Chapter 6 Methods

Contents

[Method 2](#_Toc37023361)

[Why use methods? 2](#_Toc37023362)

[Modularizing Code: 3](#_Toc37023363)

[Define a method 3](#_Toc37023364)

[Invoke 3](#_Toc37023365)

[Syntax: 4](#_Toc37023366)

[Parts of a Method / Consists of: 4](#_Toc37023367)

[Method Header 4](#_Toc37023368)

[Modifiers 5](#_Toc37023369)

[Return value type 5](#_Toc37023370)

[void 5](#_Toc37023371)

[Method Body 6](#_Toc37023372)

[Method name 6](#_Toc37023373)

[Parameters / arguments 6](#_Toc37023374)

[Method Signature 6](#_Toc37023375)

[Passing Parameters by Values 6](#_Toc37023376)

[Java is pass-by-value 7](#_Toc37023377)

[return statement 8](#_Toc37023378)

[Formal verses Actual Parameters: 9](#_Toc37023379)

[Placement of Methods & Invoking (or calling) Methods 11](#_Toc37023380)

[Methods – Matching Parameters 12](#_Toc37023381)

[Overloading Methods: 13](#_Toc37023382)

[Ambiguous invocation 14](#_Toc37023383)

[Scope of local variables 15](#_Toc37023384)

[Method Abstraction Encapsulation 15](#_Toc37023385)

[Abstraction 15](#_Toc37023386)

[Encapsulation 15](#_Toc37023387)

[Stepwise Refinement 16](#_Toc37023388)

[Stepwise Refinement Example: 16](#_Toc37023389)

[Implementation: Top-Down 16](#_Toc37023390)

[Implementation: Bottom-Up 16](#_Toc37023391)

[Benefits of Stepwise Refinement 17](#_Toc37023392)

[Simpler Program 17](#_Toc37023393)

[Reusing Methods 17](#_Toc37023394)

[Easier Developing, Debugging, and Testing 17](#_Toc37023395)

[Better Facilitating Teamwork 17](#_Toc37023396)

# Method

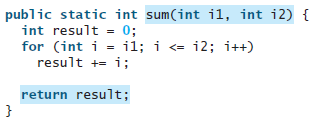
Method: a collection of statements grouped together to perform an operation

A method is a block of code which only runs when it is called.

A method is a collection of statements that perform some specific task and return the result to the caller.

Methods are used to perform certain actions, and they are known as functions in other programming languages.

Example a block of code that sums two numbers:



You can pass data, known as parameters, into a method.

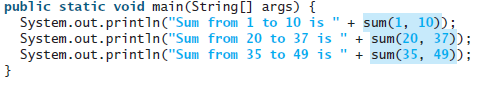
Why use methods?

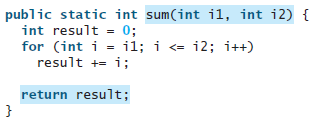
Methods can be used to define reusable code, methods organize and simplify code.

To reuse code: define the code once, and use it many times.

Utilizing methods make code modular = organized and simplified.

Example: calling the method with different values:





# Modularizing Code:

Modularizing makes the code easy to maintain and debug and enables the code to be reused.

Logic is clearer, program easier to read.

Isolates the problem for computing calculations.

Errors are confined within the methods, thus the scope of debugging is narrowed.

Methods do one thing (modularizing code), only one thing; that way if a change happens for that one thing, it is contained and its effects will not cascade through the rest of the code = easy to maintain, easy to debug

# Define a method

The idea of defining a method means to write the method header and the method body

Invoke

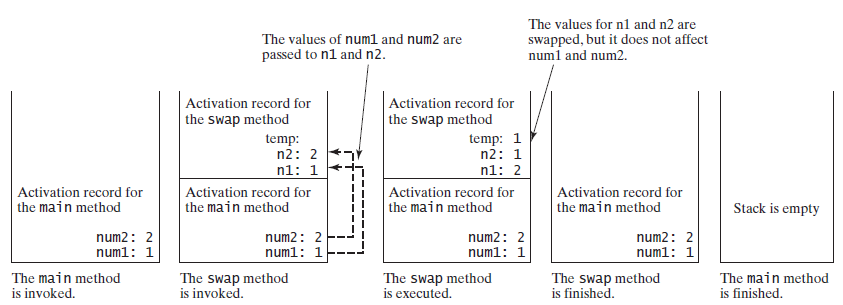
Each time a method is invoked/called, the code in that methods {} are executed. The code in that method is never executed if the method is never invoked/called.

Invoking/Calling a method: calling it from another method & passing a value, or values, to the method

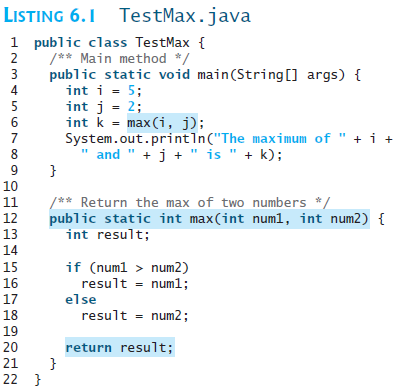
* Invoking = calling (the terms are interchangeable)
* this is the power of the method, being able to call/invoke gives the ability to reuse that code and can do so by using different parameters each time it is called/invoked

Each time a method is invoked, the system creates an activation record / activation frame that stores parameters and variables for the method and places the activation record in an area of memory known as a call stack (execution stack, runtime stack, or machine stack, generally referred to as *the stack*)

Example:



Example of invoking statement:



Parameters

Invoking statement

Method header

# Syntax:

modifier returnValueType methodName( listOfParameters ){

//body

}

# Parts of a Method / Consists of:

* method name
* Parameters
* Return type
* body



Method Header: specifies the modifiers, return value type, method name, and parameters

* must have return value type
* void method
* return statement is not required
* value-returning method
* return statement required!
* formal parameters or parameters
* these are optional! Not all methods have to have parameters/arguments

## Modifiers

For the purposes of CSC 102, the modifiers for methods will always be: public static

This means that the method will be available to the entire class

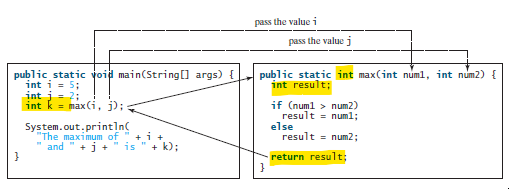
Example:



## Return value type

If a method needs to return a value, the method header has to state the return data type. ***The return type, the stated return type in the method header and the invoking statement all have to match up on this!***

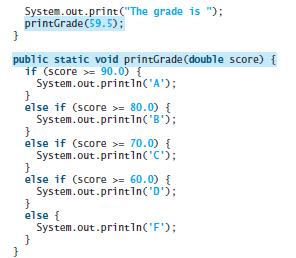
Example:



### void

If the methods does not need to return a value, then the return type is void (keyword meaning return nothing) and there is not a return statement

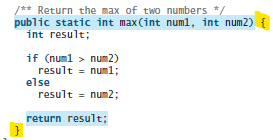
Example:



Method Body

The statements in the method to be executed are enclosed between braces.

Example:



## Method name

The name should be reflective of what that method is doing; e.g. a method that sums, should be sum, not multiply.

Remember to use the Java naming conventions; start with lower case letters

Example:



## Parameters / arguments

Any values that the method would need; if the method does not need variables passed then the parentheses are empty.

The variables are separated by commas; variables are defined by the data type then the identifier within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().

Example:



Method Signature: the method name and the parameter list together

Example:



# Passing Parameters by Values

The actual parameters/arguments are passed by value to the formal parameters/argument when invoking a method.

The power of a method is its ability to work with parameters that can change

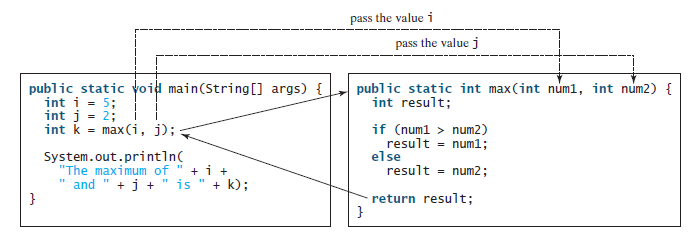
When calling/invoking a method with parameters, you need to provide the arguments given in the order as their respective parameter in the method signature.

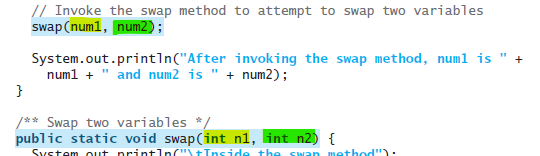
Parameter Order Association: the order of parameters listed in the method signature

NOT Rule:

* Number: how many variables
* Order: listing order of variables
* Type: data types

Examples:



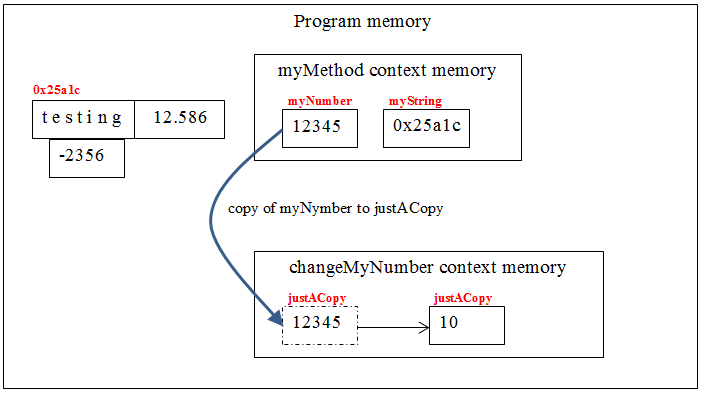


num1 value is passed to swap and stored in swap as the variable n1

num2 value is passed to swap and stored in swap as the variable n2

# Java is pass-by-value

This means it is the value that is stored in main memory for that variable is what is passed to the method. It is very important to understand this concept!



Remember that primitive data types stores the value, like myNumber stores 12345 in main memory, and reference data type, like myString stores where in memory the content of that variable is stored.

For further reading, try:

<https://www.geeksforgeeks.org/g-fact-31-java-is-strictly-pass-by-value/>

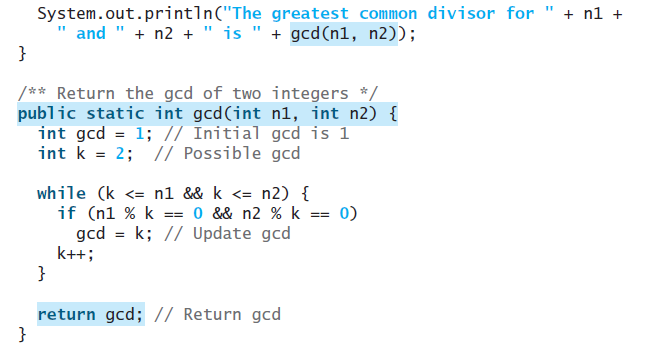
# return statement

A return statement is required for a value-returning method. (Remember that a method that returns nothing has a return type of void)

There cannot be any code after the return statement; the return statement informs the JVM to return that value and leave the method immediately.

***The return type, the stated return type in the method header and the invoking statement all have to match up on this!***

Example:



If a method returns a value, then the invoking statement has to “catch” that value. There are two ways to “catch” that value.

1. In a System.out.println line to print the value out right away

Example:

// invoking statement:

System.out.println(findMax(2, 3));

//method header

public static int findMax(int num1, int num2)

1. Assign the returning value to a variable. When you do this, the value and the assigning variable have to be of the same type, int to int, double to double, char to char, String to String, etc.

Example:

// invoking statement

int maxValue = findMax(2, 3);

//method header

public static int findMax(int num1, int num2)

**A method can only return one value!**

The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.



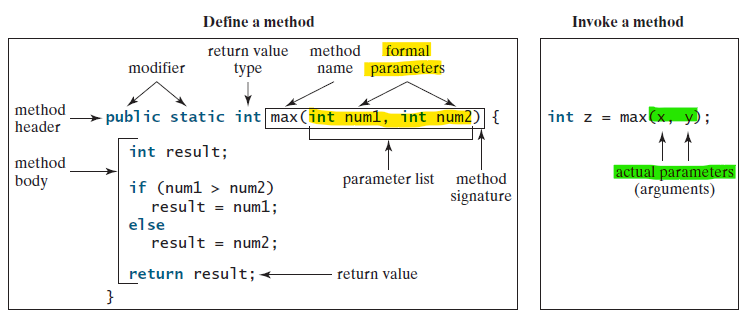
To fix this problem, delete if (n < 0) in (a), so that the compiler will see a return statement to be reached regardless of how the if statement is evaluated.

# Formal verses Actual Parameters:

Formal parameters: the parameter list on the methods header

Actual parameters: the parameters that are being passed to the method – remember that these are the values that are stored in main memory for those variables.

Examples:



import java.util.Scanner;

public class MethodExample1 {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

System.out.print("Input your name: ");

main method

String myName = input.nextLine();

welcome(myName);

System.out.println("back in main");

}// end main

**Formal Parameter**: the identifier myName

Invoking (or calling) the method welcome

**Actual parameter**: the value is *Anne* - what is passed to the welcome method

public static void welcome(String name) {

welcome method

System.out.println("Welcome to Java Methods " + name);

}// end welcome

}// end class

# Placement of Methods & Invoking (or calling) Methods

public class MethodExample3 {

public static void main(String[] args) {

Invoking a Method method

Main method

System.out.println("Start here in main method");

aMethod();

System.out.println("Back to main method");

aMethod();

System.out.println("In main method again - program ends here");

}// end main

Empty parameters

Returns nothing

aMethod method

public static void aMethod() {

System.out.println("\nIn the aMethod method, printing a line of text.\n");

}// end PrintLine Method

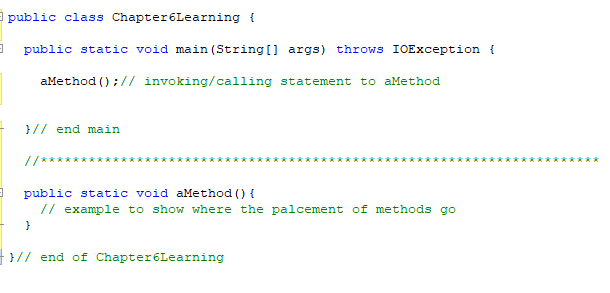
}// end class

Notice that the **invoking** (or call) is ***inside*** main

and that the **method is *outside*** of main

Methods are placed after the closing } braces of main and before the closing } of the program/class

In Java, the standard format is to have main listed first, then the methods in the order that they are used in the program.



# Methods – Matching Parameters

public class methods4Types {

public static void main (String [] args) {

int a;

int b;

Invoking methods

printHelloTenTimes();

printHelloXTimes(2);

System.out.println( "The cube of 25 is " + cubeMe(25) );

a = 6;

b = 9;

System.out.println("The greater of " + a + " and " + b + " is " +

returnGreatest(a,b));

}// end main

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//simple method that takes in one parameter, returns a float value

public static float cubeMe (int x) {

//give back to the calling program, the cube of x

return x\*x\*x;

}// end cubeMe

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//simple method that takes in two parameters, returns an int value

public static int returnGreatest (int x, int y) {

//give back to the calling program, the cube of x

if (x>y) {

return x;

}

else {

return y;}

}// end returnGreatest

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//method that takes in one parameter, returns nothing

public static void printHelloXTimes (int count){

int i;

for (i = 1; i <= count; i++) {

System.out.println("Hello world, count = " + i);

}

}// end printHelloXTimes

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//extremely simple method that takes in no parameters, returns nothing

public static void printHelloTenTimes (){

int i;

for (i = 1; i <= 10; i++) {

System.out.println("Hello world, count = " + i);

}

}// end printHelloTenTimes

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

}// end class

# Overloading Methods:

Overloading Methods enables you to define the methods with the same name as long as their method signatures are different.

Methods that perform the same function with different type of parameters should be given the same name.

Thus two or more methods have the same name in one class but the parameter list will be different.

The Java compiler determines which method to use based on the method signature, by looking at the invoking statements and looking at the method headers for a match.

Overloading makes programs clearer and more readable.

Bottom line on overloading: the method headers must have the same name and different parameters – not modifiers or return types

public static int max(int num1, int num2) {

if (num1 > num2)

return num1;

else

return num2;

}

The max method above works only with the int data type. But what if you need to find which of two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

public static double max(double num1, double num2) {

if (num1 > num2)

return num1;

else

return num2;

}

If you call max with int parameters, the max method that expects int parameters will be invoked; if you call max with double parameters, the max method that expects double parameters will be invoked.

**Overloaded methods must have a different parameter lists**.

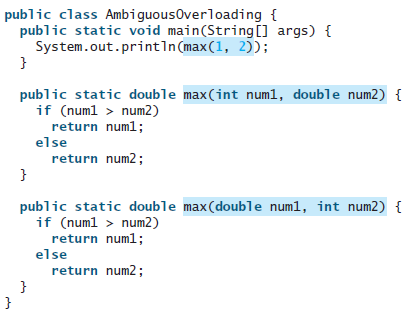
You ***cannot*** overload methods based on different modifiers or return types.

Ambiguous invocation:

Sometimes there are two or more possible matches for an invocation of a method due to similar method signature, so the compiler cannot determine the most specific match. This is referred to as ***ambiguous invocation***.

Ambiguous invocation happens when two or more possible matches for an invocation of a method occurs and the compiler cannot determine the specific match to a specific method header.

Example:



This is ambiguous since an int can be assigned to a double without casting.

# Scope of local variables

A local variable: a variable defined inside a method

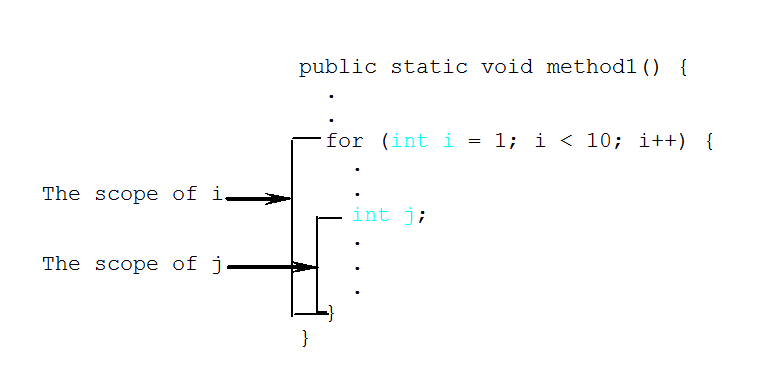
A parameter is a local variable

Scope: the part of the program where the variable can be referenced.

The scope of a local variable starts from its declaration and continues to the end of the block that contains the declaration of that variable.

A local variable must be declared before it can be used.

You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.



# Method Abstraction Encapsulation

## Abstraction

It is achieved by separating the use of a method from its implementation of the method.

The client can use a method without knowing how it is implemented.

If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature.

## Encapsulation

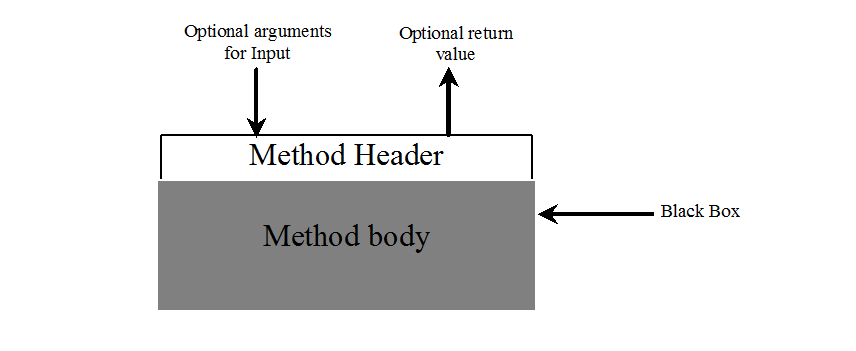
The details of the implementation are encapsulated in the method and hidden from the client who invokes the method.

This is known as information hiding or encapsulation – hiding the details (the data) from the user, you protect the data of the program this way.

The key to developing software is to apply the concept of abstraction and encapsulation!

The implementation of the method is hidden from the client in a “black box,” as shown

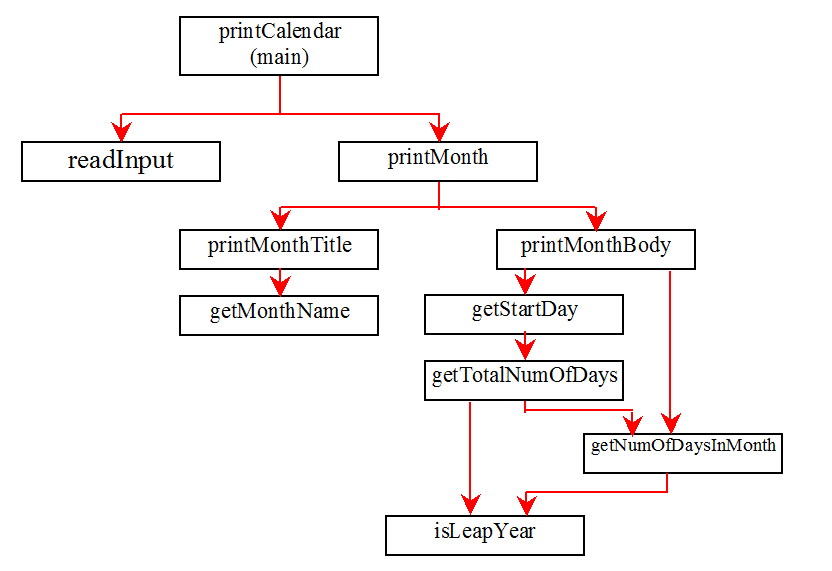
A “black box”



# Stepwise Refinement

When writing a large program, you can use the divide-and-conquer strategy, also known as **stepwise refinement**, to decompose it into sub problems (more manageable problems).

## Stepwise Refinement Example:



## Implementation: Top-Down

Top-down approach is to implement one method in the structure chart at a time from the top to the bottom. Stubs can be used for the methods waiting to be implemented. A stub is a simple but incomplete version of a method. The use of stubs enables you to test invoking the method from a caller. Implement the main method first and then use a stub for the printMonth method.

## Implementation: Bottom-Up

Bottom-up approach is to implement one method in the structure chart at a time from the bottom to the top. For each method implemented, write a test program to test it. Both top-down and bottom-up methods are fine. Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

## Benefits of Stepwise Refinement

Stepwise refinement breaks a large problem into smaller manageable subproblems. Each subproblem can be implemented using a method. This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

### Simpler Program

The print calendar program is long. Rather than writing a long sequence of statements in one method, stepwise refinement breaks it into smaller methods. This simplifies the program and makes the whole program easier to read and understand.

### Reusing Methods

Stepwise refinement promotes code reuse within a program.

This reduces redundant code.

Textbook example: The isLeapYearmethod is defined once and invoked from the getTotalNumberOfDaysand getNumberOfDayInMonthmethods.

### Easier Developing, Debugging, and Testing

Since each subproblem is solved in a method, a method can be developed, debugged, and tested individually. This isolates the errors and makes developing, debugging, and testing easier. When implementing a large program, use the top-down and/or bottom-up approach. Do not write the entire program at once. Using these approaches seems to take more development time (because you repeatedly compile and run the program), but it actually saves time and makes debugging easier.

### Better Facilitating Teamwork

When a large problem is divided into subprograms, subproblems can be assigned to different programmers. This makes it easier for programmers to work in teams.