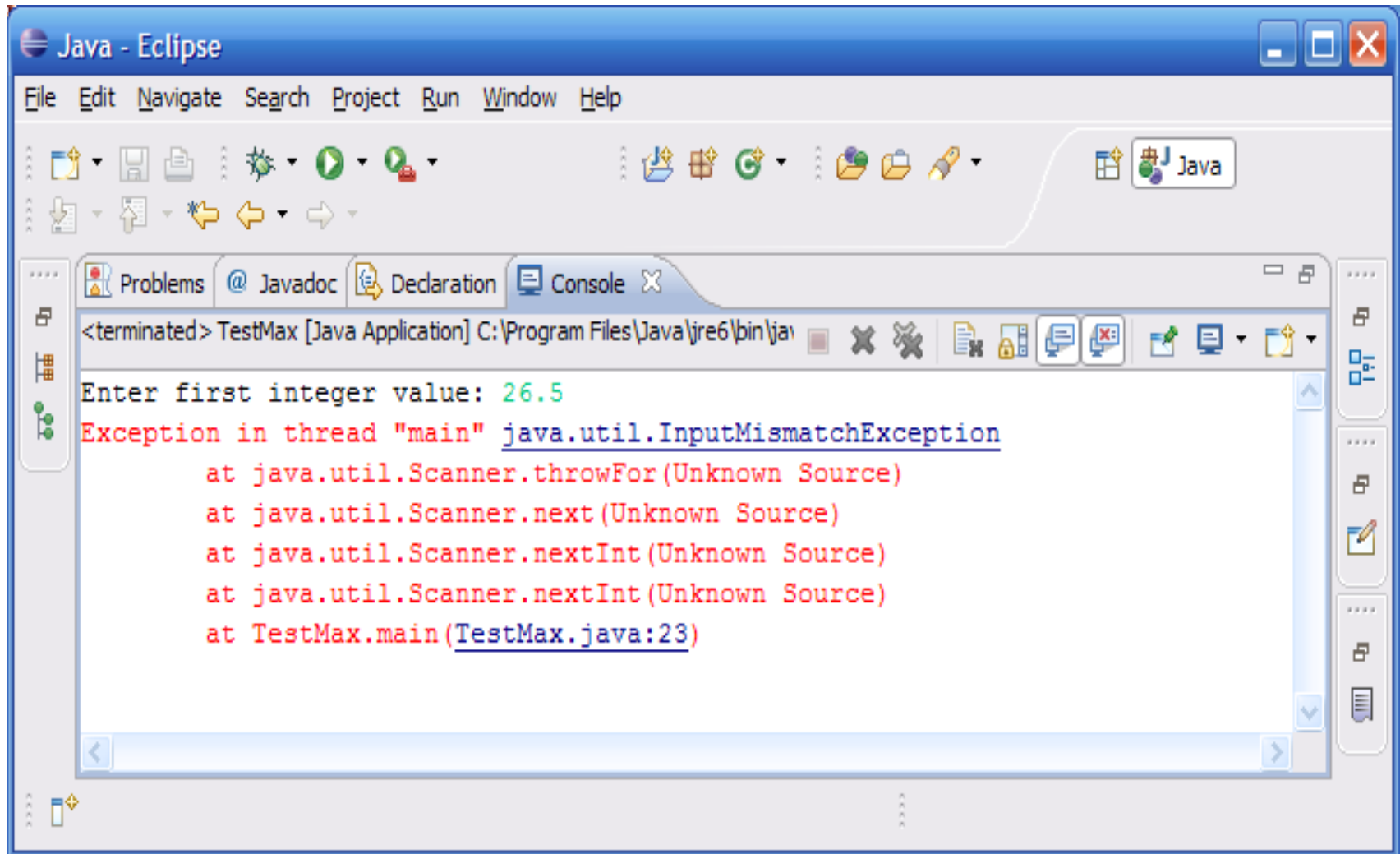
Overloading Methods

- The `max` method that we created in the previous value returning method example works only with integer parameters. What happens if you try to enter doubles? (Try it and see what happens! – or see next page!)
- The solution is to create another method with the same name but with an otherwise different signature. This is called **method overloading**.
- For example (also see pages 18-20):

```
public static double max (double num1, double num2) {  
  
    double result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```



Overloading Methods



The screenshot shows the Eclipse IDE interface. The console window is active, displaying the following text:

```
<terminated> TestMax [Java Application] C:\Program Files\Java\jre6\bin\jav  
Enter first integer value: 26.5  
Exception in thread "main" java.util.InputMismatchException  
    at java.util.Scanner.throwFor(Unknown Source)  
    at java.util.Scanner.next(Unknown Source)  
    at java.util.Scanner.nextInt(Unknown Source)  
    at java.util.Scanner.nextInt(Unknown Source)  
    at TestMax.main(TestMax.java:23)
```



Overloading Methods

Method Signatures

The **method signature** refers to the name of the method and the number and types of its parameters. The return type of the method is **not** part of its signature.

This means that two methods cannot be overloaded just because they have different return types, because they would still have the same signature. Thus, the following two methods could not be defined within the same class:

```
public void aMethod(int a, int b) { . . . }  
public int aMethod(int a, int b) { . . . }
```

However, the following two methods could be defined in the same class as an overloaded method:

```
public int anotherMethod(int a, double b) { . . . }  
public int anotherMethod(double a, double b) { . . . }
```



```
/* TestMaxWithOverloadedMethod class - simple class illustrating
 * method overloading.
 *
 * Mark Llewellyn - 6/21/2011
 * No known bugs
 */
import java.util.Scanner;

public class TestMaxWithOverloadedMethod {
/**
 * @param args
 */
    //max method for integer values
    public static int max (int num1, int num2){
        int result = 0;
        System.out.println("Inside the max method for integers.");
        if (num1 > num2)
            result = num1;
        else result = num2;
        return result;
    }//end max method for integers


    //max method for double values
    public static double max (double num1, double num2){
        System.out.println("Inside the max method for doubles.");
        double result = 0.0;
        if (num1 > num2)
            result = num1;
        else result = num2;
        return result;
    }//end max method for doubles
}
```



```
public static void main (String args[]){
    Scanner input = new Scanner(System.in);
    System.out.print("Enter first integer value: ");
    int number1 = input.nextInt();
    System.out.print("Enter second integer value: ");
    int number2 = input.nextInt();
    int biggerInteger = max(number1,number2);
    System.out.println(biggerInteger + " is the larger of " + number1 +
        " and " + number2);
    System.out.println();
    System.out.print("Enter first double value: ");
    double number3 = input.nextDouble();
    System.out.print("Enter second double value: ");
    double number4 = input.nextDouble();
    double biggerDouble = max(number3,number4);
    System.out.println(biggerDouble + " is the larger of " + number3 +
        " and " + number4);
    //end main method
} //end TestMaxWithOverloadedMethod class
```



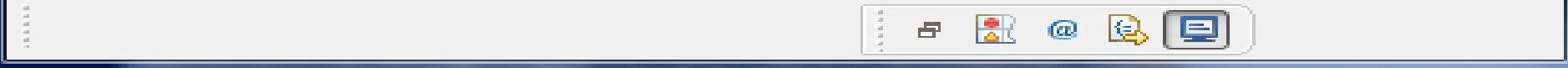
File Edit Source Refactor Navigate Search Project Run Window Help



Console

```
<terminated> TestMaxWithOverloadedMethod [Java Application] C:\Program Files\Java\j
Enter first integer value: 56
Enter second integer value: 11
Inside the max method for integers.
56 is the larger of 56 and 11

Enter first double value: 35.8
Enter second double value: 77.4
Inside the max method for doubles.
77.4 is the larger of 35.8 and 77.4
```



Overloading Methods

- The next example takes overloading to a more extreme case.
- What I did in the following example was to overload the `max` method six times. Be sure you understand this example: not only in how the overloading was done but also in how the correct overloaded method is invoked from within the main method.
- The code for this example begins to get a little bit large to put all of it in the notes, so I placed the actual source code file on the course lecture notes page under the link to this document so that you can look at it and use it. On the next couple of pages, I've only placed selected pieces of this code.
- For practice you should create a couple of more overloaded versions of the `max` method each taking in additional parameters. I'll put a solution on the practice problems later.



```
import java.util.Scanner;

public class TestMaxWithHeavyOverloading {
    /**
     * @param args
     */
    //max method for two integer values
    public static int max (int num1, int num2){
        int result = 0;
        System.out.println("Inside the max method for two integers.");
        if (num1 > num2)
            result = num1;
        else result = num2;
        return result;
    } //end max method for two integer values

    //max method for three integer values
    public static int max(int num1, int num2, int num3){
        System.out.println("Inside the max method for three integers.");
        return max(max(num1, num2), num3);
    } //end max for three integer values

    //max method for four integer values
    public static int max(int num1, int num2, int num3, int num4){
        System.out.println("Inside the max method for four integers.");
        return max(max(max(num1, num2), num3), num4);
    } //end max method for four integer values
}
```

Three overloaded versions
of the `max()` method



```
public static void main (String args[]){
    Scanner input = new Scanner(System.in);
    System.out.print("Enter first integer value: ");
    int number1 = input.nextInt();
    System.out.print("Enter second integer value: ");
    int number2 = input.nextInt();
    System.out.print("Enter third integer value: ");
    int number3 = input.nextInt();
    System.out.print("Enter fourth integer value: ");
    int number4 = input.nextInt();
    int biggerInteger = max(number1,number2);
    System.out.println("\n" + biggerInteger + " is the larger of " + number1 +
        " and " + number2);
    biggerInteger = max(number1,number2, number3);
    System.out.println("\n" + biggerInteger + " is the larger of " + number1 +
        ", " + number2 + " and " + number3);
    biggerInteger = max(number1,number2, number3, number4);
    System.out.println("\n" + biggerInteger + " is the larger of " + number1 +
        ", " + number2 + ", " + number3 + " and " + number4);
}
```

The `main()` method
invocations of the `max()`
method



TestMaxWithHeavyOverloading [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 22, 2011 2:17:31 PM)

Enter first integer value: 2
Enter second integer value: 6
Enter third integer value: 4
Enter fourth integer value: 9
Inside the max method for two integers.

```
int biggerInteger = max(number1,number2);
```

6 is the larger of 2 and 6
Inside the max method for three integers.

```
biggerInteger = max(number1,number2, number3);
```

Inside the max method for two integers.
Inside the max method for two integers.

```
biggerInteger = max(number1,number2, number3, number4);
```

6 is the larger of 2, 6 and 4
Inside the max method for four integers.
Inside the max method for two integers.
Inside the max method for two integers.
Inside the max method for two integers.

9 is the larger of 2, 6, 4 and 9



Method Abstraction And Stepwise Refinement

- The key to developing good software is to apply the concept of abstraction. You'll learn many different levels of abstraction as we work our way through the semester.
- **Method abstraction** is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are **encapsulated** in the method and hidden from the client who invokes the method. This is known as **encapsulation** or **information hiding**.
- If you (the developer) decide to change the implementation, the client program will not be affected provided that you do not change the method signature.



Method Abstraction And Stepwise Refinement

- We've already used the `System.out.print` method to display a string on the terminal (see next page for a refresher) and we've used the `JOptionPane.showInputDialog` method to read a string from a dialog box, we knew how to invoke these methods from our program, but we did not know, nor do we really care to know, how these methods are implemented.
- Thus, we have treated the implementation of these methods as a “black box”, in that we know what kind of service is provided by the black box, but we don't need to know how they are implemented in order for us to take advantage of the service they provide.

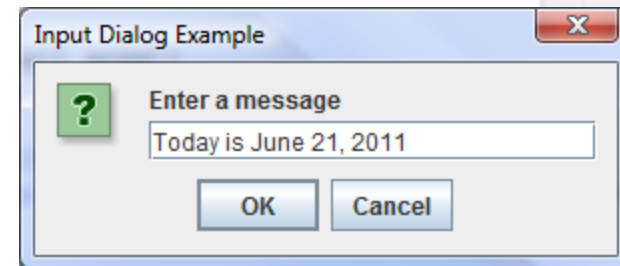


```

/** InputDialogBoxExample illustrates entering input from a dialog box.
 *
 * @author Mark Llewellyn May 23, 2011
 * No known bugs
 */
import javax.swing.JOptionPane;

public class InputDialogBoxExample {
/**
 * @param args
 */
    public static void main (String[] args){
        String input = JOptionPane.showInputDialog(null,
            "Enter a message", "Input Dialog Example",
            JOptionPane.QUESTION_MESSAGE);
        System.out.println("The value entered in the dialog box was: " +
            input);
    } //end main method
} //end class InputDialogBoxExample

```



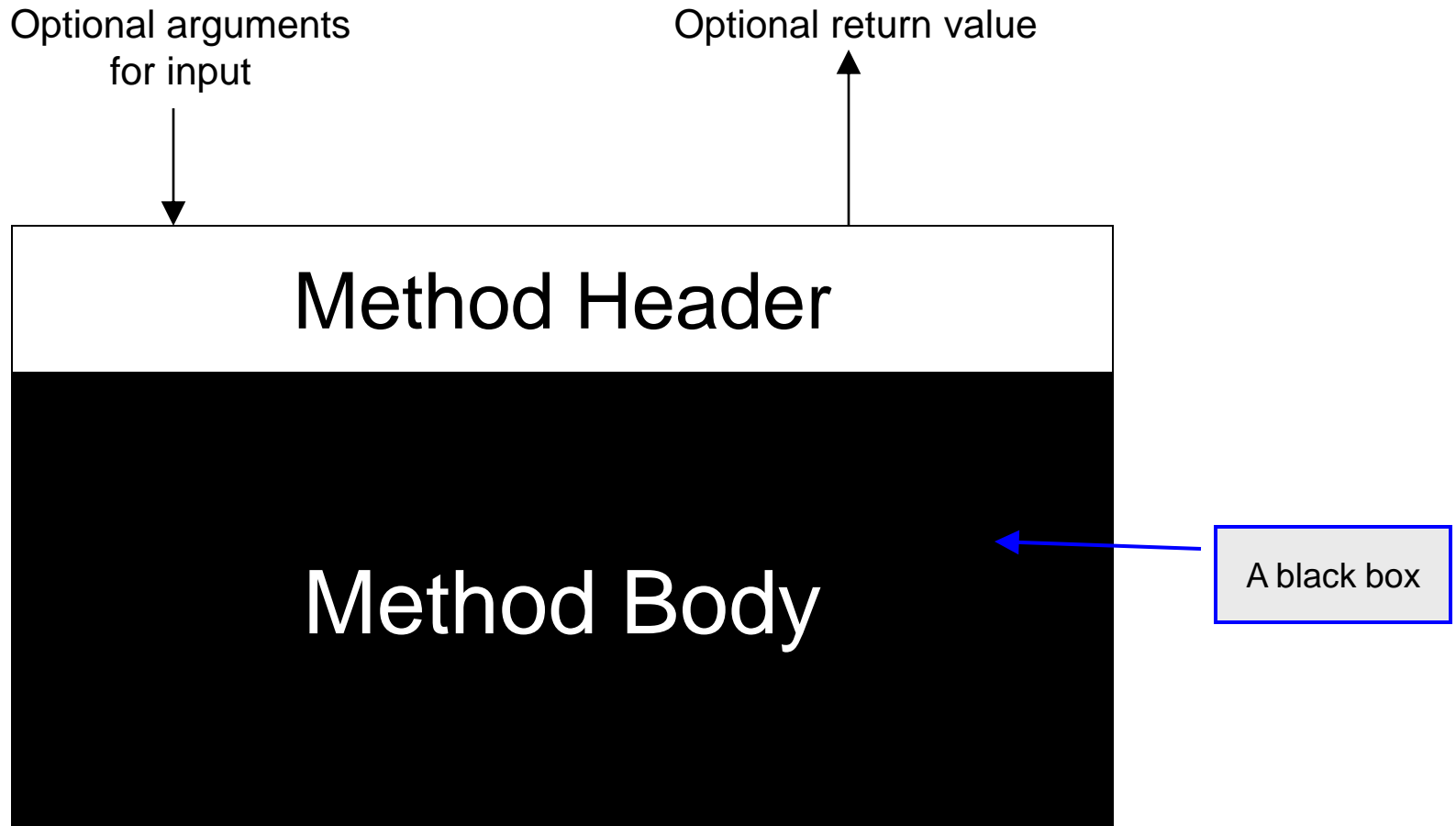
```

<terminated> InputDialogBoxExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 21, 2011 2:55:04 PM)
The value entered in the dialog box was: Today is June 21, 2011

```



Method Abstraction And Stepwise Refinement

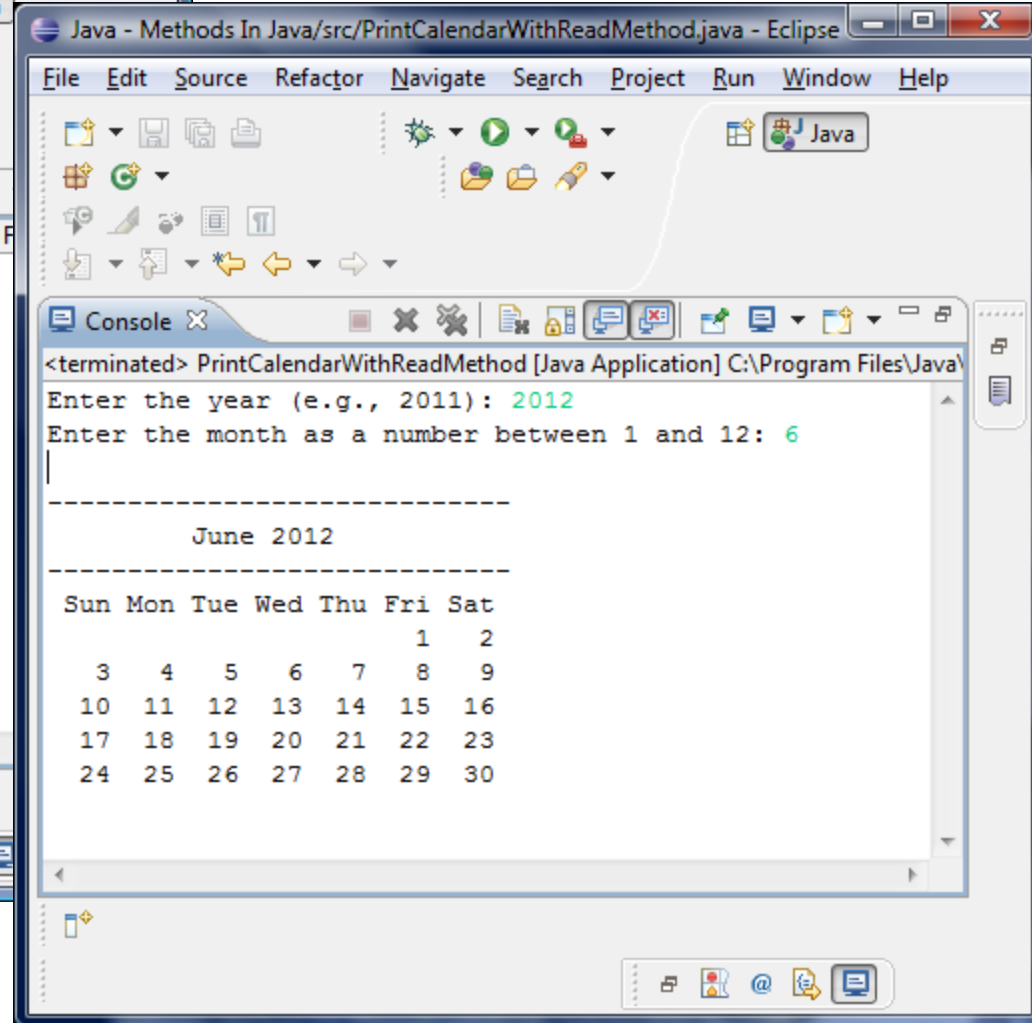
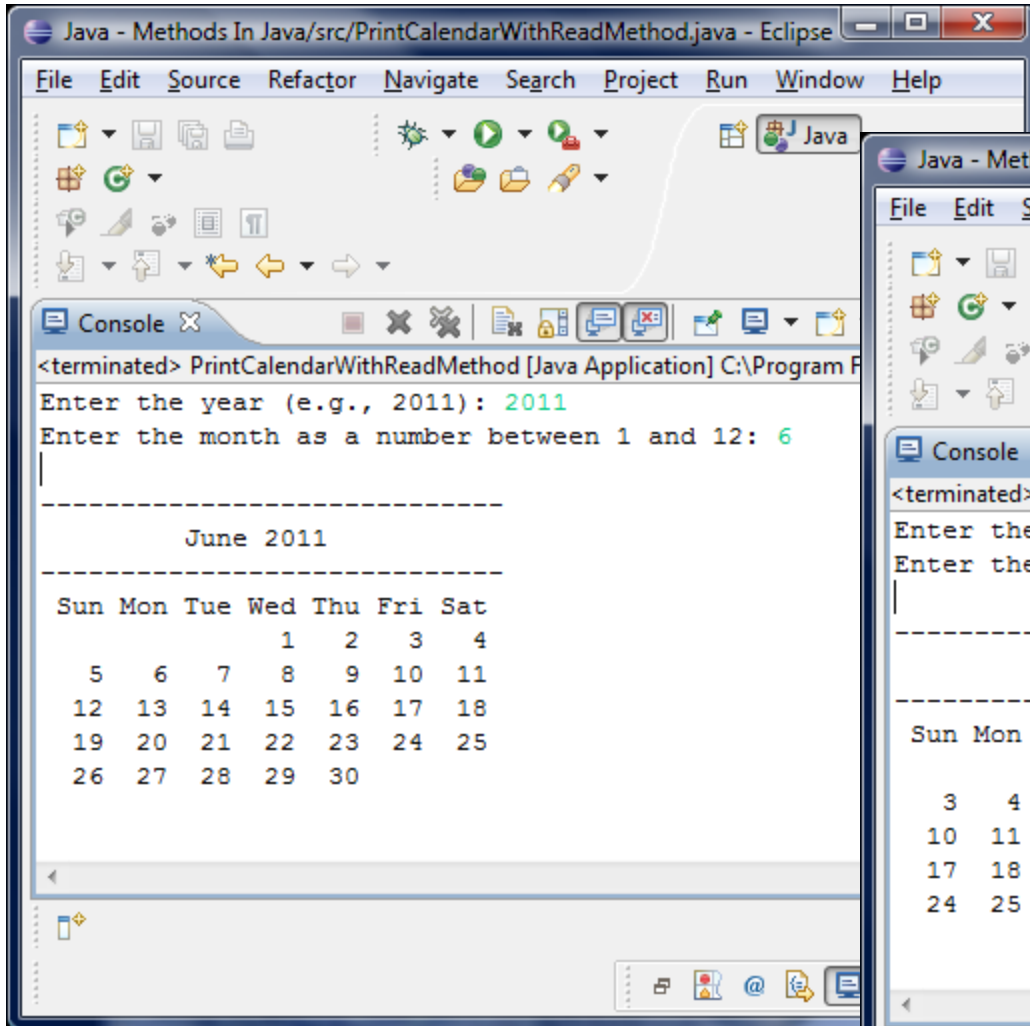


Method Abstraction And Stepwise Refinement

- The concept of method abstraction can also be applied to the process of developing programs.
- When developing a large program, you can use the divide and conquer strategy, also known as **stepwise refinement**, to decompose it into sub-problems. The sub-problems can be further decomposed into smaller, more manageable problems.
- To illustrate this process, we'll develop a program that will display the calendar for a given month of the year. The program will ask the user to enter the year and the month for the calendar they would like to see and then build and display that calendar on the screen. We want the program to work for any year.



Method Abstraction And Stepwise Refinement



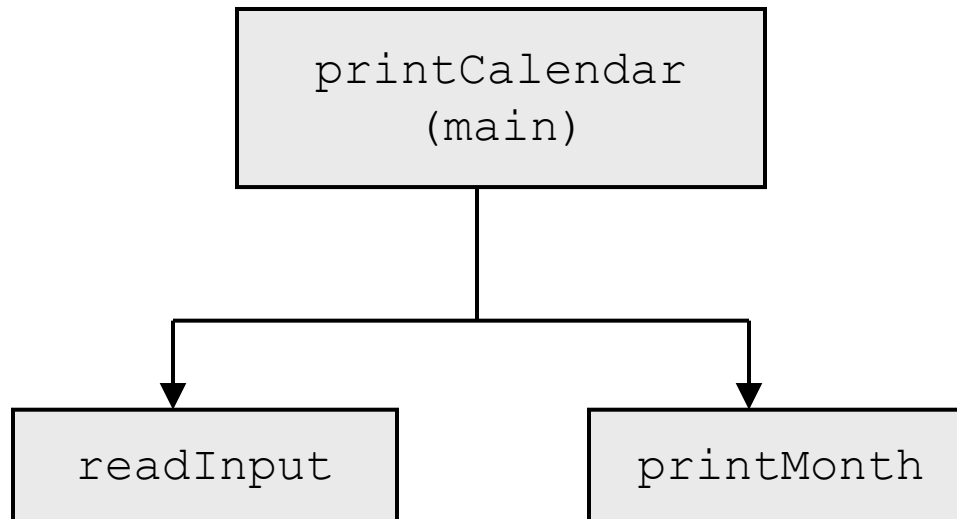
Method Abstraction And Stepwise Refinement

- How would you approach solving this problem? (Please don't say that you would immediately head to the computer and start hammering out a Java program!)
- By trying to work out a solution to every detail of the problem initially, you may actually block or obscure the problem solving process. Solving the problem should be a smooth systematic process and not a hap-hazard detail-oriented approach in which it is actually more likely to overlook a detail than is the case with a more systematic approach.
- The correct approach is to use method abstraction to isolate the details from design and only later implement the details.



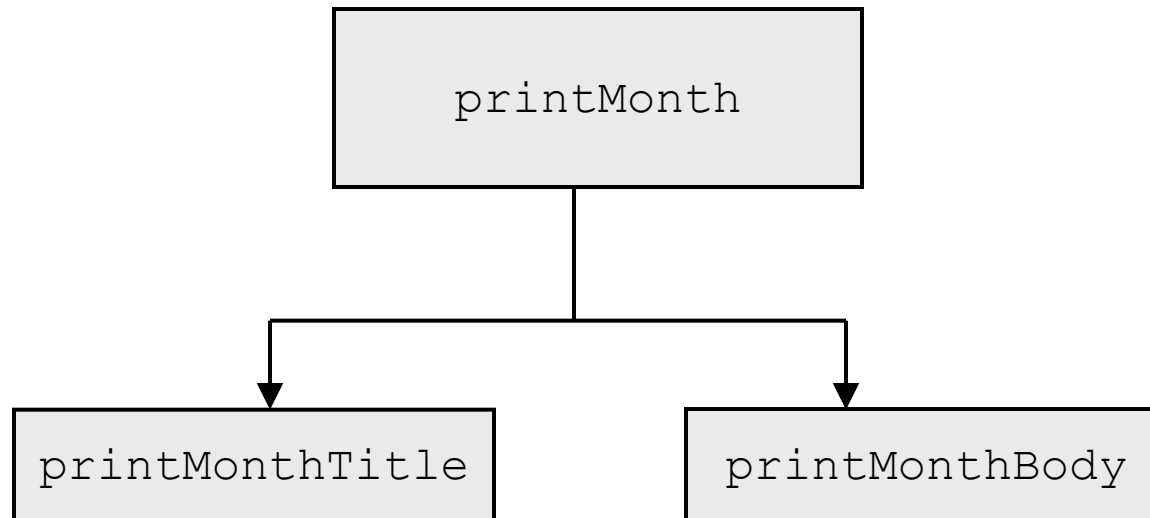
Method Abstraction And Stepwise Refinement

- At the highest level of abstraction in this problem, we'll view the problem of printing the calendar as a problem that contains two sub-problems.
 - Sub-problem 1 is reading the input from the user (`readInput`).
 - Sub-problem 2 is printing the monthly calendar (`printMonth`).

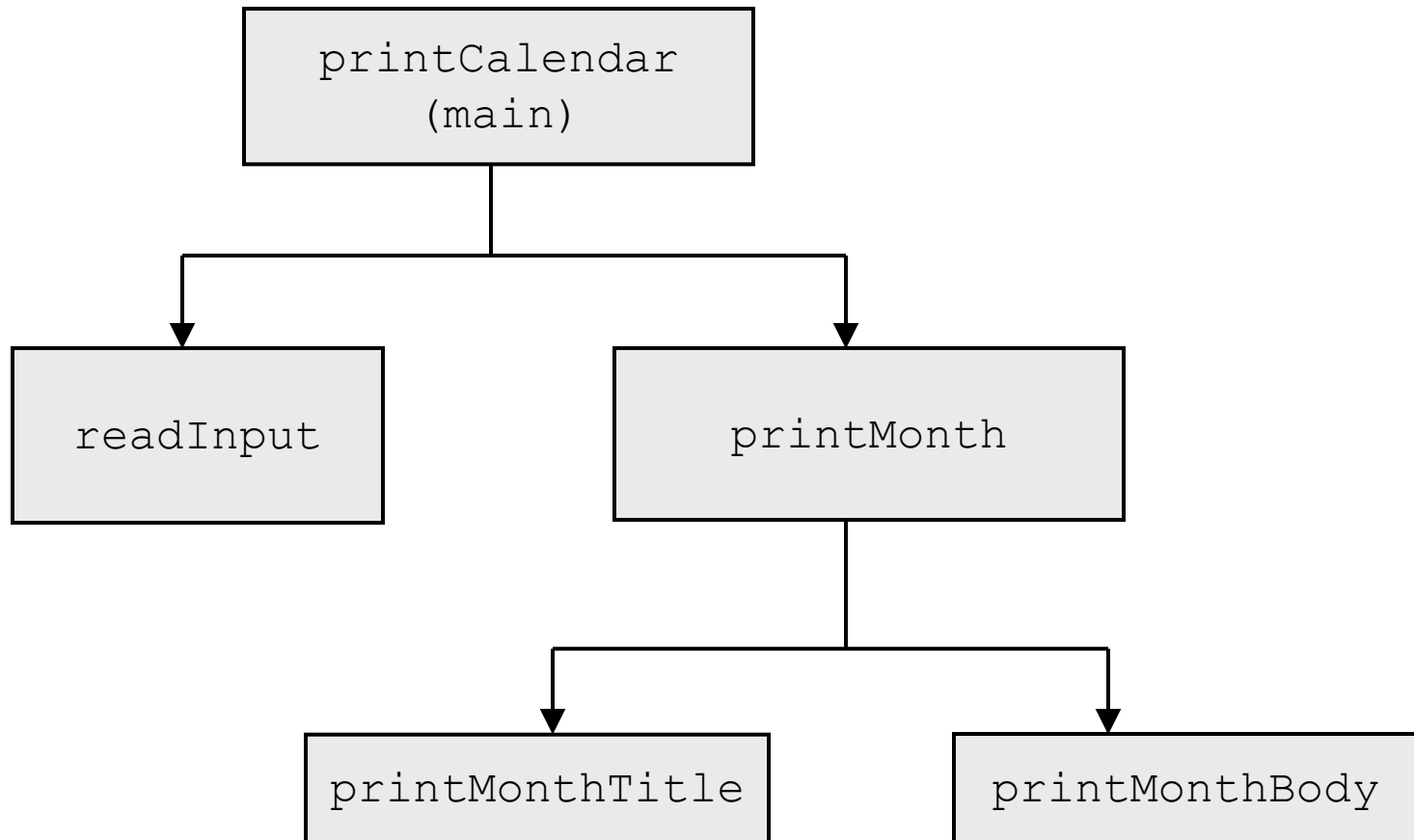


Method Abstraction And Stepwise Refinement

- After thinking about the problem some more, we'll decide that the sub-problem of printing the monthly calendar consists of two sub-problems itself:
 - Sub-problem 1 is printing the monthly title (header part of the calendar) (`printMonthTitle`).
 - Sub-problem 2 is printing the body of the calendar (`printMonthBody`).

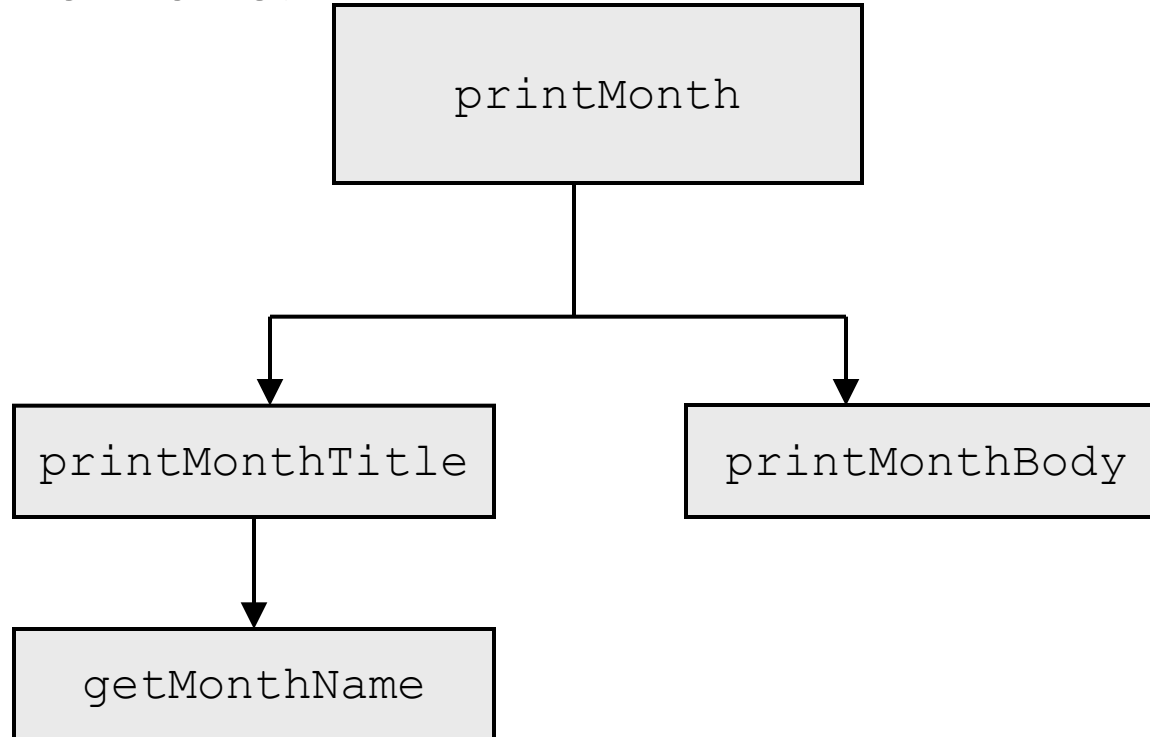


Method Abstraction And Stepwise Refinement



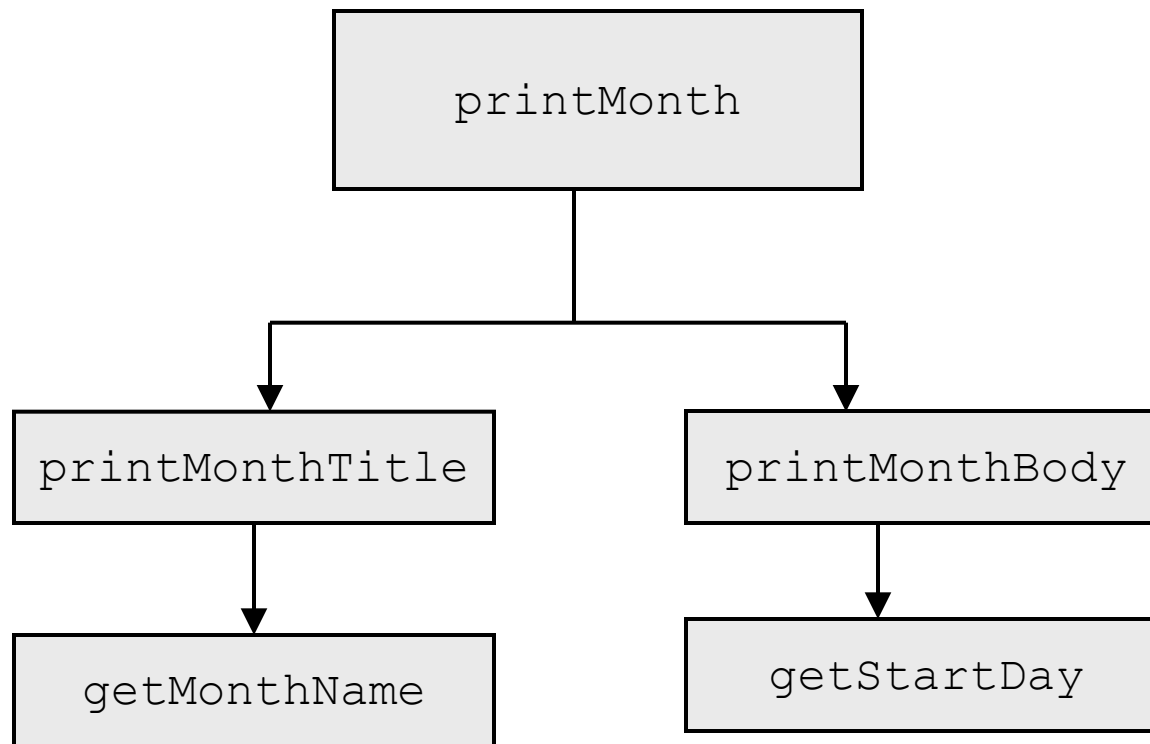
Method Abstraction And Stepwise Refinement

- The `printMonthTitle` sub-problem consists of printing three lines: the month and year on one line, a dashed line, and a line containing the days of the week.
- You need to get the month name from the numeric input supplied by the user. We'll do this with a sub-problem titled `getMonthName`.



Method Abstraction And Stepwise Refinement

- In order to print the month body, we'll need to know which day of the week is the first day of the month. This sub-problem will be called `getStartDay`.

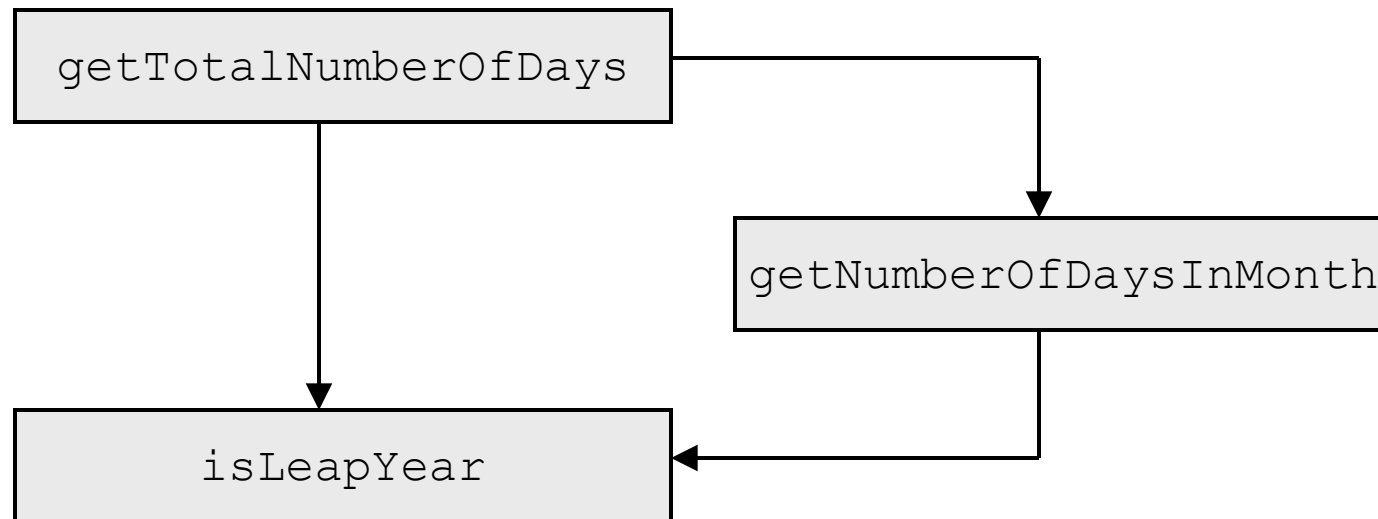
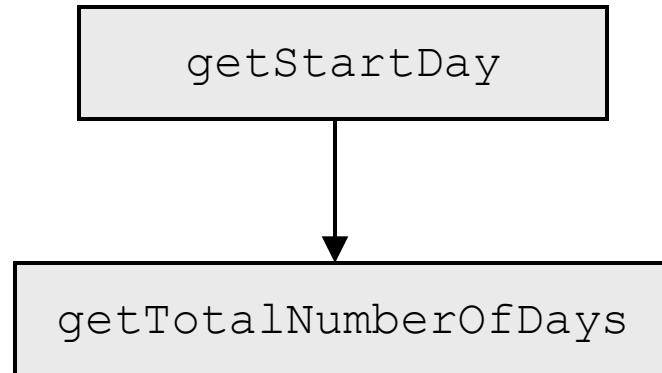


Method Abstraction And Stepwise Refinement

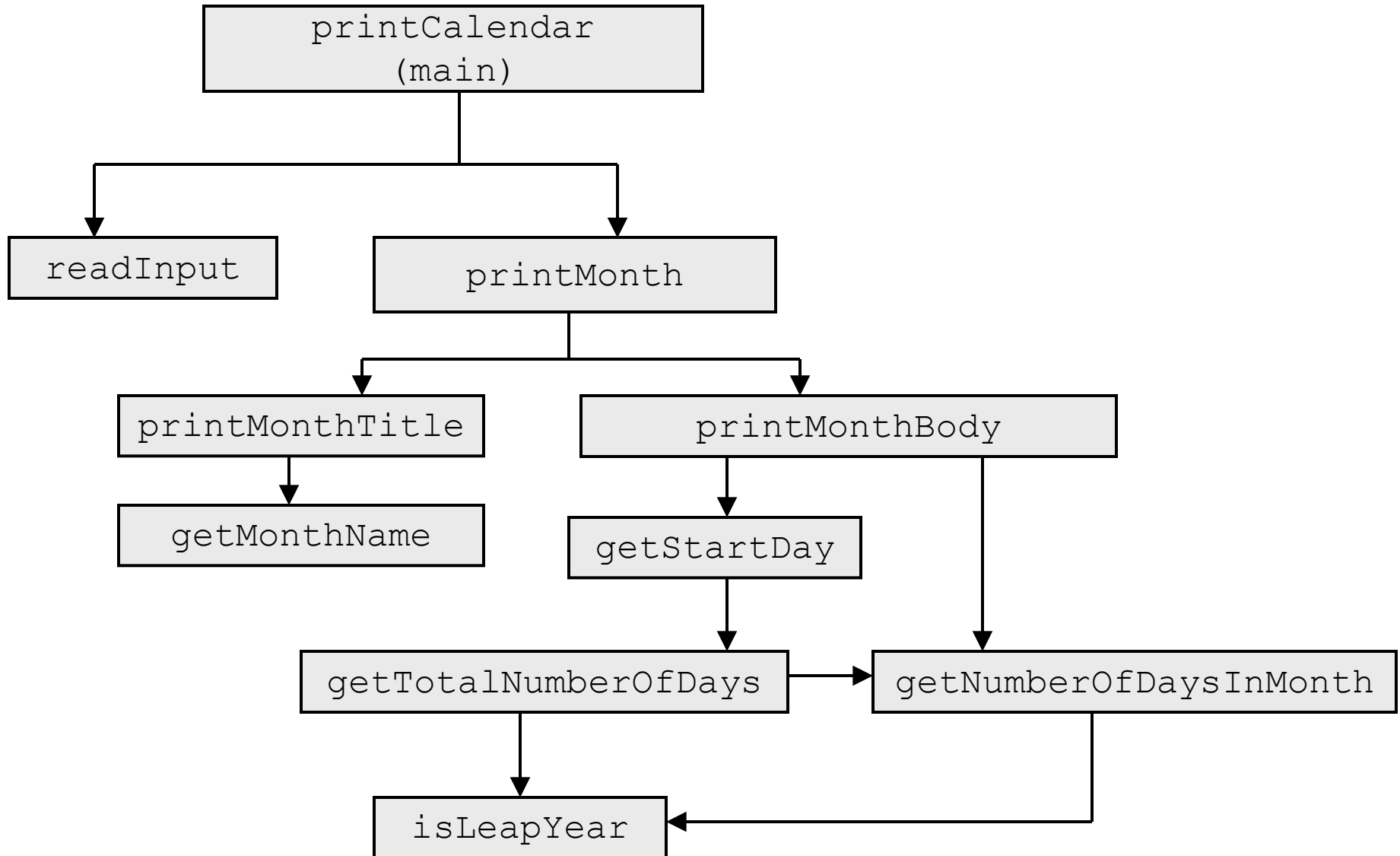
- How do we get the day of the week that begins the month?
- There are several ways to do this. One of the simplest is to use the Calendar class built-in to Java. However, we're going to take a different approach for this problem. Suppose that we know that the day January 1, 1800 was a Wednesday (`startDay1800 = 3`). We can compute the total number of days that have elapsed between January 1, 1800 and the first date of the calendar month in question.
- The start day for our calendar would be $(\text{totalNumberOfDays} + \text{startDay1800}) \% 7$.
- Further, we need to figure in leap years (`isLeapYear`).
- So we need to refine our sub-problem structure a bit more.



Method Abstraction And Stepwise Refinement



Method Abstraction And Stepwise Refinement



Top-Down or Bottom-Up Implementation

- Now that we have designed our solution to the problem, its time to begin implementation.
- In general, a sub-problem in our design will correspond to a method in the implementation. Although, some sub-problems may be so simple that a separate method is not warranted and it may be combined in another method.
 - Decisions of this sort should be based on whether the overall program will be easier to read if the sub-problem remains as a separate method or is incorporated into another sub-problem's method implementation.
 - For example, in this problem, we might justifiably implement the `readInput` sub-problem in `main` rather than create a separate method.



Top-Down or Bottom-Up Implementation

- We can use either a top-down or bottom-up approach to implementation.
- A **top-down** approach implements one method in the structure diagram at a time from the top to the bottom of the diagram.
 - Working from more general sub-problems toward more specific sub-problems.
- A **bottom-up** approach implements one method in the structure diagram at a time from the bottom of the diagram to the top.
 - Working from more specific sub-problems toward more general sub-problems.



Top-Down or Bottom-Up Implementation

- With either approach, a common technique for implementation is to create stubs for each sub-problem in the structure diagram.
- A **stub** (or **stub method**) is a simple, working but incomplete version of a method.
- The use of stubs allows you to quickly build the framework of the program, which is filled in as you complete either a top-down or bottom-up approach.
- In our example, using a top-down approach, we would implement `main` first, followed by a stub for `printMonth`.



```
/* PrintCalendarStubVersion class - illustrates stepwise-refinement for
 * application development. This class contains only stub methods
 * Used in Methods in Java class notes - Summer 2011
 *
 * Mark Llewellyn - 6/21/2011
 * Not completed version - stub methods only
 */
import java.util.Scanner;

public class PrintCalendarStubVersion {
    /** Main method */
    public static void main(String[] args) {
        // Prompt the user to enter year
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter year
        System.out.print("Enter the year (e.g., 2011): ");
        int year = input.nextInt();

        // Prompt the user to enter month
        System.out.print("Enter the month as a number between 1 and 12: ");
        int month = input.nextInt();

        // Print calendar for the month of the year
        PrintCalendarStubVersion.printMonth(year, month);
    } //end main method
```



```
/** Print the calendar for a month in a year */
static void printMonth(int year, int month) {
    System.out.println(month + " " + year);
} //end printMonth method

/** Print the month title, e.g., June, 2011 */
static void printMonthTitle(int year, int month) {
    System.out.println("Print Month Title" + month + " " + year);
} //end printMonthTitle method

/** return the name of the desired month */
static String getMonthName(int month) {
    return("January"); //dummy value
} //end getMonthName method

/** print the body of the month */
static void printMonthBody(int year, int month) {
    System.out.println("In printMonthBody "+ month + " " + year);
} //end printMonthBody method

/** return the starting day of the month */
static int getStartDay(int year, int month) {
    return (1); //dummy value
} //end getStartDay method
```



```
/** return the total number of elapsed days */
static int getTotalNumberOfDays(int year, int month) {
    return (10000); //dummy value
} // end getTotalNumberOfDays method

/** Get the number of days in a month */
static int getNumberOfDaysInMonth(int year, int month) {
    return (31);
} //end getNumberOfDaysInMonth method

/** Determine if it is a leap year */
static boolean isLeapYear(int year) {
    return (true);
} //end isLeapYear method

} //end PrintCalendarStubVersion class
```



Implementation Details

- Continuing with our example or printing a monthly calendar based on the month and year supplied by the user, we now will focus on implementation issues.
- So far we've only completed the structural design of our solution and created a Java class that implemented stubs for each of the methods we defined in our solution. The desired output and the structural design chart are repeated on the next two pages.



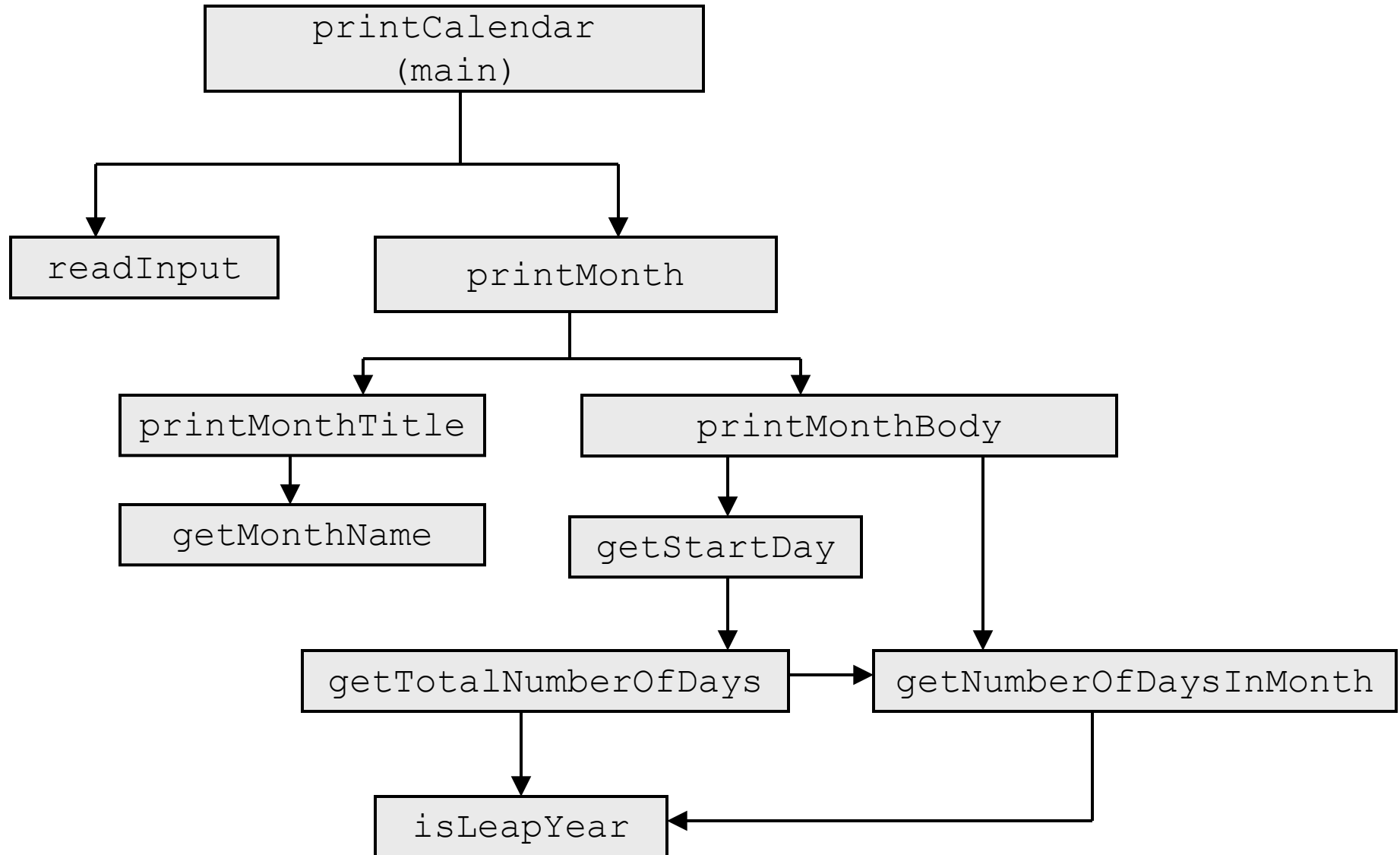
Output from the calendar program

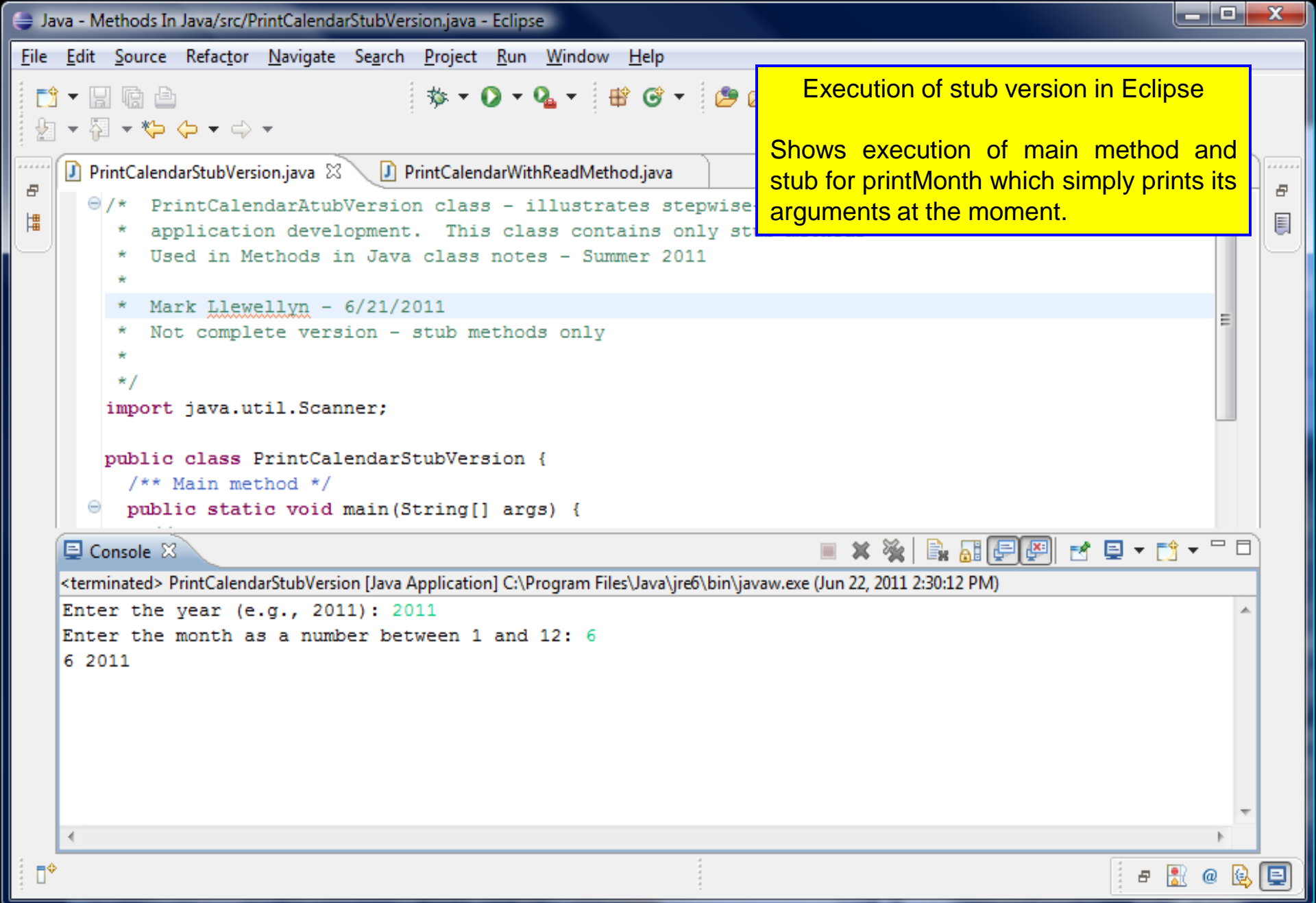
```
Console X
<terminated> PrintCalendarWithReadMethod [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun
Enter the year (e.g., 2011): 2011
Enter the month as a number between 1 and 12: 6

-----
                June 2011
-----

Sun Mon Tue Wed Thu Fri Sat
      1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```







Execution of stub version in Eclipse

Shows execution of main method and stub for printMonth which simply prints its arguments at the moment.

```
/* PrintCalendarStubVersion class - illustrates stepwise
 * application development. This class contains only stub
 * Used in Methods in Java class notes - Summer 2011
 *
 * Mark Llewellyn - 6/21/2011
 * Not complete version - stub methods only
 */
import java.util.Scanner;

public class PrintCalendarStubVersion {
    /** Main method */
    public static void main(String[] args) {
```

```
<terminated> PrintCalendarStubVersion [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 22, 2011 2:30:12 PM)
Enter the year (e.g., 2011): 2011
Enter the month as a number between 1 and 12: 6
6 2011
```



Implementation Details

- As we develop the method bodies for the various methods as defined in our structure chart, we'll have some specific implementation details that must be dealt with at the time.
- However, our abstraction gives us the luxury of only needing to deal with those details when we are developing the code for that specific method. When dealing with other methods, those details specific to other methods are not important – we've abstracted away those details.
- For example, when we need to develop the `isLeapYear(int year)` method, we'll need to know that leap years are years that are evenly divisible by 4. Years that are evenly divisible by 100 are not leap years unless they are also evenly divisible by 400 (in the Gregorian calendar at least).

So we could say:

```
if (year modulo 4 is 0) and (year modulo 100 is not 0)
    or (year modulo 400 is 0) then leap else no_leap
```

In Java we would express this condition as:

```
return(year % 400 == 0 || (year % 4 == 0 && year % 100 != 0));
```



Implementation Details

- Similarly, to implement the method:

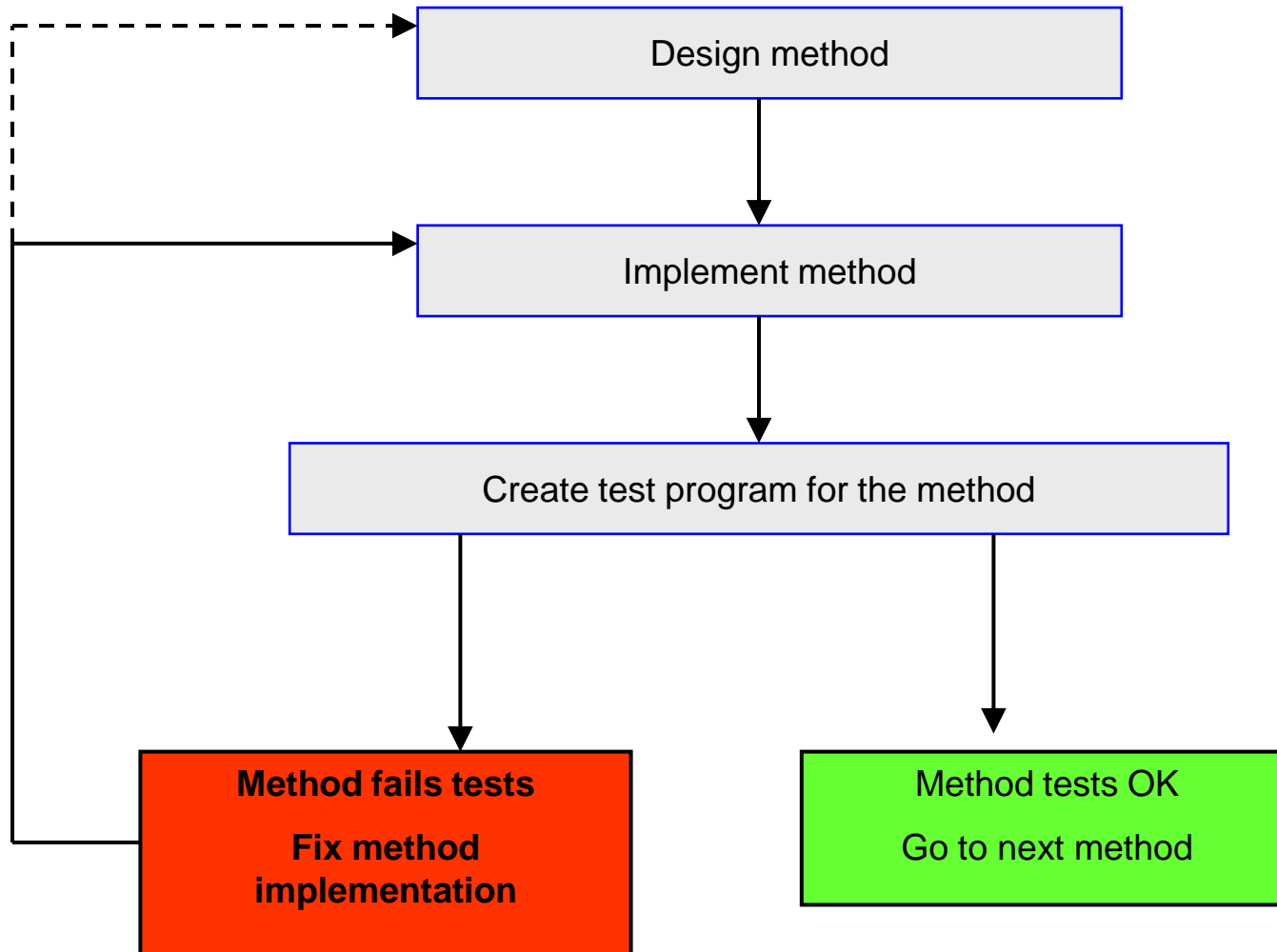
`getTotalNumberOfDaysInMonth(int year, int month)`, we'll need to know that January, March, May, July, August, October, and December each have 31 days. April, June, September, and November each have 30 days. February has 28 days in a common year and 29 days in a leap year. So a regular year has 365 days and a leap year has 366 days.

- To implement

`getTotalNumberOfDays(int year, int month)`, we'll get sum the total number of days between January 1, 1800 and the first day of the month of the calendar. To do this we can calculate the total number of days from January 1, 1800 to January 1 of the year in question and then add the remaining days in the calendar year to the first of the month in question.



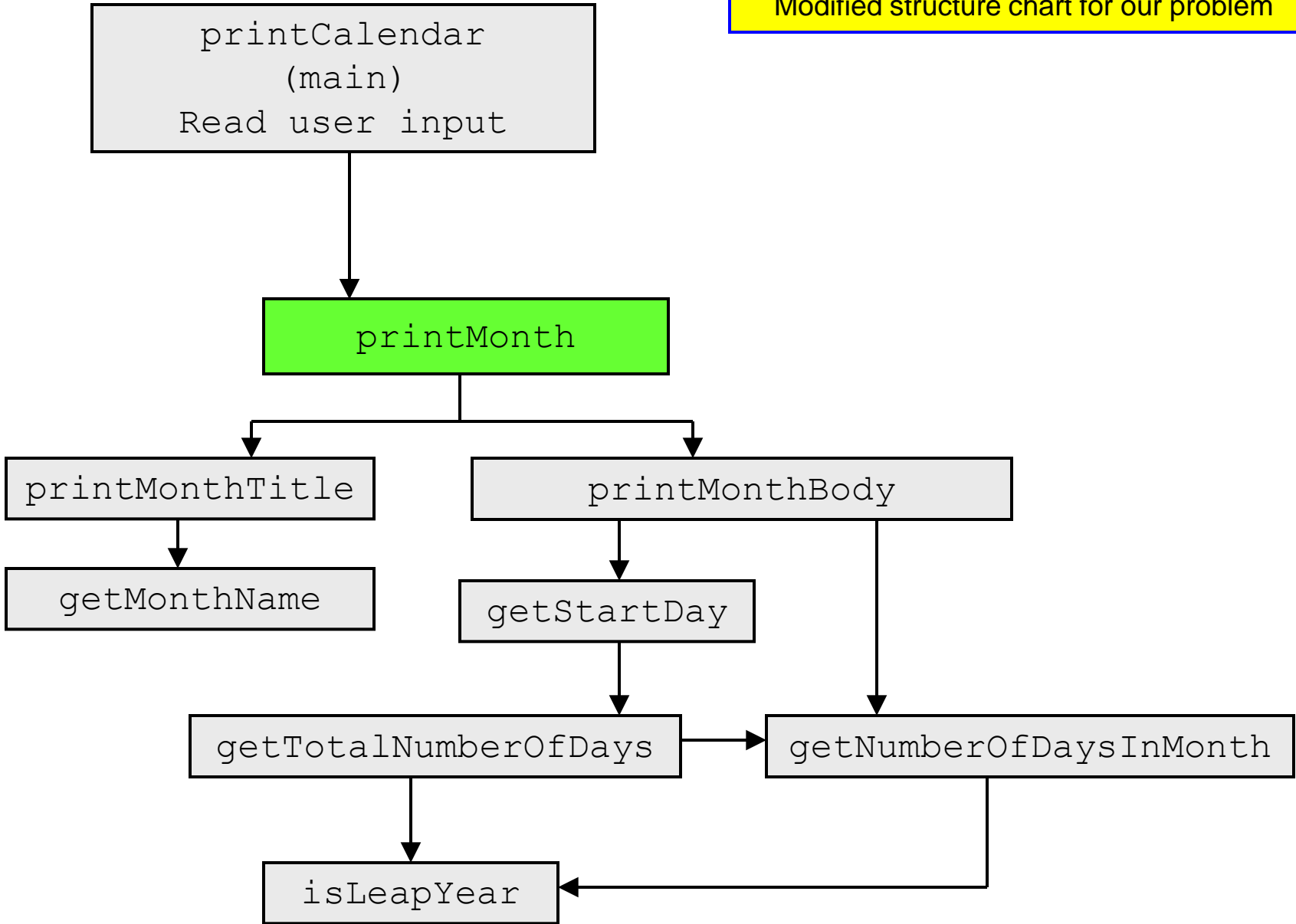
Implementation Details



Implementation Details

- We'll continue our development of this program using a top-down approach. This means that we would first implement the main method (the one on the top of our structure chart).
- The main method, based on our structure chart is based on two sub-problems, `readInput` and `printMonth`.
- We'll decide here, for the sake of readability that the statements that would comprise the `readInput` method, which involve reading the year and month for which the user would like the calendar, will simply be coded as statements in the `main` method. The alternative is to create a method for this, however, in this case, it is relatively simple input and it would seem unnecessary to place this code in a method, so we'll simply move those tasks to the `main` method.
 - Note that if we did implement a `readInput` method that `main` would consist of only two lines of code, the first to invoke the `readInput` method and the second would be to invoke the `printMonth` method.





Implementation Details

- Next, we'll focus on the `printMonth` method. As you can see from the structure chart on the previous page, this sub-problem consists of two sub-problems: (1) `printMonthTitle` and (2) `printMonthBody`. Thus, the implementation of this method will consist of two method invocations for the respective sub-problems.



Implementation Details

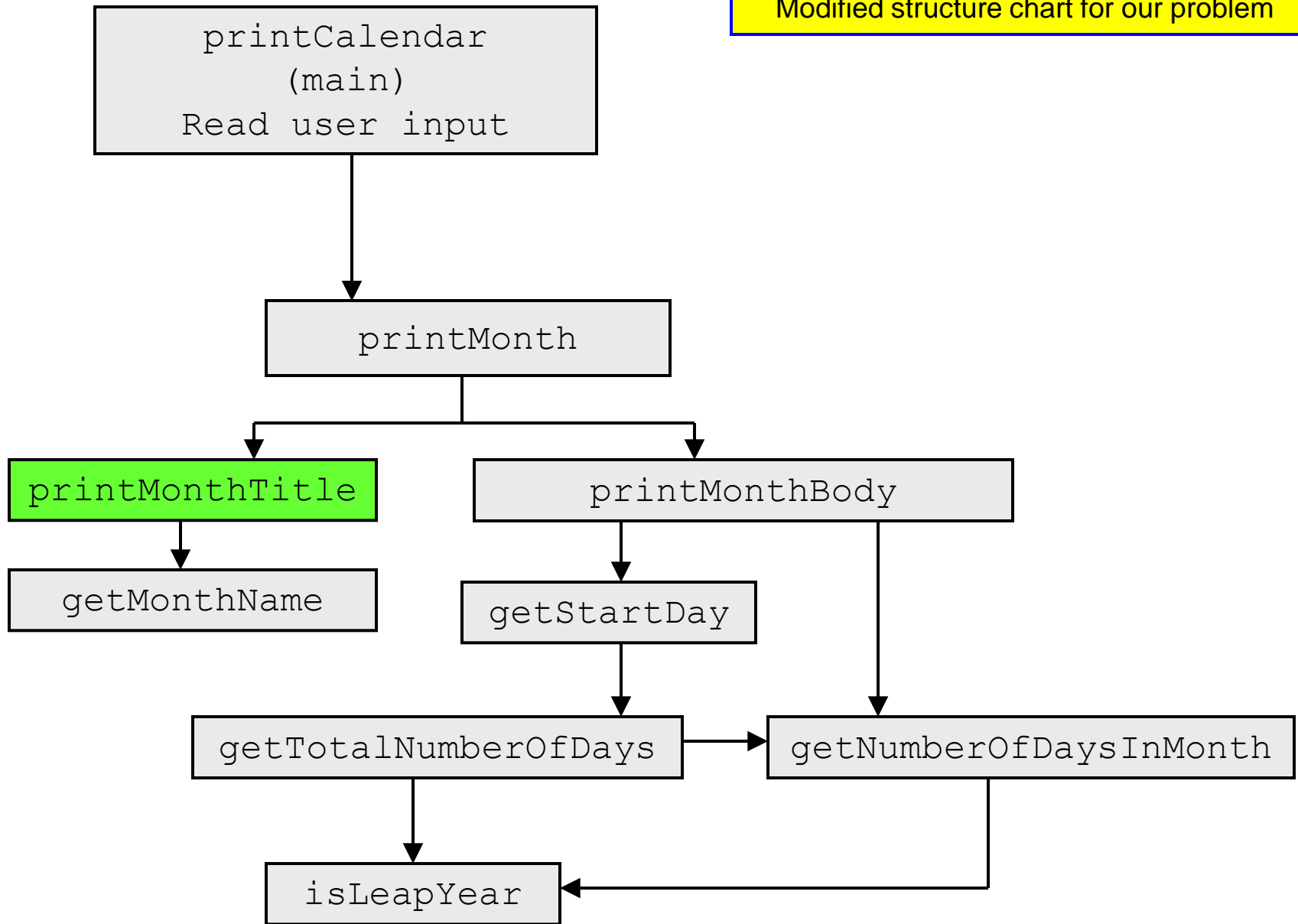
Stub version

```
/** stub for printMonth */  
public static void printMonth(int year, int month) {  
    System.out.print(month + " " + year);  
} //end printMonth method
```

Complete version

```
/** Print the calendar for a month in a year */  
static void printMonth(int year, int month) {  
    // Print the headings of the calendar  
    printMonthTitle(year, month);  
  
    // Print the body of the calendar  
    printMonthBody(year, month);  
}
```





Implementation Details

- Using the top-down approach, we'll work with the `printMonthTitle` sub-problem first. This task consists of building the header information for the calendar in question. So this method will need to know the year and month from the input, so we set both as arguments to the method. The rest is handled with print statements.



Implementation Details

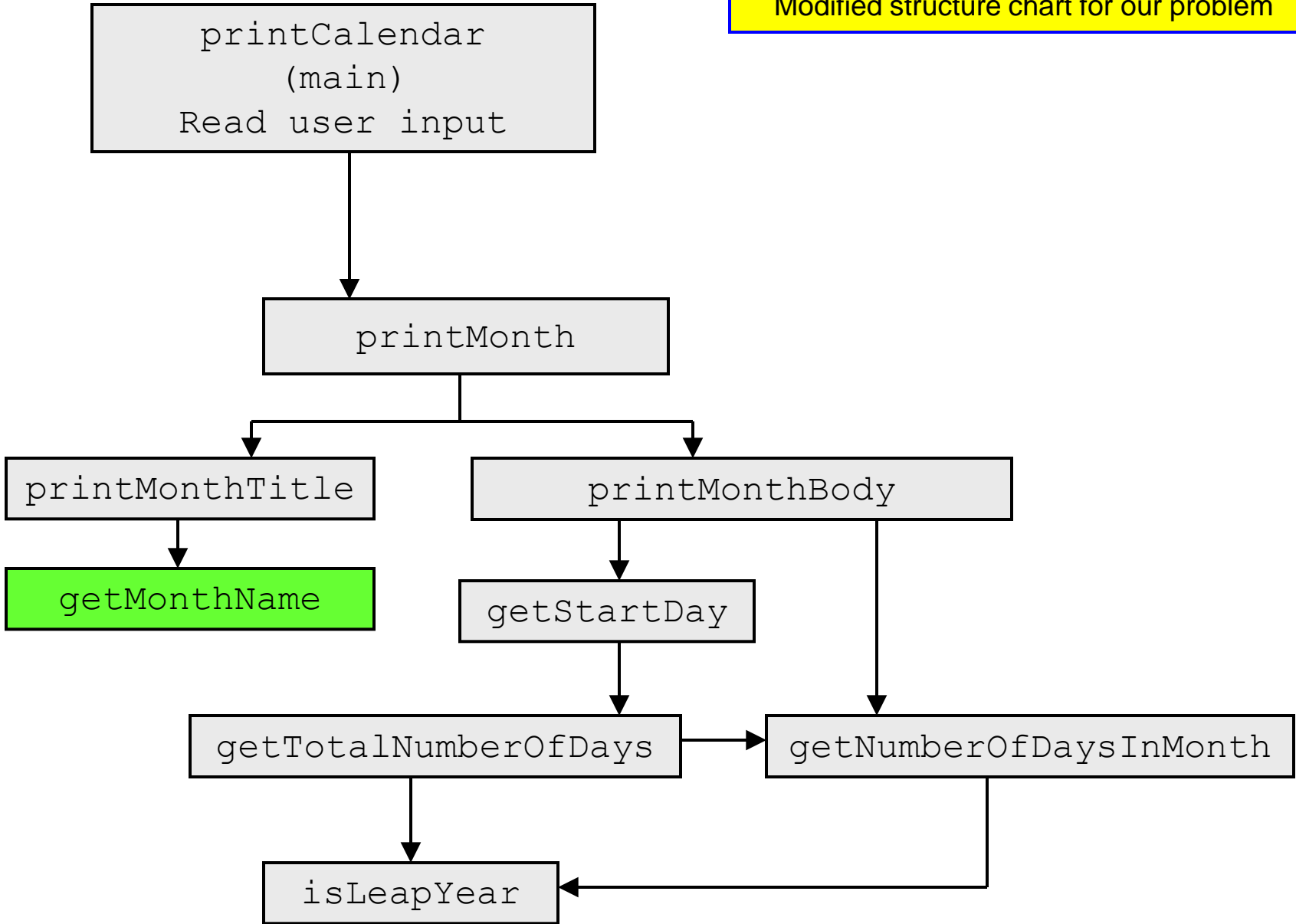
Stub version

```
/** stub for printMonthTitle */  
public static void printMonthTitle(int year, int month) {  
} //end printMonthTitle method  
/** stub for printMonthBody */
```

Complete version

```
/** Print the month title, e.g., June, 2011 */  
static void printMonthTitle(int year, int month) {  
    System.out.println();  
    System.out.println("-----");  
    System.out.println("          " + getMonthName(month)  
        + " " + year);  
    System.out.println("-----");  
    System.out.println(" Sun Mon Tue Wed Thu Fri Sat");  
}
```





Implementation Details

Stub version

```
/** stub for getMonthName */  
public static String getMonthName(int month) {  
    return ("January"); //dummy value  
} //end getMonthName method
```

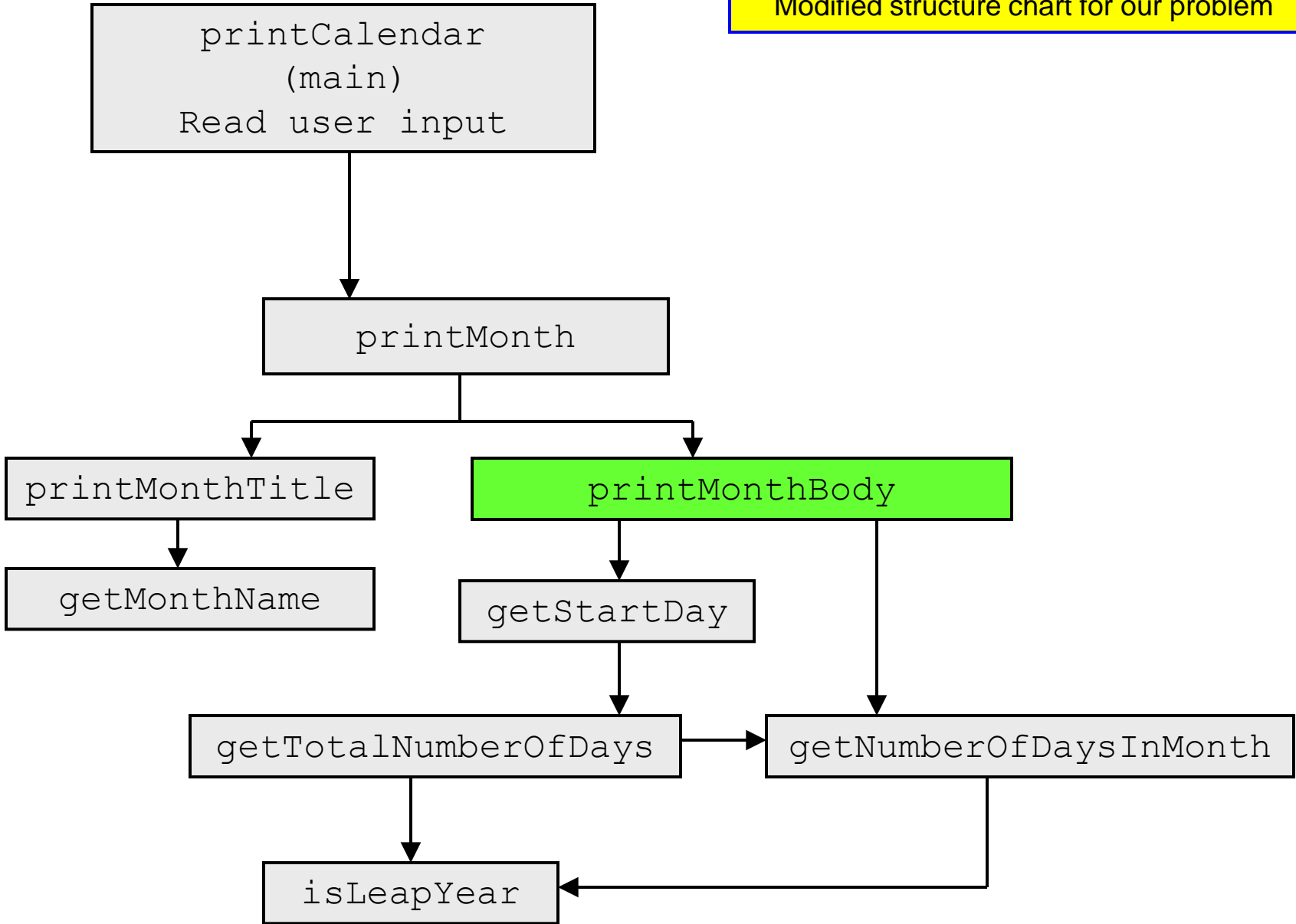


Implementation Details

Complete version

```
/** Get the English name for the month */
static String getMonthName(int month) {
    String monthName = null;
    switch (month) {
        case 1: monthName = "January"; break;
        case 2: monthName = "February"; break;
        case 3: monthName = "March"; break;
        case 4: monthName = "April"; break;
        case 5: monthName = "May"; break;
        case 6: monthName = "June"; break;
        case 7: monthName = "July"; break;
        case 8: monthName = "August"; break;
        case 9: monthName = "September"; break;
        case 10: monthName = "October"; break;
        case 11: monthName = "November"; break;
        case 12: monthName = "December";
    }
    return monthName;
}
```





Implementation Details

Stub version

```
public static void printMonthBody(int year, int month) {  
  }//end printMonthBody method
```



Implementation Details

Complete version

```
/** Print month body */
static void printMonthBody(int year, int month) {
    // Get start day of the week for the first date in the month
    int startDay = getStartDay(year, month);

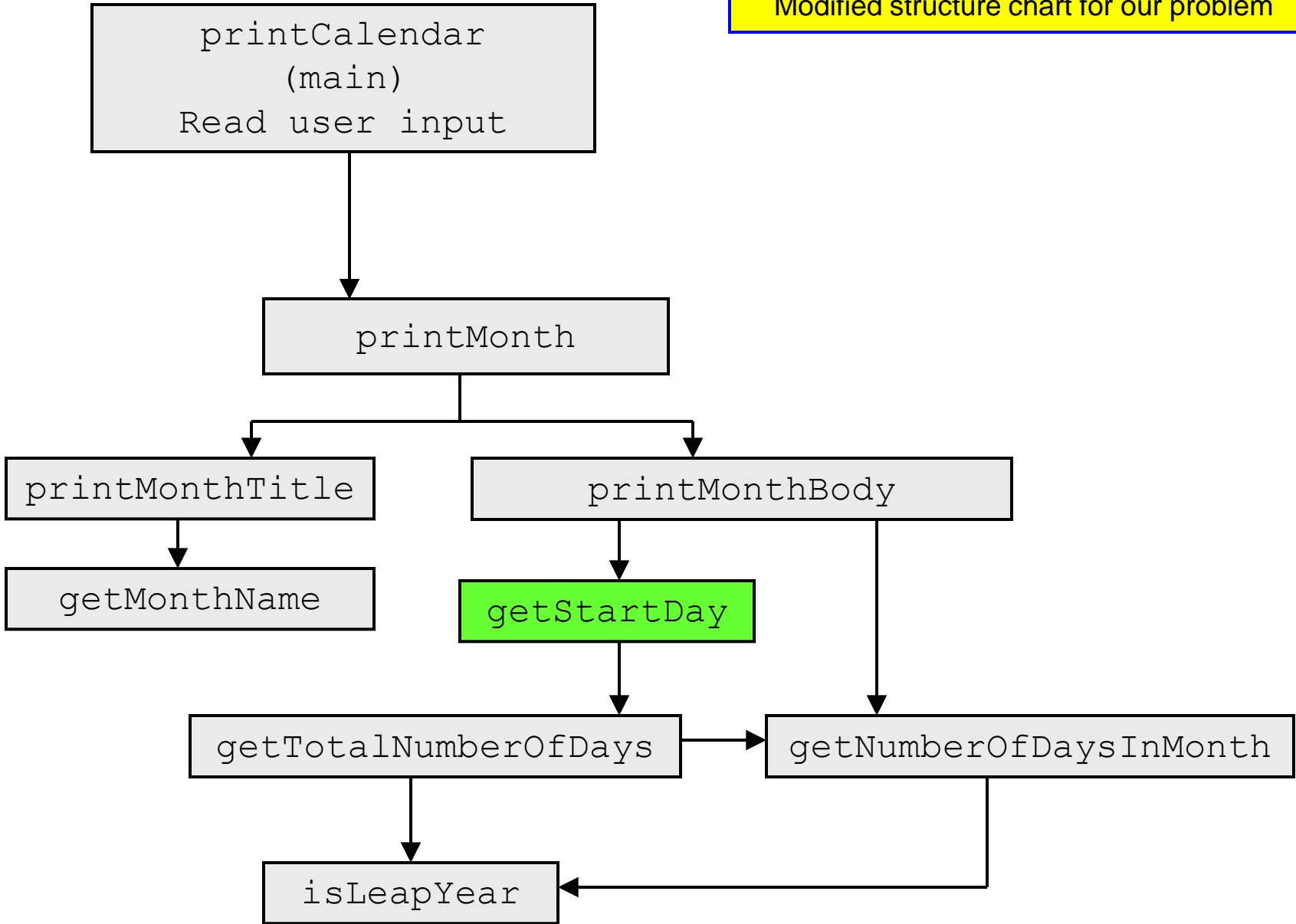
    // Get number of days in the month
    int numberOfDaysInMonth = getNumberOfDaysInMonth(year, month);

    // Pad space before the first day of the month
    int i = 0;
    for (i = 0; i < startDay; i++)
        System.out.print("  ");

    for (i = 1; i <= numberOfDaysInMonth; i++) {
        if (i < 10)
            System.out.print("  " + i);
        else
            System.out.print(" " + i);

        if ((i + startDay) % 7 == 0)
            System.out.println();
    }
    System.out.println();
}
```





Implementation Details

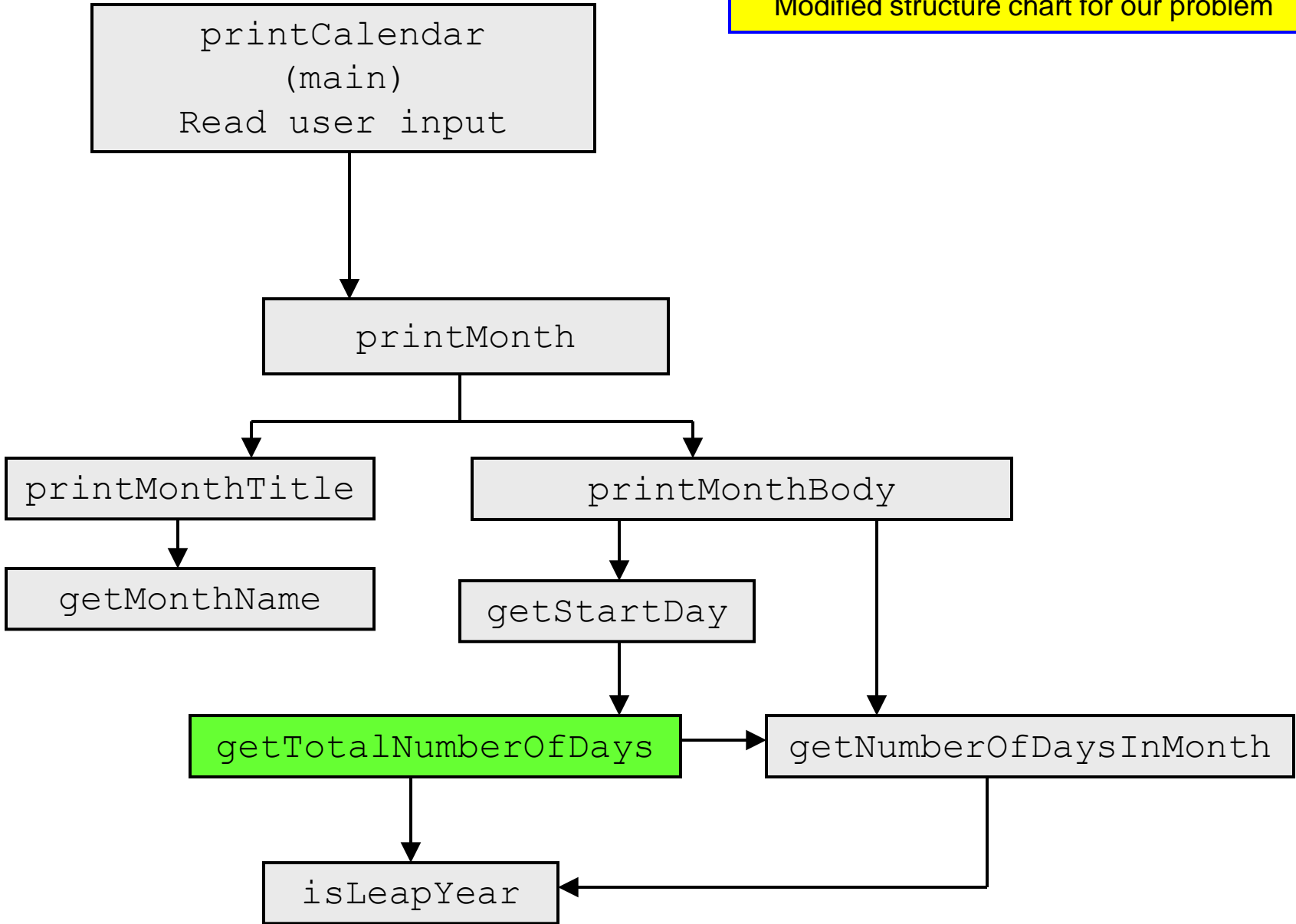
Stub version

```
/** stub for getStartDay */  
public static int getStartDay(int year, int month) {  
    return 1; //dummy value  
} //end getStartDay method
```

Complete version

```
/** Get the start day of month/1/year */  
static int getStartDay(int year, int month) {  
    final int START_DAY_FOR_JAN_1_1800 = 3;  
    // Get total number of days from 1/1/1800 to month/1/year  
    int totalNumberOfDays = getTotalNumberOfDays(year, month);  
  
    // Return the start day for month/1/year  
    return (totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7;  
}
```





Implementation Details

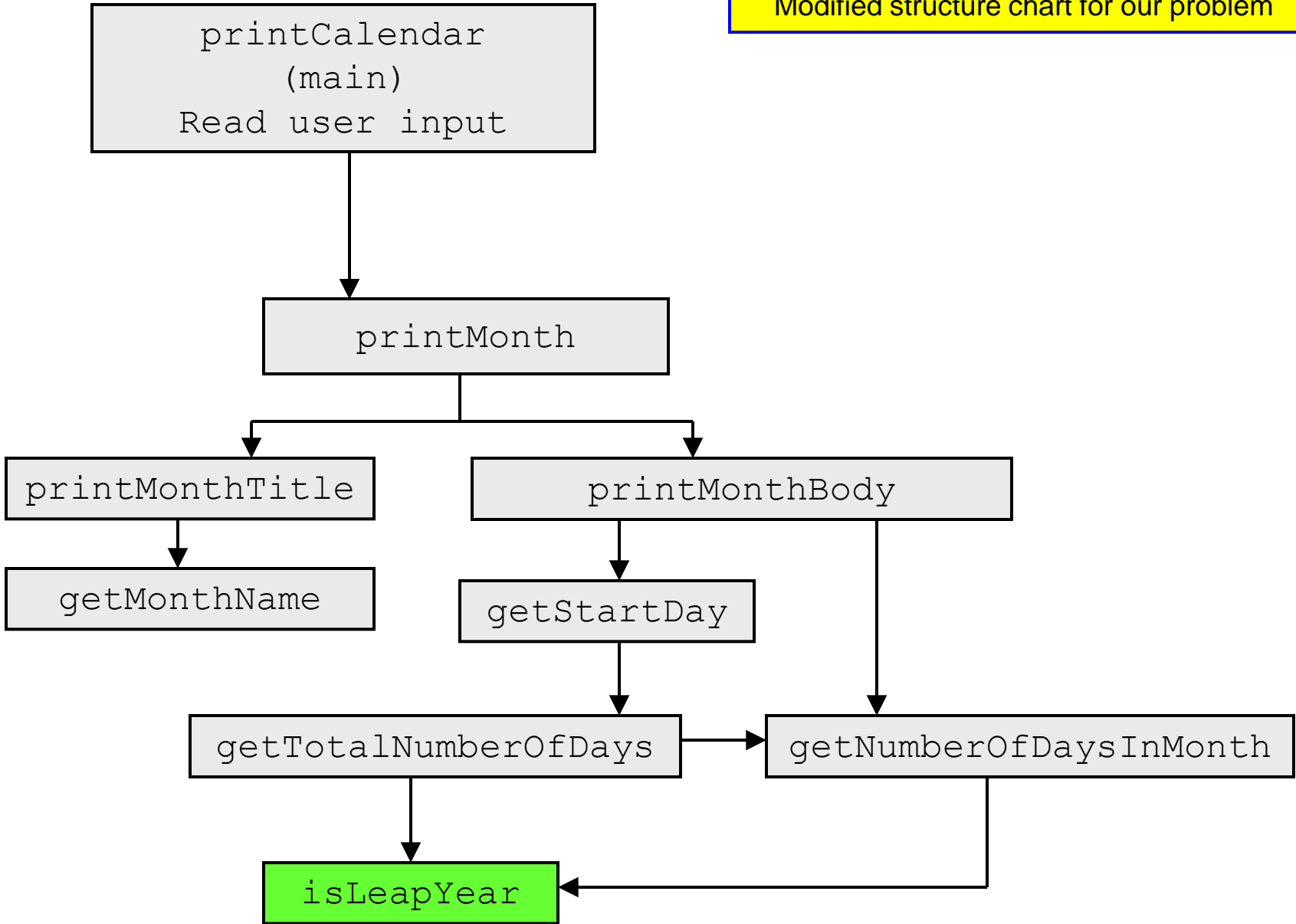
Stub version

```
/** stub for getTotalNumberOfDays */  
public static int getTotalNumberOfDays(int year, int month) {  
    return 10000; //dummy value  
} //end getTotalNumberOfDays
```

Complete version

```
/** Get the total number of days since January 1, 1800 */  
static int getTotalNumberOfDays(int year, int month) {  
    int total = 0;  
    // Get the total days from 1800 to 1/1/year  
    for (int i = 1800; i < year; i++)  
        if (isLeapYear(i))  
            total = total + 366;  
        else  
            total = total + 365;  
    // Add days from Jan to the month prior to the calendar month  
    for (int i = 1; i < month; i++)  
        total = total + getNumberOfDaysInMonth(year, i);  
    return total;  
}
```





Implementation Details

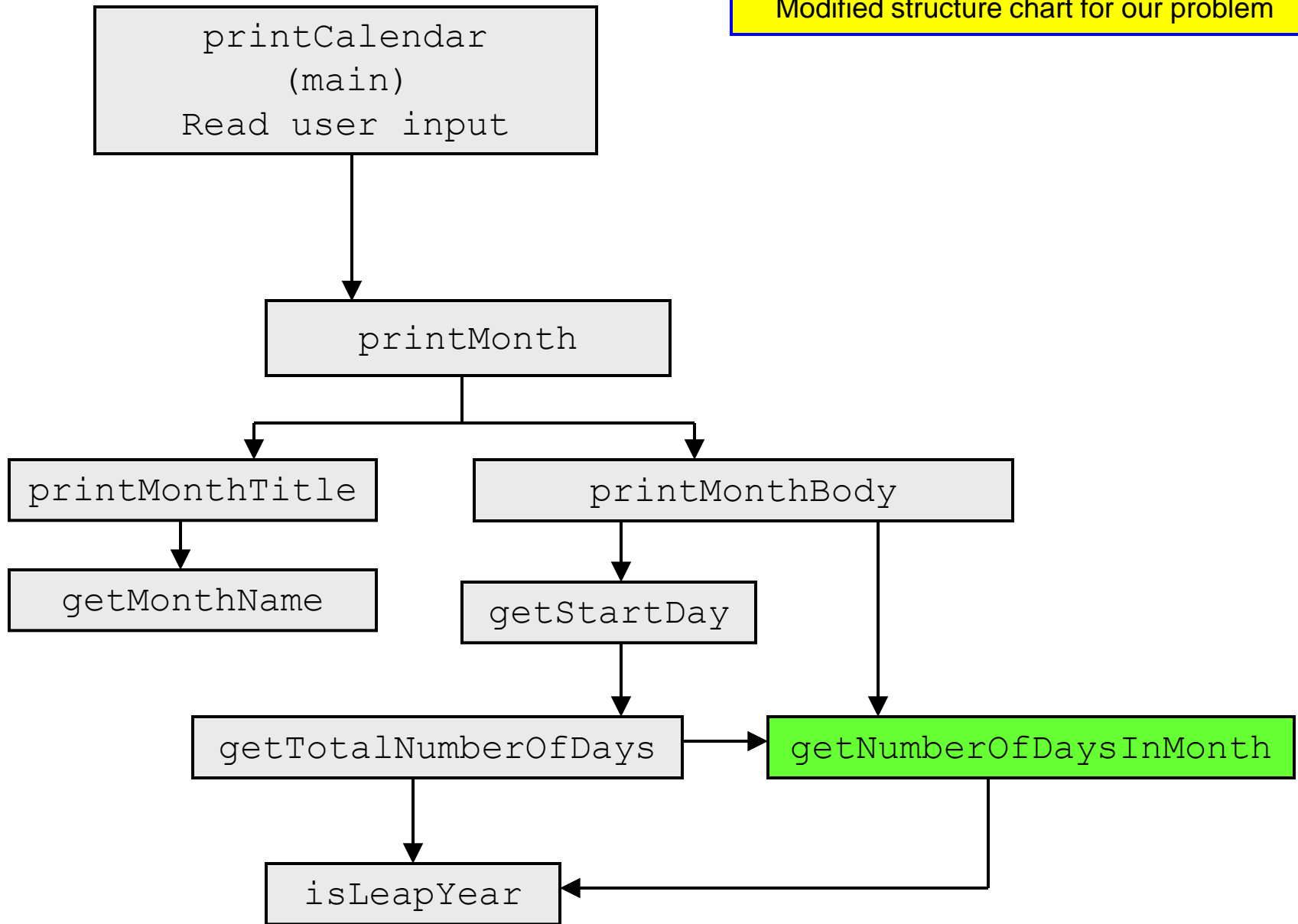
Stub version

```
/** stub for isLeapyear */  
public static boolean isLeapYear(int year) {  
  return true; //dummy value  
} //end isLeapYear method}
```

Complete version

```
/** Determine if it is a leap year */  
static boolean isLeapYear(int year) {  
  return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);  
}
```





Implementation Details

Stub version

```
/** stub for getNumberOfDaysInMonth */  
public static int getNumberOfDaysInMonth(int year, int month) {  
    return 31; //dummy value  
} //end getNumberOfDaysInMonth method
```

Complete version

```
/** Get the number of days in a month */  
static int getNumberOfDaysInMonth(int year, int month) {  
    if (month == 1 || month == 3 || month == 5 || month == 7 ||  
        month == 8 || month == 10 || month == 12)  
        return 31;  
  
    if (month == 4 || month == 6 || month == 9 || month == 11)  
        return 30;  
  
    if (month == 2) return isLeapYear(year) ? 29 : 28;  
  
    return 0; // If month is incorrect  
}
```



Implementation Details

- We're done!
- All of the subprograms have been implemented and tested as we went from top to bottom in our structure chart.
- A complete program listing for this problem is available on the course website.



Practice Problem

1. Rewrite the implementation of the `PrintCalendar` application so that instead of having the main method handle the user input, it is done via methods as per our original design diagram. In this case, the main method will do nothing except invoke the methods for reading the input and printing the calendar.

Note that there are several different ways in which this can be done. I'll put a couple of different solutions on the website for you to look at.

2. Create the UML class diagram for the `PrintCalendar` class as we originally designed it.

