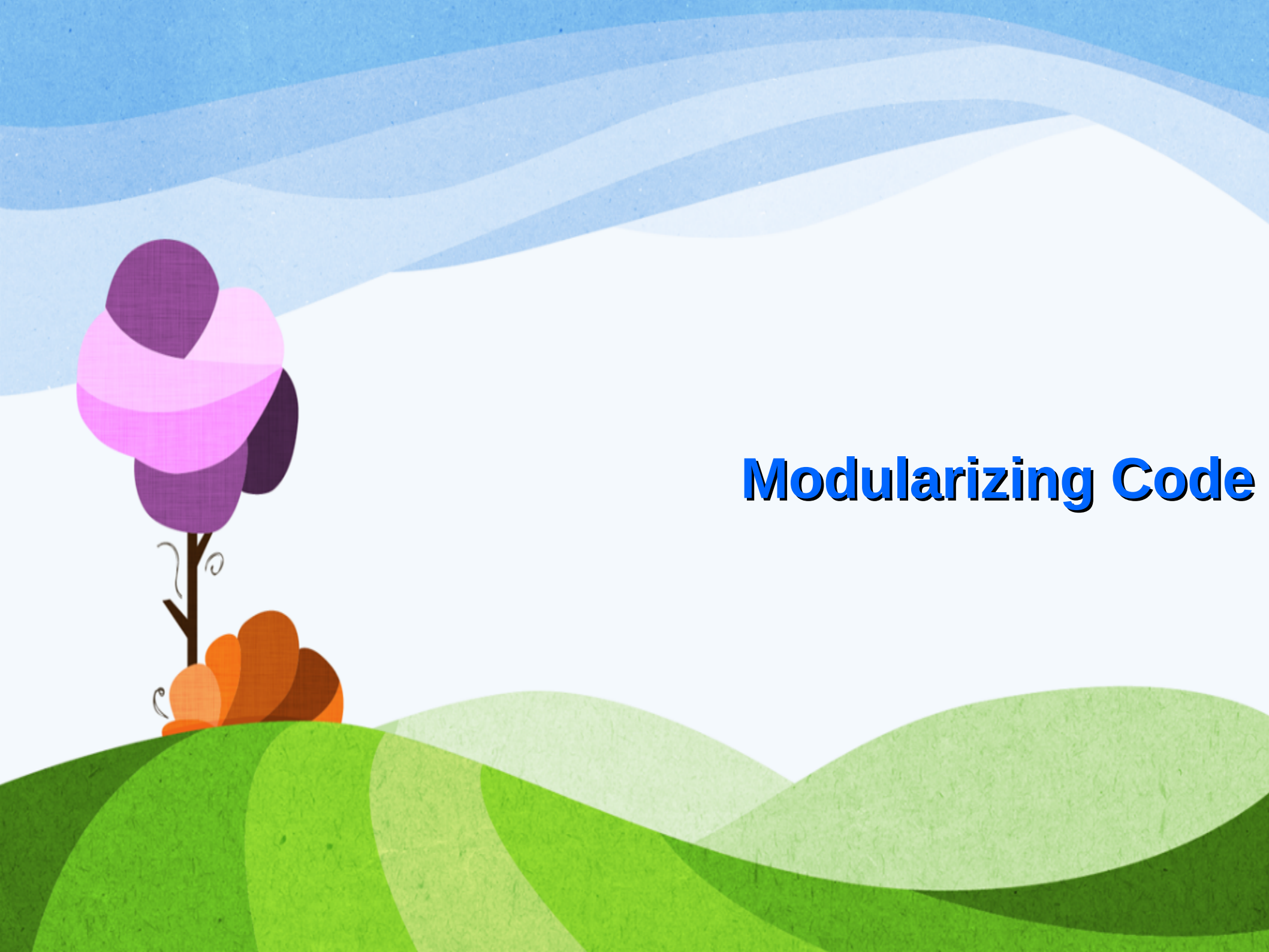
CS-2011 Introduction to Programming I

California State University, Los Angeles
Computer Science Department

Lecture X: Methods II


Modularizing Code

Modularizing Code

- ***modularizing code***: large amounts of code can be broken down into smaller "modules", each of which could function on its own and be reused over and over again.

- makes code easier to maintain and debug:
  - instead of focusing on the program as a whole, now you can focus on and test only one small portion of the overall program.
  - by testing methods one at a time, you can eliminate bugs much faster, than if you were to write a whole program and then test it.

Modularizing Code

- methods give you some advantages:
  - the code inside the method is isolated from the rest of the code in the program.
  - program logic becomes clear and the program is much easier to read
  - any errors which occur in a method will be contained in that method, which narrows the scope of debugging (you don't have to look all over the place for the bug you can just go to the specific method).

- See Code:
  - **`PrimeNumberNoMethods.java`**
  - **`PrimeNumberMethod.java`**


The Call Stack

The Call Stack

- When a method is invoked:
  - The JVM creates an ***activation record*** (a.k.a. ***activation frame***)
    - record which stores the data (variable values) for the method in a special area of memory known as the ***call stack***.

- the call stack is also known as the ***execution stack, runtime stack,*** or ***machine stack*** (often shortened to "the stack")

The Call Stack

- ***calling method*:** a method which calls another method

- **called method:** the method being called by another method.

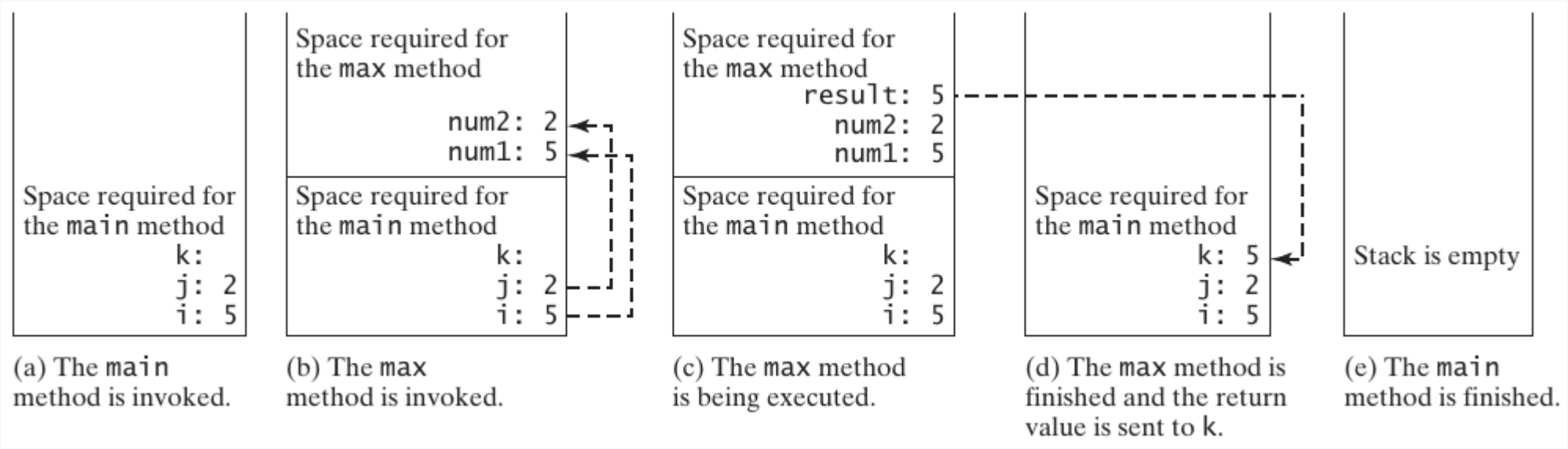
- when a method calls another method:
  - the ***calling method's*** activation record is kept intact
  - a new activation record is created for the ***method being called.***

- when the new method is finished:
  - control is returned to the caller
  - the activation record is removed from the stack.

The Call Stack

- Stacks store elements in a ***last-in, first-out (LIFO)*** order
  - The last item placed ON the stack, is the first item taken off of the stack.
- A stack is also a data structure, which you will learn about in CS-203.
  - you always work with the item on the TOP of the stack.
  - ***push onto the stack***: place an item on the top of the stack.
  - ***pop off the stack***: remove an item from the top of the stack.

The Call Stack



(a) The `main` method is invoked.

(b) The `max` method is invoked.

(c) The `max` method is being executed.

(d) The `max` method is finished and the return value is sent to `k`.

(e) The `main` method is finished.

Tracing the Call Stack

i is declared and initialized in the stack space allocated for **main**

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

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

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

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

  return result;
}
```

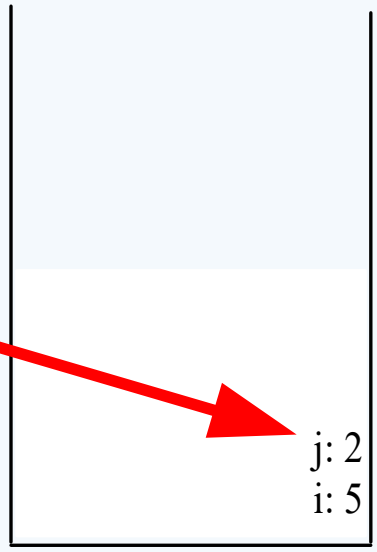
The main method is invoked.

Tracing the Call Stack

j is declared and initialized in the stack space allocated for **main**

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

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

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

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

  return result;
}
```



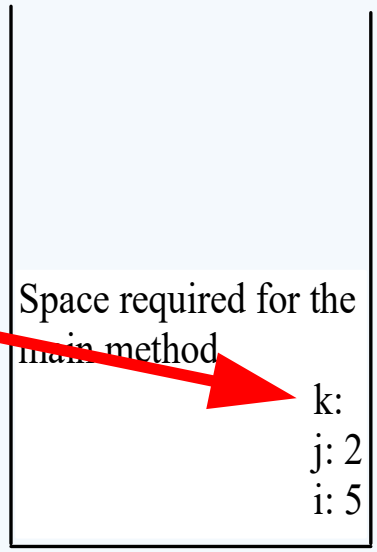
The main method is invoked.

Tracing the Call Stack

k is declared (but not yet initialized) in the stack space allocated for **main**

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

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

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

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

  return result;
}
```



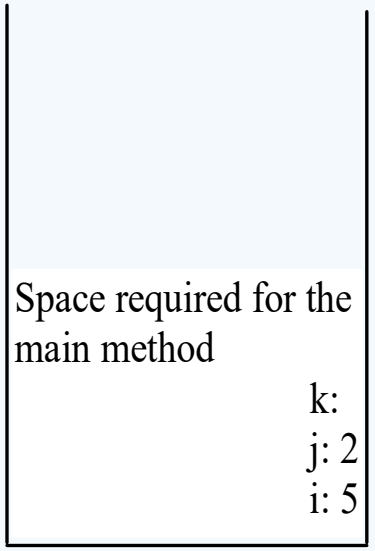
The main method is invoked.

Tracing the Call Stack

invoke **max(i, j)**

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

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

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

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

  return result;
}
```



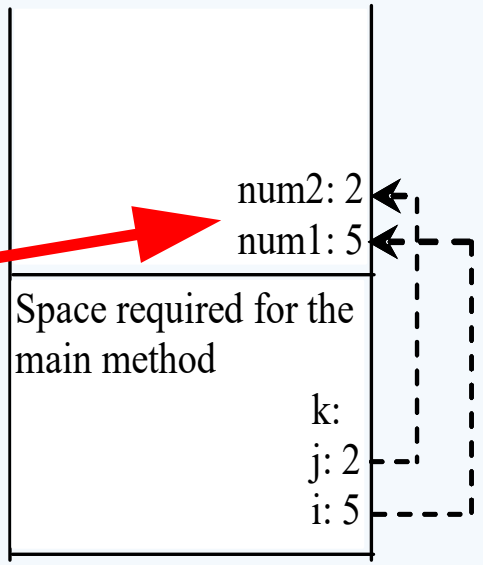
The main method is invoked.

Tracing the Call Stack

Allocate a new section on top of the stack
for the **max** method
pass (copy) the value of i to num1
pass (copy) the value of j to num2

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

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

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

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

  return result;
}
```



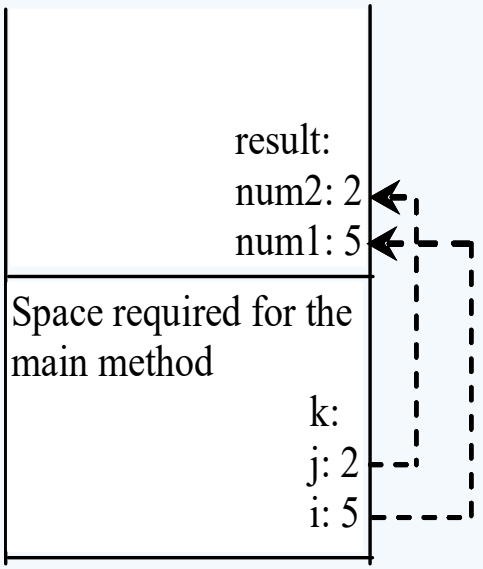
The max method is invoked.

Tracing the Call Stack

Declare result

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

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

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

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

  return result;
}
```



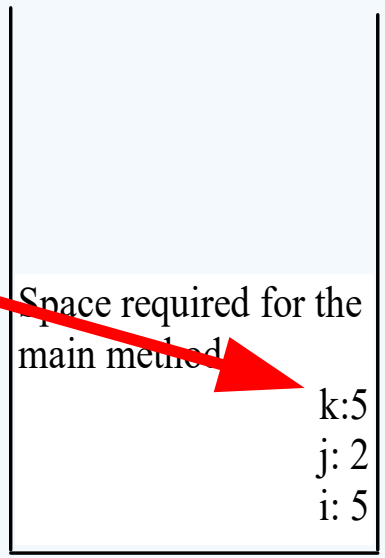
The max method is invoked.

Tracing the Call Stack

Return from the **max** method
the space for **max** on the stack is freed
the value returned from **max** is assigned to k

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

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

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

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

  return result;
}
```



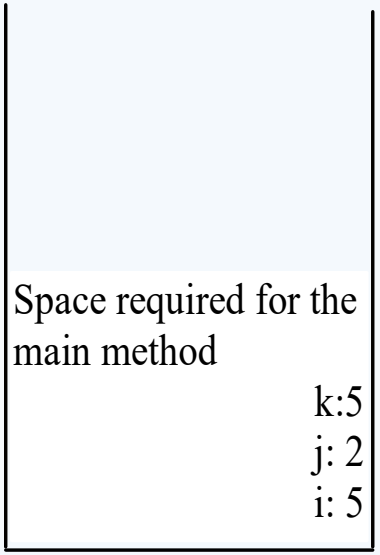
The main method
is invoked.

Tracing the Call Stack

Execute the print statement.

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

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

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

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

  return result;
}
```



The main method is invoked.


Overloading Methods

Overloading Methods

- What if we want to define multiple methods which accepts parameters of different types and performs the same function?
  - Eg. **`max(int, int)`** only works with integers.

- What if we need to determine which of two floating-point numbers has the maximum value?

- Solutions
  - Define **`maxDouble(double, double)`**
    - define a new method with a different name
    - not so great because then you have to remember the exact method name for all variations of the parameter lists.
  - Define overloaded methods
    - This way is better

Overloading Methods

- ***method overloading***: define two or more methods with the **same name** but **different parameter lists**.

- **different parameter lists** means**:**
  - having a different number of parameters or
  - having parameters with different data types.

- Example:

```
public static int max(int x, int y)
public static double max(double x, double y)
public static double max(int x, double y)
public static int max(int x, int y, int z)
```

Overloading Methods

- Compiler decides which method to use by matching the calling method's argument list to one of the overloaded method's parameter lists
  - the compiler will **always** try to choose the best match (if one exists).

- Example:

A. `public static int max(int x, int y)`
B. `public static double max(double x, double y)`
C. `public static double max(int x, double y)`
D. `public static int max(int x, int y, int z)`

`max(2, 3)` matches method A
`max(10.5, 11.5)` matches method B
`max(7, 13.2)` matches method C
`max(45, 23, 56)` matches method D

Overloading Methods

- **Differences in return type are irrelevant** in method overloading
  - Overloaded methods can have different return types
  - Methods with different return types but the same signature cause a compilation error

- **Different modifiers are irrelevant** in method overloading
  - `public` and `static` are modifiers

- **THE ONLY PART OF A METHOD THAT IS CONSIDERED WHEN OVERLOADING IS THE METHOD SIGNATURE (METHOD NAME AND PARAMETER LIST)**

Overloading Methods

- See Code: **`TestMethodOverloading.java`**

- Can you do **`max(2, 2, 5)`**? If so, which of the max methods is invoked?

- If the definition of **`max(int, int)`** is deleted, will the problem still work?

- Why is the method **`max(double, double)`** not invoked for the call **`max(3,4)`**?

Ambiguous Invocation

- What happens if there are two or more possible matches for an invocation of a method?
- Will the compiler make an arbitrary choice?
- How does the compiler choose?
- Answer: The compiler cannot decide for itself which to use, so this will trigger a compile error. This is called ***ambiguous invocation***.

Ambiguous Invocation

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }

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

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

Ambiguous Invocation

- Suppose you already defined a method

```
public static int myMethod(int n1, int n2, double n3)
```

- Which of the following overloaded methods can be defined in the same class?

```
public static int myMethod(int p1, int p2, double p3)
public static int myMethod(double n1, int n2, int n3)
public static double myMethod(int n1, int n2, double n3)
public static int  myMethod(int n1, int n2)
public static double  myMethod(int n1, int n2, int n3)
```

Method Abstraction and Stepwise Refinement

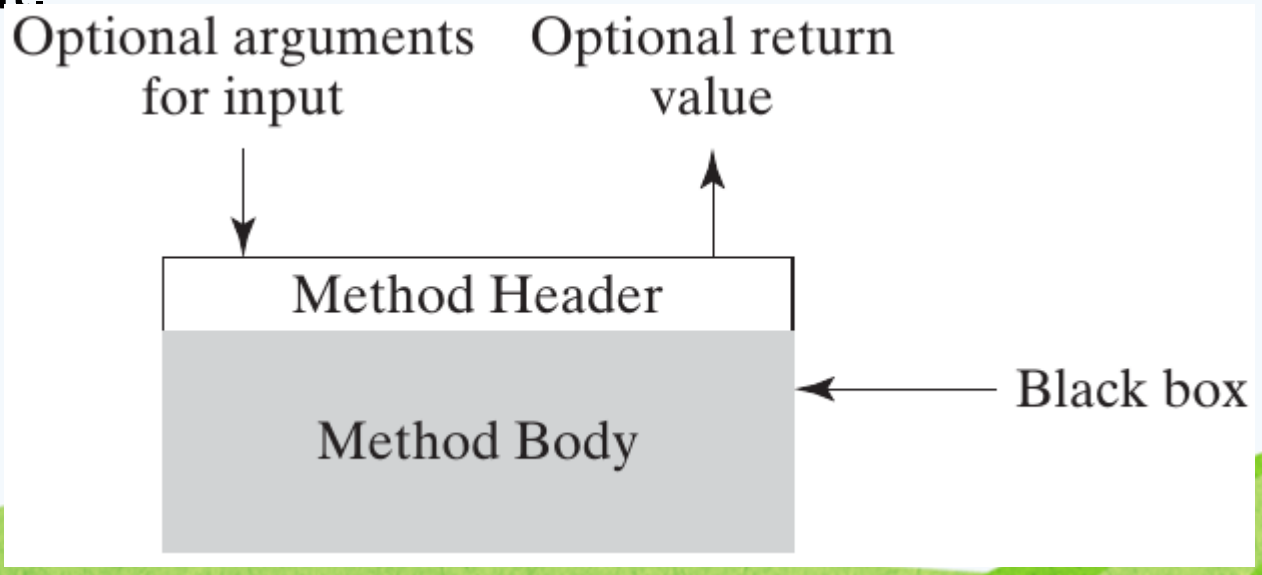
Method Abstraction

- ***method abstraction***: is the idea of separating the use of a method from its implementation
  - you don't need to know HOW something works in order to use it.
  - the details of how the method works are ***encapsulated*** in the method and hidden from anyone who uses the method.
    - this is known as ***encapsulation*** or ***information hiding***

- if you change HOW the method works (its implementation), the client program should not be affected as long as you don't change the method header.

Method Abstraction

- Think of the body of the method as a "black box"
  - this term comes up a lot in software engineering
  - you can see the input going into the box
  - you can see the output coming out of the box
  - you cannot see inside the box or what the box is doing with the input, how it produces the output, or any of its inner workings.
  - you only know how to "turn the box on", give it input, and get its output.



Stepwise Refinement

- the concept of method abstraction can also be applied to the process of developing a program.

- this is especially handy with larger programs
  - you can break up the program using a ***divide-and-conquer*** strategy (also known as ***stepwise refinement***)
  - here you break a larger problem down into smaller subproblems that are much easier to work with.
  - these subproblems can be broken down even further as necessary

Stepwise Refinement

- Let's write a program that displays the calendar for a given month of the year.

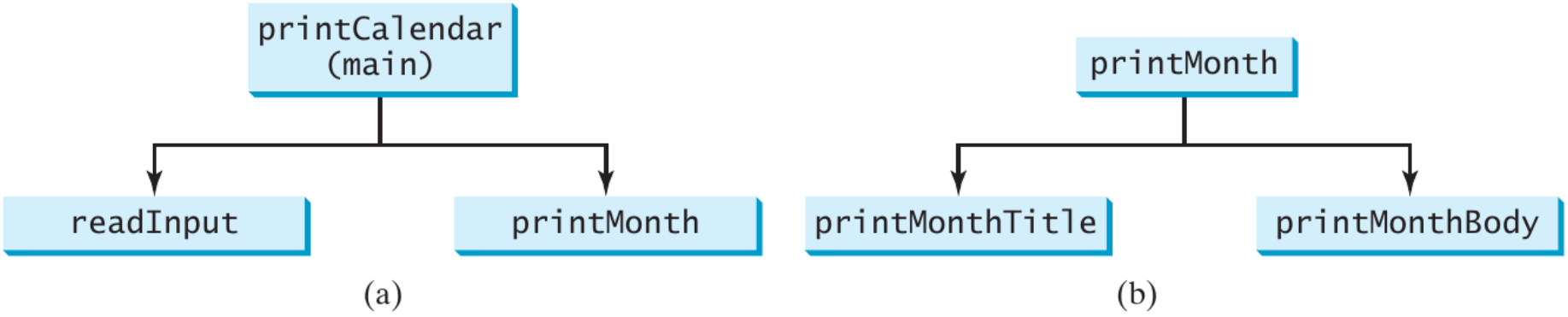
- The program asks the user for the year and the month and displays the entire calendar for the month.

```
Enter full year (e.g., 2012): 2012 ↵ Enter

Enter month as number between 1 and 12: 3 ↵ Enter

         March 2012
-----------------------------
 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
```

Top-Down Design

- How do we start? Should we jump right in coding?
  - NO NO NO, take time to plan out the program first.
  - let's use a ***top-down design*** (designing the problem from the top and working our way down)

- Let's break the problem down into subproblems first and work on the pieces before combining them into one program.
  - subproblem 1: get the user input
  - subproblem 2: print the calendar for the requested month



The structure chart shows that the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth` in (a), and that `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody` in (b).

Top-Down Design

- **`Scanner`** will get the input for the year and the month.

- Printing the calendar can be broken into two more subproblems:
  - printing the month title (the month, year, a dashed line, and the days of the week)
    - the month name is based on its numeric month (1 – 12) and can be calculated by a **`getMonthName`** method.
  - printing the month body
    - need to know which day of the week is the first day of the month (**`getStartDay`**)
    - need to know how many days the month has (**`getNumberOfDaysInMonth`**)

Top-Down Design

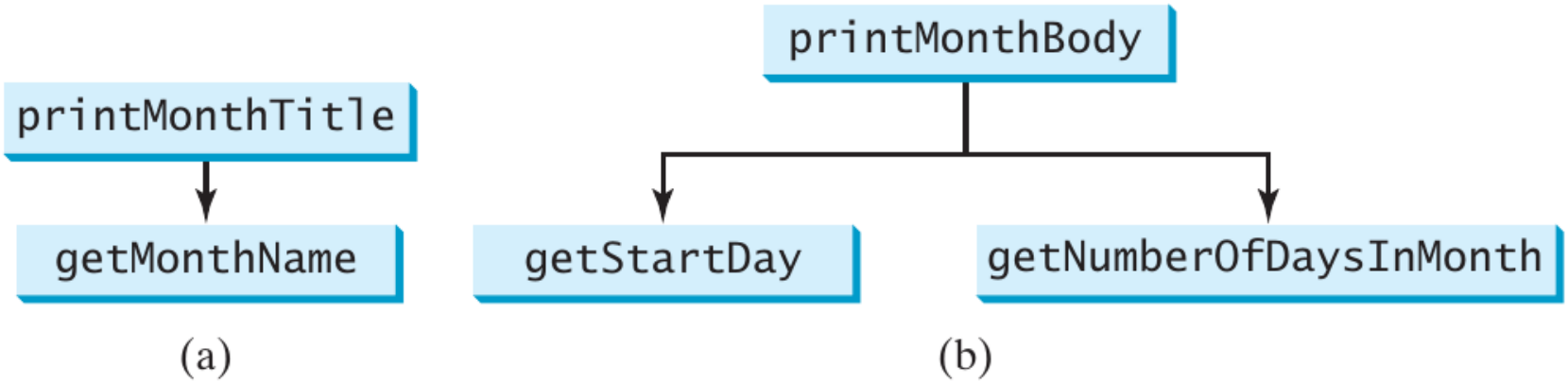


FIGURE 6.9 (a) To `printMonthTitle`, you need `getMonthName`. (b) The `printMonthBody` problem is refined into several smaller problems.

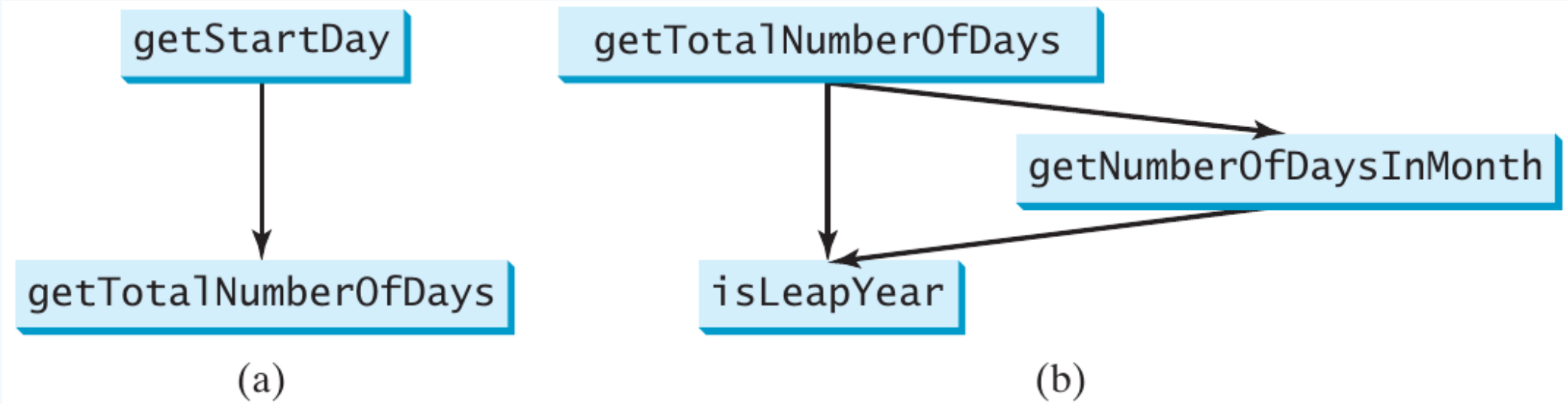
Top-Down Design

- How to calculate the start day?
  - assume you know the start day for January 1, 1800 (which was a Wednesday) let's make this a constant

    ```
    START_DAY_FOR_JAN_1_1800 = 3
    ```

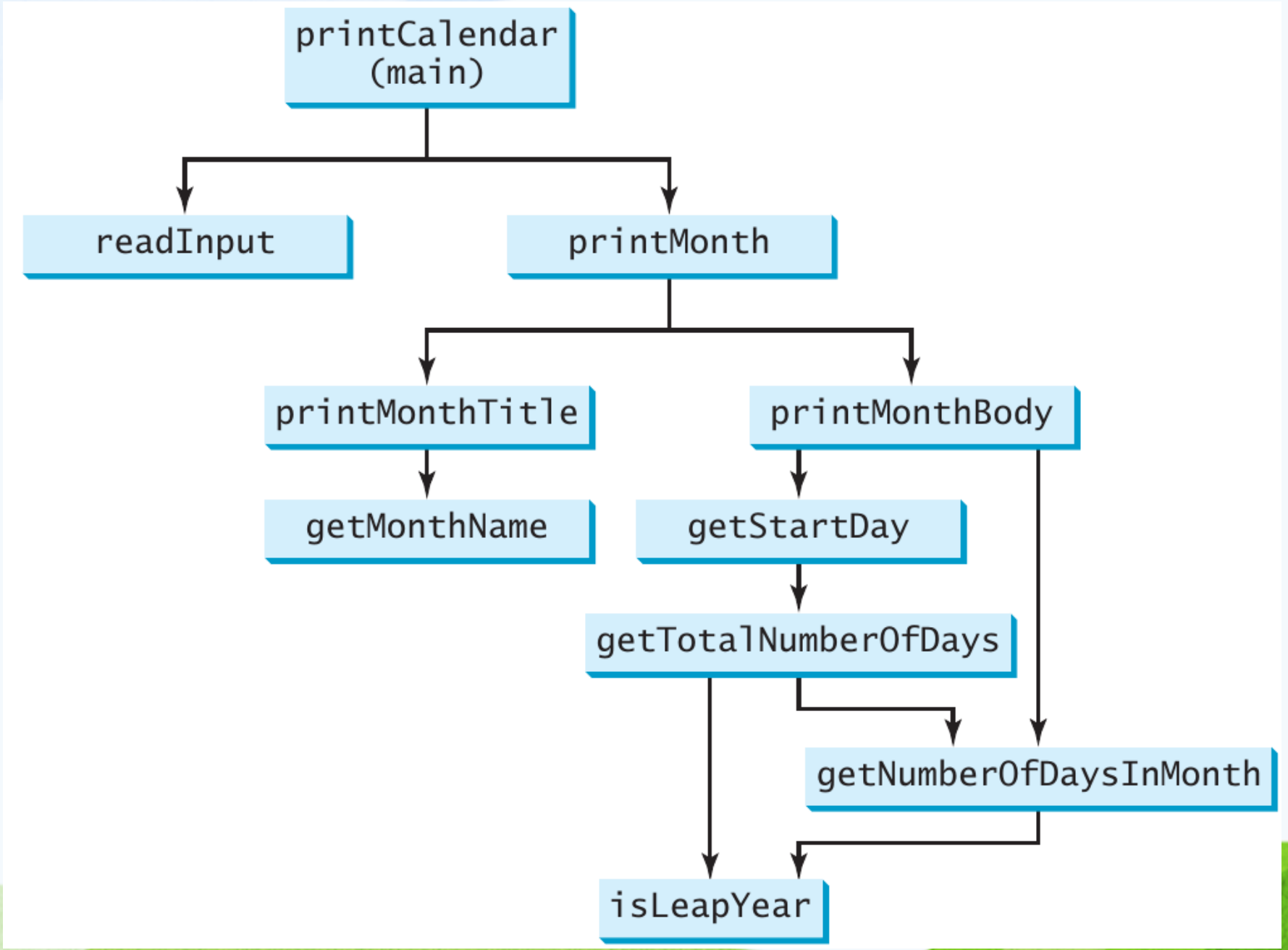
  - calculate the total number of days (**`totalNumberOfDays`**) between January 1, 1800 and the first date of the calendar month.
    - the start day is **`(totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7`**
    - every week has seven days so we mod by 7
  - **`getStartDay`** can be refined into **`getTotalNumberOfDays`**.
  - to get the total number of days we have to know if a year is a leap year or not AND the number of days in each month
  - **`getTotalNumberOfDays`** can be refined into two more subproblems:
    - **`isLeapYear`**
    - **`getNumberOfDaysInMonth`**

Top-Down Design



(a) To `getStartDay`, you need `getTotalNumberOfDays`. (b) The `getTotalNumberOfDays` problem is refined into two smaller problems.

Top-Down Design



Top-Down and/or Bottom-Up Implementation

- Now to implement the program. Notice each subproblem corresponds to a method in the implementation.
  - Decide if any subproblems can be easily combined or if there are any unnecessary methods.

- Decide on a "top-down" or a "bottom-up" approach.

Top-Down and/or Bottom-Up Implementation

- ***top-down implementation***: implements one method in the structure chart at a time from the top to the bottom

- this technique makes use of ***stubs*** (a simple but incomplete version of a method)
  - these are used for methods waiting to be implemented (think of them as place holders) and they allow the program to be correctly compiled and tested.
  - stubs help you to build a framework or outline of your program

- implement the main method first and then use a stub for **`printMonth`**

Top-Down and/or Bottom-Up Implementation

```
public class PrintCalendar {
  /** Main method */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter year
    System.out.print("Enter full year (e.g., 2012): ");
    int year = input.nextInt();

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

    // Print calendar for the month of the year
    printMonth(year, month);
  }

  /** A stub for printMonth may look like this */
  public static void printMonth(int year, int month){
    System.out.print(month + " " + year);
  }

  /** A stub for printMonthTitle may look like this */
  public static void printMonthTitle(int year, int month){
  }

  /** A stub for getMonthBody may look like this */
  public static void printMonthBody(int year, int month){
  }
```

Top-Down and/or Bottom-Up Implementation

```
  /** A stub for getMonthName may look like this */
  public static String getMonthName(int month) {
    return "January"; // A dummy value
  }

  /** A stub for getStartDay may look like this */
  public static int getStartDay(int year, int month) {
    return 1; // A dummy value
  }

  /** A stub for getTotalNumberOfDays may look like this */
  public static int getTotalNumberOfDays(int year, int month) {
    return 10000; // A dummy value
  }

  /** A stub for getNumberOfDaysInMonth may look like this */
  public static int getNumberOfDaysInMonth(int year, int month) {
    return 31; // A dummy value
  }

  /** A stub for isLeapYear may look like this */
  public static Boolean isLeapYear(int year) {
    return true; // A dummy value
  }
}
```

Top-Down and/or Bottom-Up Implementation

- ***bottom-up implementation***: implement one method from the tree at a time from the bottom up.

- test each method individually by calling the method and displaying its results.

- this allows you to completely debug a method separately from the entire program and eliminate most of the errors as you are coding.