Programming Design and Implementation

Lecture 5: Submodules

Updated: 25th March, 2020

Mark Upston
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

"Zero" Marks

- A student who does any of the following in a submitted, assessable answer will receive heavy penalties, up to and including **zero** marks for that question:
  - Uses **`continue`**
  - Uses **`break`** in any other place than a **`switch`** statement
  - Uses **`goto`**
  - Has more than one **`return`** statement in a method
  - Has a **`return`** statement in a method anywhere but the last statement of the method
  - Uses **`System.exit()`** anywhere but the last statement of the **`main()`** method
  - Uses global variables for anything other than class fields
  - Uses a ternary operator
- Note: similar efforts in pseudo code will also receive zero marks

Modularity

- Submodules are used to break an algorithm into smaller, manageable, modular parts
- Each submodule *should* be limited to a single task (in this unit it **must** be)
  - Ease of design
  - Readability
  - Code re-use
  - Debugging
- Submodules **must** be useful
  - A submodule like this:

```
public static void printMySubmodule()
{
    System.out.println("I wrote a submodule just to do this!");
}
```

is pointless

Submodules

- Submodules must be designed
  - Pseudo-code in this unit
- Look at the requirements
  - For each task in the requirement specification, ask yourself: *"Is there a submodule already written and tested for this task?"*
    - **Yes**: Use it
    - **No**: Design one
- See worked example at the end of this lecture

Designing Submodules

- You must write a contract first
- A contract is the agreement between the submodule and the outside world (i.e., the rest of the algorithm)
- A contract consists of:
  - **IMPORT**: Information which is supplied to the sub module
  - **EXPORT**: Information that the submodule will pass back to the calling submodule
  - **ASSERTION**: Facts that will be true after the submodule has been executed
    - Assertions act as hooks for tracking down errors in the algorithm
    - It may be that no assertion can be made

Methods

- Methods are Submodules
  - Java Methods are the implementation of a submodule
- A method consists of:
  - A Header
  - A Body
- The Header consists of:
  - The Type of method
  - The Name of the method
  - The Parameters to the method
    - In Java, each parameter **must** be declared separately
- The Body consists of a sequence of Java statements encapsulated by '**{**' and '**}**'

The `main()` Method

- A Java program starts by executing the `main()` method
  - The `main()` method provides the starting point for the program
  - Within the `main()` method:
    - Other methods will be invoked
    - Objects will be created
    - Methods within those objects invoked

```
 public static       void      main   (String [] args) { .... }
|-Type of method-| |-Return-| |-Name-| |- Parameters -| |-Code-|
```

*Note: The "Code" above **must** be formatted to Curtin Coding Standards*

- The Parameters above will be covered in the I/O Lecture

Comments

- Comment blocks should be used to describe all submodules
  - Method Contract
    - **`IMPORT`**, **`EXPORT`** and **`ASSERTION`**
  - Purpose of the submodule (its job)
  - Dates Modified
- This will often be omitted in these notes to save room and for clarity
- You are expected to do this for **<u>all</u>** of your programs/classes

Comment Blocks

```
/************************************************
 * Author: Mark Upston                          *
 * Date: 25/02/2020                             *
 * Purpose: To do something                     *
 ************************************************/
public class MyJavaApp
{
    public static void main(String[] args)
    {
        // Code
    } // End main

...
```

Comment Blocks (2)

```
...

    /**************************************************
     * Name: myMethod                                 *
     * Date: 25/02/2020                               *
     * Import: a (int), b (int)                       *
     * Export: val (double)                           *
     * Purpose: Do do a part of something             *
     **************************************************/
                     // Parameters declared individually
    public static double myMethod(int a, int b)
    {
        // Code
    } // End myMethod
} // End Class
```

void Methods

- A **void** method does its intended job, then passes control back to the calling method, but with no return value
  - i.e., No export
  - See functional methods for returned values

Example: Pseudo Code

```
MAIN:
    sayHello <- "World"
END MAIN

SUBMODULE: sayHello
IMPORT: aParam
EXPORT: none
ALGORITHM:
    OUTPUT "I would like to say hello to " aParam
END SUBMODULE
```

Example: Java

```java
import java.util.*;
public class MyWorldClass
{
    public static void main(String[] args)
    {
        sayHello("World");
    }

    // SUBMODULE: sayHello
    // IMPORT: aParam (String)
    // EXPORT: none
    public static void sayHello(String aParam)
    {
        System.out.println("I would like to say hello to "
                              + aParam);
    }
}
```

Invoking (calling) methods

- There are two possibilities when one method calls another:
  - The method being called is in the same class as the calling method
  - The method being called is in a different class
- If the method is in the same class then it is invoked as in the previous example
- If the method is in a different class then:
  - If the method is not a static method then it can only be invoked via an `Object` variable
    - `variableName.methodName();`
    - `sc.nextDouble();`
  - If the method is a static method then it can also be invoked by specifying the class name
    - `Math.sqrt(9);`

Functional Methods

- In mathematics a function is the mechanism used to map a set of inputs into one output value
  - `z = f(x, y);`
- In Java we have methods which can return a single piece of data to the calling module
  - These methods have the type of data they will return specified in the header

    ```
    public int calculateMyAge(String name)
    ```
- Remember **void** methods are methods which do not return anything

    ```
    public void sayHello(String aParam)
    ```

Submodule Example: Pseudo Code

```
MAIN:
    INPUT inches
    cms := convertInchesToCms <- inches
    OUTPUT "CMs are: " + cms
END MAIN

SUBMODULE: convertInchesToCms
IMPORT: ins
EXPORT: cm
ALGORITHM:
    cm = ins * 2.54
END convertInchesToCms
```

Submodule Example: Java

```
import java.util.*;
public class MyThirdProgramClass
{
    public static void main(String[] args)
    {
        double inches, cms;
        Scanner sc = new Scanner(System.in);
        System.out.print("Input Inch: ");
        inches = sc.nextDouble();
        cms = convertInchesToCms(inches);
        System.out.println("CMs are: " + cms);
    }

    private static double convertInchesToCms(double ins)
    {
        double cm;
        cm = ins * 2.54;
        return cm;
    }
}
```

Submodule Example (2): Pseudo Code

```
MAIN:
    INPUT x
    INPUT y
    average := calculateMean <- x, y
    OUTPUT average
END MAIN

SUBMODULE: calculateMean
IMPORT: a, b
EXPORT: mean
ALGORITHM:
    mean := (a + b) / 2.0
END calculateMean
```

Submodule Example (2): Java

```
import java.util.*;
public class MyFourthProgramClass
{
    public static void main(String[] args)
    {
        int x, y;
        double average;
        Scanner sc - new Scanner(System.in);

        System.out.print("Enter and integer: ");
        x = sc.nextInt();
        System.out.print("Enter another integer: ");
        y = sc.nextInt();
        average = calculateMean(x, y);
        System.out.println("Mean of " + x + " & " + y + " is: "
                                    + average);
    } // End main
    ...
```

Submodule Example (2): Java

```
...
   /*****************************************************
    * SUBMODULE: calculateMean                            *
    * IMPORT: a (int), b (int)                            *
    * EXPORT: mean (double)                               *
    ******************************************************/
    private static double calculateMean(int a, int b)
    {
        return (double)(a + b) / 2.0;
    } //End calculateMean
} // End Class
```

Functions and Methods

- We are dependant, in most non-trivial arithmetic, on various functions
  - sin, log, sqrt, etc.
- Calculators have predefined functions that return a value when handed argument(s)
  - Provide **`x`** and it returns **`sin(x)`**
  - Arguments are passed to a methods parameters
- In Java, functions are implemented as non-void methods
- Java provides a library of such methods in the **`Math`** class
- Predefined functions avoid "reinvention of the wheel" & allow reusability of code already written and tested
  - If a function does not exist, you must write the method for it
- Values are handed to a function is the argument list and a single result is returned in the name of the function

Java Math Class

- This class contains the methods that perform mathematical functions we need in non-trivial calculations
- It is part of the "`package.java.lang`" which is automatically imported for you into every Java program
- It is **`final`** so we can't have subclasses (see later), has no constructors and its methods are static:
  - Means the class name must always be used before the method name
  - See slide 15
  - e.g., `Math.sqrt(x);`
- Two constants:
  - `Math.E`
  - `Math.PI`

Some methods in the Math class

Function	Argument	Constraints	Result
abs(x)	int/long/float/double		Same as arg
ceil(x)	double		double
floor(x)	double		double
round(x)	float/double		int/long
rint(x)	double		double
os(x)	double	x in radians	double
exp(x)	double		double
log(x)	double	x > 0	double
pow(x,y)	double	x^y, x > 0	double
sqrt(x)	double	x >= 0	double
max(x,y)	int/long/float/double		Same as arg
min(x,y)	int/long/float/double		Same as arg

Examples

```
Expression                  Result
-------------------------------
Math.round(2.6)             3
Math.round(-3.15)           -3
Math.rint(-3.7)             -4.0
Math.ceil(3.7)              4.0
Math.floor(3.7)             3.0
Math.pow(2.0, 3.0)          8.0
Math.sqrt(-9)               NaN
Math.abs(-4)                4
Math.abs(4.0)               4.0
```

The Assignment Operator

- Recall that in programming the equals sign is performing the action of place the value of the **expression** on the right hand side into the variable on the left hand side
  - Change the left hand side to be the same as the value of the right hand side
  - Remember:
    - **= != ==**

Expressions

- An expression can be:
  - A literal constant (e.g., **`42`**)
  - A named constant (e.g., **`Math.PI`**)
  - A variable (e.g., **`lifeTheUniverseAndEverything`**)
  - The return value of a functional method (e.g., **`Math.sin()`**)
  - A combination of all of the above, joined together with a valid set of operators (e.g., **`Math.PI / 180.0`**)

Reminder - Expression Guidelines

- Never use mixed mode arithmetic
  - Use type casting to avoid mixed mode arithmetic
- Precedence rules are the same as in mathematics
- Use parentheses to simplify readability of complex expressions
- Use intermediate steps to split complex expressions into explicitly seperate steps
- Don't over-parenthesise simple expressions
- Beware or algebraic simplicity:

$$x = \frac{y - p}{z - q}$$

- This is written in Java as `x = (y - p) / (z - q)`

Invoking (calling) a Method

- To call a method, an **expression** must be provided for each matching parameter in the called method
- When the method is invoked:
  - Each **expression** is evaluated
  - The result (argument) is passed to the relevant method parameter
  - The method becomes the active module

Expressions Evaluated for Methods

```
d = (-b + Math.sqrt(Math.pow(b, 2.0) - 4.0 * a * c)) / (2.0 * a);
```

- Precedence is preserved
- Methods have a very high precedence
- Steps
  1. `temp1 = Math.pow(b, 2.0);`
  2. `temp2 = temp1 - 4.0 * a * c;`
  3. `temp3 = Math.sqrt(temp2);`
  4. `d = (-b + temp3) / (2.0 * a);`

Method Parameters

- Variables declared to be local to a method can be:
  - Declared inside the method block
  - Declared as a method parameter
- The difference is that:
  - The parameter variables get their initial values from the calling method
  - The others are auto initialised
    - Although, good programming practice means that we should explicitly initialise these variables

```
public void exampleOfAMethod(int paramOne, float paramTTwo)
{
    int localOne = 0;
    float localTwo = 0.0;
    ...;
}
```

Characteristics of Method Parameters

- Each parameter is initialised with the result of the matching expression in the method call
- Parameters are local variables which means that each parameter has no knowledge as to where its initial value came from
- A modification to the value of a parameter will have no effect on the value of the corresponding argument in the calling module

Exceptions to the Rules

- Non OO programming languages (and some OO languages) have a second way of defining parameters
  - This second way is called passing by reference
    - This will be covered in UCP (`COMP1000`)
- In Java there are two data types which are automatically passed by reference:
  - References to objects
  - Arrays (a special type of object)
- A variable which has been declared as being a particular class type will contain the address of the object (a reference to the object)
- Any modification to the object will be reflected in the calling module
- Classes are covered in Lecture 6

Reference Variables

- To create an object, you must:
  - Declare a variable to be of the desired class
  - Use the **new** operator to create an instance of that class
  - The variable is set to the memory location of the object
  - It becomes a *reference* ('it points to the') object
  - Example:

```
Frog froggie;
froggie = new Frog("Kermit");
froggie.leap();
```

- You have already seen this
  - Remember `Scanner sc = new Scanner(System.in);`

Primitive vs. Reference: Variables

- Primitive Variables:
  - Direct access to the data
  - Modification of the data involves modification to the contents of the variable
- Reference Variables:
  - Indirect access to the data
  - Modification of the data within an object **must** be done via object methods
  - Example:

```
Frog froggie;
String frogName = froggie.getName();
froggie.setName("Kermit the Second");
```

Primitive vs. Reference: Example

- In Java, reference variables contian the address of objects
- Objects are created from classes
  - We wont be discussing classes in detail, until Lecture 7
- For now, think of an object as something which contains data
- This data can only be accessed via special methods
- When objects are created, we call that object construction
- Imagine a class called **`Integer`**
  - Objects of this class contain an integer
  - To access the integer, we call a method:
    - `getValue();`
  - To modify the integer we call a method:
    - `setValue()`
  - To create a new object we supply an int to the constructor:
    - `myInt = new Integer(5);`

Primitive vs. Reference: Example (2)

- This means that when we create an **`Integer`** object, we store the address of the object in the Java reference variable
- In other words, the address in the variable refers to the location
- Some languages call this type of variable a **pointer**
- Hence in Java:
  - A primitive variable contains the actual data
  - A reference variable contains the memory address of the data
- Example:

```
int iAmAPrimitive;
Integer iAmAnObject;
iAmAPrimitive = 5;
iAmAnObject = new Integer(5)
```

iAmAPrimitive	1000	5
iAmAnObject	1004	2000
	1008	
	~	
elsewhere in memory	2000	5
	2004	
	2008	

Primitive vs. Reference: Parameters

- This means that, in Java:
  - If we supply a primitive variable as a parameter to a method, then its contents contains the actual data and so a copy of its contents is passed into the sub module.
  - If we supply a reference variable as a parameter to a method, then its contents contains the memory address of the data and so a copy of the address is passed into the sub module.
- To demonstrate, the next slide shows a Java main method which will create a primitive and an object and then pass them to seperate methods:
  - Each method will modify the integer and we can see the effect in the related variables in main:
    - The primitive variable wont change
    - The integer inside the object will change

TryToChange Example

```
import java.util.*;
public class TryToChange
{
    public static void tryToChangePrimitive(int inInt)
    {
        inInt = inInt + 42;
    }
    public static void tryToChangeObject(Integer inInt)
    {
        int tempInt;
        tempInt = inInt.getValue();
        tempInt = tempInt + 42;
        inInt.setValue(tempInt);
    }
...
```

TryToChange Example (2)

```
...
    public static void main(String[] args)
    {
        int intPrimitive = 5;
        Integer intObject = new Integer(5);

        System.out.println("Before");
        System.out.println("Primitive is: " + intPrimitive);
        System.out.println("Object is: " + intObject);

        tryToChangePrimitive(intPrimitive);
        tryToChangeObject(intObject);

        System.out.println("After");
        System.out.println("Primitive is: " + intPrimitive);
        System.out.println("Object is: " + intObject);
    }
}
```

Example: Compiling and Running

- To compile:

```
mark@314lab:~/PDI/$ javac TryToChange.java
```

- To run:

```
mark@314lab:~/PDI/$ java TryToChange
```

- Produces the output:

```
Before
Primitive is: 5
Object is: 5
After
Primitive is: 5
Object is: 47
```

Passing and Returning Arrays

- Arrays can be passed as a parameter to a method, just like any other variable
  - One difference from primitives: passing the array doesn't *copy* the array, it only passes a *reference* to the array
    - In this way, arrays act more like objects, rather than primitives
  - So if the method changes the passed-in-array, it will affect the 'original' array in the calling method
    - Since the two are in fact the exact same array
- You can also return an array from a method

Passing and Returning Arrays: Java

```java
public static void main(String[] args)
{
    int[] anArray = new int[3];
    int[] copyOfAnArray;

    anArray[0] = 1;
    anArray[1] = -16;
    anArray[2] = 5;

    copyOfAnArray = copyIntArray(anArray);
}

...
```

Passing and Returning Arrays: Java (2)

```
...

public static int[] copyIntArray(int[] arrayToCopy)
{
    int[] dupArray;
    dupArray = new int[arrayToCopy.length];

    for(int ii = 0; ii < arrayToCopy.length; ii++)
    {
        dupArray[ii] = arrayToCopy[ii];
    }
}
```

Java toString() Method

- A convention is that all Java objects should have a **toString()** method
  - This method is used to provide a **String** representation of the data stored in the object
    - Covered later

The Java String class

- We must **never** forget that **String**'s are objects
- Whats wrong with:

```
(nameOne == nameTwo)
```

The equals() Method

- So whats wrong with:

  `(nameOne == nameTwo)`
- Comparing the contents of two reference variables is no use, because we are simply comparing two memory addresses and not the information stored at the addresses
- The **equals()** method overcomes this problem

  `(nameOne.equals(nameTwo))`
- A Java convention is that all Java objects have an **equals()** method

Modular Grade Example

Pseudo Code:

```
SUBMODULE: markGrade
IMPORT: mark (Integer)
EXPORT: grade (Character)
ALGORITHM:
    newMark := mark DIV 10
    CASE newMark OF
        8, 9 or 10
            grade := 'H'
        7
            grade := 'D'
        6
            grade := 'C'
        5
            grade := 'P'
        DEFAULT
            grade := 'F'
    ENDCASE
END markGrade
```

Java:

```java
public static char markGrade(int mark)
{
    int newMark = mark / 10;
    char grade;
    switch(newMark)
    {
        case 8: case 9: case 10:
            grade = 'H';
        break;
        case 7:
            grade = 'D';
        break;
        case 6:
            grade = 'C';
        break;
        case 5:
            grade = 'P';
        break;
        default:
            grade = 'F';
    }
    return grade;
}
```

Modular Input Algorithm

- In your programs use the following template:
  - Note: We will modify it again slightly when we cover Exceptions

```
SUBMODULE: inputValueFromUser
IMPORT: prompt (String), lower (Integer), upper (Integer)
EXPORT: value (Integer)
ASSERTION: value will be in the range of lower and upper inclusive
ALGORITHM:
    outputPrompt := prompt
    DO
        OUTPUT outputPrompt + lower + upper
        INPUT num
        outputPrompt := "Error please enter a number in the valid
                         range" + newline + prompt
    WHILE((num < lower) OR (num > upper))
    ASSERTION: lower <= value <= upper
END inputValueFromUser
```

- This can be used for Real input as well, just modify the IMPORT and EXPORT parameters

Modular sqrt: Pseudo

```
SUBMODULE: sqrt
IMPORT: number (Real)
EXPORT: squareRoot (Real)
ALGORITHM:
    squareRoot := number / 2.0
    COMMENT: First guess

    DO
        t := squareRoot
        COMMENT: Next guess will be closer

        squareRoot := (t + (number / t)) / 2.0 (REALS)
    WHILE((t - squareRoot) IS NOT 0.0)
    ASSERTION: When (t - squareRoot) is 0, we cannot get
               any closer
END sqrt
```

Modular sqrt: Java

```
public static double sqrt(double number)
{
    double t;
    double squareRoot = number / 2.0;

    do
    {
        t = squareRoot;
        squareRoot = (t + (number / t)) / 2.0;
    } while(Math.abs(t - squareRoot) > 0.0000000001);

    return squareRoot;
}
```

Modular FOR Loop: Pseudo

```
SUBMODULE: printArray
IMPORT: myArray
EXPORT: none
ASSERTION: myArray is output to the screen
ALGORITHM:
    FOR index := 0 TO myArray LENGTH CHANGEBY 1
        OUTPUT myArray[index]
    ENDFOR
END printArray
```

Modular FOR Loop: Java

```
/***********************************************
 * SUBMODULE: printArray                       *
 * IMPORT: myArray (double[])                  *
 * EXPORT: none (void)                         *
 * ASSERTION: Contents of array output to screen  *
 ***********************************************/
public static void printArray(double[] myArray)
{
    for(int ii = 0; ii < myArray.length; ii++)
    {
        System.out.println(myArray[ii]);
    }
}
```

Modular Factorial: Pseudo

```
SUBMODULE: calcNFactorial
IMPORT: n (Integer)
EXPORT: nFactorial (Integer)
ASSERTION: if n is 0 or negative, then nFactorial is 1

ALGORITHM:
    nFactorial := 1
    FOR ii := 2 TO n CHANGEBY 1
        nFactorial := nFactorial * ii
    ENDFOR

ALTERNATE ALGORITHM:
    nFactorial := 1
    FOR ii := n DOWNTO 2 CHANGEBY -1
        nFactorial := nFactorial * ii
    ENDFOR
```

FOR Loop Example (2): Java

```
/*******************************************************
 * SUBMODULE: calcNFactorial                            *
 * IMPORT: n (int)                                      *
 * EXPORT: nFactorial (long)                            *
 * ASSERTION: if n 0 or negative, then nFactorial is 1 *
 *******************************************************/
public static long calcNFactorial(int n)
{
    long nFactorial = 1;
    for(int ii = 2; ii <= n; ii++)
    {
        nFactorial *= (long)ii;
    }
    return nFactorial;
}
```

FOR Loop Example (3): Algorithm

```
SUBMODULE: nChooseR
IMPORT: n (Integer), r (Integer)
EXPORT: nChooseR
ASSERTION: No validation of n and r, both of which should be positive
           and n >= r
ALGORITHM:
    nChooseR := n! / ((n - r)! * r!)
```

- Does this mean we still need 3 FOR loops, one for each factorial calculated?

FOR Loop Example (3): Java

- No, use the **calcNFactorial()** method we just wrote
  - Code once, use many

```
public static int nChooseR(int n, int r)
{
    long bottom;
    bottom = calcNFactorial(n - r) * calcNFactorial(r);
    return (int)(calcNFactorial(n) / bottom);
}
```

Modular Algorithm Design: Pseudo

- This algorithm is the modular version of the Times Table Program

```
MAIN:
    maxTable := integerInput <- "Enter the number of Times Tables: ",
                                 1, 12
    FOR table = 1 TO maxTable CHANGEBY 1
        outputTable <- table
    ENDFOR
    ASSERTION: 1 to maxTable times table output to user.
END MAIN

...
```

Modular Algorithm Design: Pseudo (2)

```
SUBMODULE: inputValueFromUser
IMPORT: prompt (String), lower (Integer), upper (Integer)
EXPORT: value (Integer)
ASSERTION: value will be in the range of lower and upper inclusive
ALGORITHM:
    // SEE SLIDE 49
END inputValueFromUser

SUBMODULE: outputTable
IMPORT: table (Integer)
EXPORT: none
ASSERTION: table is in the range of 1 to 12
ALGORITHM:
    OUTPUT "The " table " Times Table"
    FOR number := 1 TO 12 CHANGEBY 1
        OUPTUT table " x " number " = " (table * number)
    ENDFOR
    ASSERTION: table Times Table is output to the user
END outputTable
```

Modular Algorithm Design: Java

```
import java.util.*;
public class TimesTable
{
    public static void main(String[] args)
    {
        int maxTable
        maxTable = integerInput("Please enter the number
                                 of times tables", 1, 12);
        for(int table = 1; table <= maxTable; table++)
        {
            outputTable(table);
        }
    }

...
```

Modular Algorithm Design: Java (2)

```
...

    public static int integerInput(String prompt, int lower, int upper)
    {
        int value;
        Scanner sc = new Scanner(System.in);
        String outputPrompt = prompt;
        do
        {
              System.out.print(outputPrompt + " between " + lower +
              value = sc.nextInt();
              outputPrompt = "Error: please enter a number in
                                the valid range\n " + prompt;
        } while((value < lower) || (value > upper));
        return value;
    }

...
```

Modular Algorithm Design: Java (3)

```
...

    public static void outputTable(int table)
    {
        System.out.println("The " + table + " Times Table");
        for(int number = 1; number <= 12; number++)
        {
            System.out.println(table + " x " + number + " = "
                               + (table * number));
        }
    }
}
```

Designing Submodules

- PI (π) can be estimated as the sum of the infinite series:

$$\pi = \sqrt{12}\sum_{n=0}^{\infty}\frac{(-3)^{-n}}{2n+1} = \sqrt{12}\left(\frac{(-1)^n}{1*3^0}+\frac{(-1)^n}{3*3^1}+\frac{(-1)^n}{5*3^2}+\frac{(-1)^n}{7*3^3}+...\right)$$

- Design an algorithm that will:
  - Input a number of terms to approximate π, your algorithm should repeat until the number of values input is between 20 and 100 (inclusive)
  - Calculate the value of each term, storing it in an array
  - After all of the individual terms have been calculated, calculate the final value of π, storing it in the last element of the array
  - Upon completion of all the calculations, the algorithm should output each value in the array to the user

Each Task Becomes a Submodule

```
MAIN:
    terms := integerInput <-- "Please input a value between 20 and 100
                               (inclusive)", 20, 100
    array := Real ARRAY (SIZE OF terms)
    top := -1

    FOR term := 0 TO terms - 1 CHANGEBY 1
        top := top * -1
        array[term] := top / calcTerm <-- term  // Real division
    END FOR

    array[terms] := calcPI <-- array
    outputArrayContents <-- array
END MAIN

COMMENT: "Instead of a for loop, could use a submodule for calcPI,
          which calls another submodule for adding"
```

Each Task Becomes a Submodule (2)

```
SUBMODULE: integerInput
IMPORT: prompt (String), lower (Integer), upper (Integer)
EXPORT: value (Integer)
ASSERTION: value will be in the range of lower and upper inclusive
ALGORITHM:
    // SEE SLIDE 49
END integerInput

SUBMODULE calcTerm
IMPORT termNo (Integer)
EXPORT term (Real)
ASSERTION: term will be the value of the denominator
ALGORITHM:
    term := (2.0 * term + 1.0) * 3.0 ^ term
END calcTerm
```

Each Task Becomes a Submodule (3)

```
SUBMODULE: calcPI
IMPORT: array (Real ARRAY)
EXPORT: pi (Real)
ASSERTION: pi will be the summation of the array times by sqroot 12
ALGORITHM:
    pi := 0.0
    FOR count := 0 to array.length - 2 CHANGEBY 1
        pi := pi + array[count]
    END FOR
    pi = pi * 12.0 ^ (1/2)
END calcPI

SUBMODULE outputArrayContents
IMPORT array (Real ARRAY)
EXPORT none
ASSERTION: Will output all the elements of the array to the screen
ALGORITHM:
    FOR count := 0 to array.length -1 CHANGEBY 1
        OUTPUT "Value is: " array[count]
    END FOR
END calcTerm
```

"Zero" Marks

- A student who does any of the following in a submitted, assessable answer will receive heavy penalties, up to and including **zero** marks for that question:
  - Uses **`continue`**
  - Uses **`break`** in any other place than a **`switch`** statement
  - Uses **`goto`**
  - Has more than one **`return`** statement in a method
  - Has a **`return`** statement in a method anywhere but the last statement of the method
  - Uses **`System.exit()`** anywhere but the last statement of the **`main()`** method
  - Uses global variables for anything other than class fields
  - Uses a ternary operator
- Note: similar efforts in pseudo code will also receive zero marks

Next Week

- The next Lecture will cover:
  - Basic Model Classes