CSI 117 Introductory Object-Oriented Program Analysis and Design

# Unit 3
# Modules

***Text Reference:***
- ***Gaddis, Chapter 3***
- ***Appendix C (P596 - Defining a Module, Calling a Module, and Parameter Variables)***

## Objectives

- Define the term module
- Describe the benefits of using modules in a program design
- Use pseudocode to design and call a module
- Design modules with no parameters
- Use local variables in a module
- Design modules using parameters to exchange information
- Pass arguments to a module
- Explain the difference between Pass by Value and Pass by Reference parameters
- Use global variables and global constants



## Introduction

- A **module** is a group of statements that exists within a program for the purpose of performing a specific task
- Most programs are large enough to be broken down into several subtasks or modules
- **Divide and conquer**: It's easier to tackle smaller tasks individually
- Modules are also called **procedures, subroutines, subprograms, methods, and functions**



## Introduction (cont.)

- 5 benefits of using modules
  - Simpler code
    - Small modules easier to read than one large one
  - Code reuse
    - Can call modules many times
  - Better testing
    - Test separate and isolate then fix errors
  - Faster development
    - Reuse common tasks
  - Easier facilitation of teamwork
    - Share the workload

## Defining and Calling a Module

- The code for a module is known as a module **definition**.

```
Module showMessage()
    Display "Hello world."
End Module
```

- To **execute** the module, write a pseudocode statement that calls it.

```
Call showMessage()
```

- Note the new pseudocode keywords

```
Module
End Module
Call
```

5

## Defining and Calling a Module (cont.)

- Module's naming rules:
  - Contain no spaces
  - Contain no special character except underscore (_)
  - Must begin with a letter
  - Contain no keywords
- Module's naming conventions:
  - should be descriptive enough so that anyone reading the code can guess what the module does
  - Should start with lowercase letter
  - Use camelCase
  - Because modules perform actions, most programmers prefer to use **verb** in module names

6

## Defining and Calling a Module (cont.)

- Definition contains two parts:
  - A **header**
    - The starting point of the module
    - Contains the keyword **Module**, the name of the module, and ( )
  - A **body**
    - consists of the statements within the module

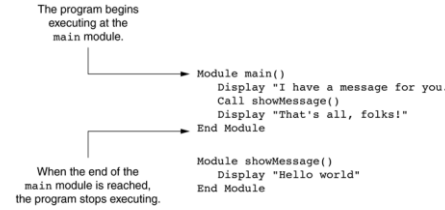
```
Module module_name( )      <-- Header
    Statement
    Statement              <-- Body
    Etc.
End Module                 <-- End of module
```

**Note:** It is common practice in most programming languages to put a set of parenthese ( ) after a module name

7

## Defining and Calling a Module (cont.)

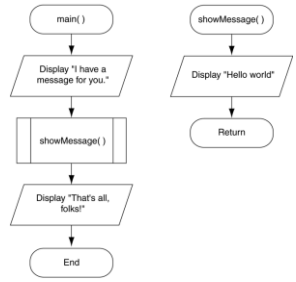
- Notice that the program contains a module named **main()**.
- Every program MUST contain exactly one module named **main()**. Program execution always begins with **main()**.
- A call must be made to the module in order for the statements in the body to execute.




8

## Defining and Calling a Module (cont.)

- When flowcharting a program with modules, each module is drawn **separately**




9

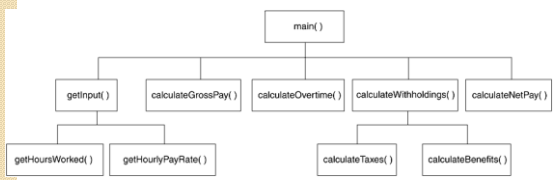
## Defining and Calling a Module (cont.)

- A **top-down design** is used to break down an algorithm into modules by the following steps:
  - The overall task is broken down into a series of subtasks.
  - Each of the subtasks is repeatedly examined to determine if it can be further broken down.
  - **A module is created for each subtask**.

10

## Defining and Calling a Module (cont.)

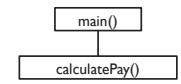
- A hierarchy chart gives a visual representation of the relationship between modules.
- The details of the program are excluded.




11

## Defining and Calling a Module (cont.)

- Example: Let's develop a modular program that will allow a user to enter the number of hours an empolyee worked and the hourly pay rate. The program will calculate and display the total amount of the pay check. Two modules
  - The main() module
  - calculatePay() that will input all the data, perform the calculations, and display the result.
- Let's start with the hierarchy chart:



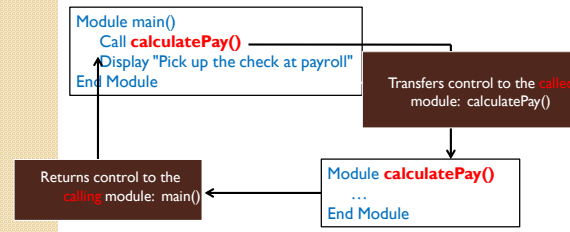

12

```
Module main()
   Call calculatePay()       //call (invoke or execute) the module
   Display "Pick up the check at payroll"
End Module

Module calculatePay()        //define (create) a module
  //declare local variables
   Declare Real hours
   Declare Real payRate
   Declare Real grossPay

   Display "Enter the number of hours worked"
   Input hours
   Display "Enter the hourly pay rate"
   Input payRate
   Set grossPay = hours * payRate
   Display "Total pay is: ", grossPay
End Module
```



## Defining and Calling a Module (cont.)

- When a program executes, the operating system calls the main() module
- **Invoking**, or **calling**, a module causes it to execute






## Local Variables

- A ***local variable*** is declared inside a module and **cannot** be accessed by statements that are outside the module.
- ***Scope*** describes the part of the program in which a variable can be accessed.
- Variables with the same scope must have different names.



## Arguments and Parameters

- Sometimes modules need to share data
- Modules use **arguments** and **parameters** to share data
  - arguments on **the calling statement** and
  - parameters in **the module header**

## Arguments and Parameters (cont.)

- An **argument** is a value that is sent to a called **(invoked)** module
- **Arguments** are found in the **calling statement**
- When calling a module with arguments, you can pass
  - Literals (like 25, or "Jones", or 'B')
  - Variables (like numStudents or empName)
  - Constants (like MAX_STUDENTS)



## Arguments and Parameters (cont.)

- A **parameter** is a variable that **receives the value of an argument** passed to it when the module is invoked (called)
- **Parameters** are found in a **module header**
- Each parameter in the module header must include:
  - a data type and a
  - parameter name
- A **parameter list** contains multiple parameter declarations separated by commas. Multiple arguments can be passed sequentially into a **parameter list**.
- Each time a module is called, its parameters are reinitialized to use the new argument values that are passed in when the call takes place



## Arguments and Parameters (cont.)

- Arguments passed to a module must match the parameters in the module's header in **three** ways:
  - **Number**: the number of arguments is the same as the number of parameters
  - **Data Type**: the data type of each argument matches the data type of the corresponding parameter
  - **Order**: the order of the arguments matches the order of the parameters



## Arguments and Parameters (cont.)

**Example:** Two arguments passed into two parameters

```
Module main()
   Display "The sum of 12 and 45 is"
   Call showSum(12, 45)
End Module

Module showSum(Integer num1, Integer num2)
   Declare Integer result
   Set result = num1 + num2
   Display result
End Module
```

## Module calculatePay() with No Parameters

```
Module main()
   Call calculatePay()
   Display "Pick up the check at payroll"
End Module

Module calculatePay()
   Declare Real hours
   Declare Real payRate
   Declare Real grossPay

   Display "Enter the number of hours worked"
   Input hours
   Display "Enter the hourly pay rate"
   Input payRate
   Set grossPay = hours * payRate
   Display "Total pay is: ", grossPay
End Module
```

main() has no parameters

No arguments are passed to calculatePay()

calculatePay() has no parameters



## Creating Modules With Parameters

- Example:
  - Suppose that in the previous example, we decide to input the values in the **main()** module
  - Then **main()** must **pass arguments** to the **calculatePay() parameters** in order to communicate the data



## Creating Modules With Parameters

```
Module main()
    Declare Real hours, payRate
    Display "Enter the number of hours worked"
    Input hours
    Display "Enter the hourly pay rate"
    Input payRate
    Call calculatePay(hours, payRate)
    Display "Pick up the check at payroll"
End Module

Module calculatePay(Real hoursWorked, Real hourlyPayRate)
    Declare Real grossPay

    Set grossPay = hoursWorked * hourlyPayRate
    Display "Total pay is: ", grossPay
End Module
```

Two arguments are passed to calculatePay()

The module has two parameters



## Creating Modules With Parameters

- Pass **by Value** vs. Pass **by Reference**
  - Pass by ***Value*** means that only a copy of the argument's value is passed into the module.
    - **One-way** communication: the calling module can only communicate with the called module, as shown in our previous example.
  - Pass by ***Reference*** means that the argument is passed into a reference variable.
    - Two-way communication: the calling module can communicate with called module; and called module can modify the value of the argument.

## Creating Modules With Parameters

- Example:
  - Suppose now we decide to input the values in the **main()** module, and to also output the value of **grossPay** in the **main()** module
  - Then **calculatePay()** must use **pass by value** parameters for the input data, and a **pass by reference** parameter for **grossPay** in order to communicate the data back to the **main()** module



## Creating Modules With Parameters

```
Module main()
    Declare Real grossPay
    Call calculatePay(grossPay)
    Display "Total pay is: ", grossPay
    Display "Pick up the check at payroll"
End Module

Module calculatePay(Real Ref grossPay)
    Declare Real hours, payRate
    Display "Enter the number of hours worked"
    Input hours
    Display "Enter the hourly pay rate"
    Input payRate
    Set grossPay = hours * payRate
End Module
```

Notice the new keyword *Ref* for a pass by reference parameter. The calculatePay() module can change the value of this parameter and the new value will be communicated back to the calling module. That means the main() module can display the new value for grossPay.



## Global Variables & Global Constants

- A **global variable** is accessible to all modules.
  - Its scope is the entire program
  - It is declared outside of all modules, usually at the top of the program
- Should be avoided because:
  - They make debugging difficult
  - Making the module dependent on global variables makes it hard to reuse module in other programs
  - They make a program hard to understand



## Global Variables & Global Constants (cont.)

- A **global constant** is a named constant that is available to every module in the program.
  - Also declared outside all modules, usually at the top of the program
- Since a program cannot modify the value of a constant, these are safer than global variables.