Programming Logic and Design

Using Methods
Objectives

• Review how to use a simple method with local variables and constants
• Create a method that requires a single parameter
• Create a method that requires multiple parameters
• Create a method that returns a value
• Pass an array to a method
Objectives (continued)

- Overload methods
- Learn how to avoid ambiguous methods
- Use prewritten, built-in methods
- Create an IPO chart
- Learn to reduce coupling and increase cohesion
Review of Simple Methods

- **Method**: programming module that contains a series of statements to perform a task
  - Can invoke a method from another program or method
  - Program can contain unlimited number of methods
  - Method can be called unlimited number of times

- **Method must include:**
  - Header
  - Body
  - Return statement
Review of Simple Methods (continued)

- Simple methods require no **arguments** and send no **return values**
- Variables are **in scope** (**local**) to the method in which they are declared
- Two or more methods may access the same data
  - Pass data from one method to another
Figure 7-1 A program that calculates the user’s weight on the moon

```
start
    num weight
    num MOON_FACTOR = 0.166
    num moonWeight
    printInstructions()
    get weight
    moonWeight = weight * MOON_FACTOR
    print moonWeight
stop

printInstructions()
    num langCode
    string ENGLISH_PROMPT = "Please enter your weight in pounds >> "
    string SPANISH_PROMPT = "Por favor entre en su peso en libras >> "
    print "1 = English or 2 = Español >> "
    get langCode
    if langCode = 1 then
        print ENGLISH_PROMPT
    else
        print SPANISH_PROMPT
return
```
Figure 7-2  Output of moon weight calculator program in Figure 7-1
Creating Methods That Require a Single Parameter

- Some methods require outside information
- **Parameters**: communications received by a method
- **Argument**: value in parentheses when calling a method
- **Implementation hiding**: encapsulation of method details
  - Program need not know how method works internally
- **Interface to the method** is the only part of the method the **client** sees
Figure 7-3  Moon weight program that passes an argument to a method
start
    num languageCode
    num weight
    num MOON_FACTOR = 0.166
    num moonWeight
    print "1 - English or 2 - Español >> "
    get languageCode
    printInstructions(languageCode)
    get weight
    moonWeight = weight * MOON_FACTOR
    if languageCode = 1 then
        print "At ", weight,
        " earth pounds, you would weigh ", moonWeight,
        " pounds on the moon."
    else
        print "En ", weight,
        " libras de al tierra, cargarias ", moonWeight,
        " libras en la luna."
    endif
stop

printInstructions(num langCode)
    string ENGLISH_PROMPT = "Please enter your weight in pounds >> "
    string SPANISH_PROMPT = "Por favor entre en su peso en libras >> "
    if langCode = 1 then
        print ENGLISH_PROMPT
    else
        print SPANISH_PROMPT
    return

Figure 7-3  Moon weight program that passes an argument to a method (continued)
Figure 7-4  Typical execution of moon weight program in Figure 7-3
Creating Methods That Require a Single Parameter (continued)

• Method declaration must include
  – Type of the parameter
  – Local name for the parameter
• Parameters in parentheses hold values that are “dropped in” to the method
• Variable declared in method header is a local variable
  – Goes out of scope when method ends
• Variables passed to a method are passed by value
  – Copy of the value sent to the method
    • Stored in a new memory location
Figure 7-5 Alternate version of the moon weight program in which the programmer uses the same identifier for two variables
Figure 7-5  Alternate version of the moon weight program in which the programmer uses the same identifier for two variables (continued)
Creating Methods That Require Multiple Parameters

• Method can require more than one parameter
• List data types and local identifiers in method header
• Parameters are separated by commas in the header
• Any number of parameters in any order
• When calling a method, arguments must match in order and in type
Figure 7-6 A program that calls a `computeTax()` method that requires two parameters
Creating Methods That Return Values

- Variables declared within a method go out of scope when method ends
- Retain a value that exists in a method, return the value from the method
- Methods that return a value must have a **return type**
- Return type can be any data type
- **Void method** returns nothing
Figure 7-7  A payroll program that calls a method that returns a value
Figure 7-8 A program that uses a method’s returned value without storing it
Creating Methods That Return Values (continued)

• Method’s return type is known as the method’s type
• Value in a return statement sent from called method to calling method
• Method’s type must match return type
• Most programming languages allow multiple return statements
Figure 7-9  Unrecommended approach to returning one of several values
Figure 7-10  Recommended approach to returning one of several values

```plaintext
num findLargest(num first, num second, num third)
num largest
if first > second AND first > third
    largest = first
else
    if second > third
        largest = second
    else
        largest = third
endif
endif
return largest
```
Passing an Array to a Method

• Array: list of elements
  – Single element used in same manner as single variable of the same type
• Single array element passed by value
  – Copy of array element passed to method
• Entire array passed by reference
• Method receives memory address of the array
  – Accesses values in array elements
• Changes to array elements within method are permanent
Figure 7-11  PassArrayElement program
void tripleTheValue(num oneVal)

    print “In tripleTheValue() method, value is “, oneVal

    oneVal = oneVal * 3

    print "After change, value is “, oneVal

    return

start
    num LENGTH = 4
    num someNums[LENGTH] = 10, 12, 22, 35
    num x
    print “At beginning of the program...”
    x = 0
    while x < LENGTH
        print someNums[x]
        x = x + 1
    endwhile
    x = 0
    while x < LENGTH
        tripleTheValue(someNums[x])
        x = x + 1
    endwhile
    print "At end of the program..."
    x = 0
    while x < LENGTH
        print someNums[x]
        x = x + 1
    endwhile
    stop

void tripleTheValue(num oneVal)
    print “In tripleTheValue() method, value is “, oneVal
    oneVal = oneVal * 3
    print “After change, value is “, oneVal
    return

**Figure 7-11** PassArrayElement program (continued)
Figure 7-12 Output of PassArrayElement program
Figure 7-13 PassEntireArray program
Figure 7-13 PassEntireArray program (continued)
start
num LENGTH = 4
num someNums[LENGTH] = 10, 12, 22, 35
num x
print "At beginning of main() method..."
x = 0
while x < LENGTH
  print someNums[x]
  x = x + 1
endwhile
quadrupleTheValues(someNums)
print "At end of main() method..."
x = 0
while x < LENGTH
  print someNums[x]
  x = x + 1
endwhile
stop

void quadrupleTheValues(num[] vals)
num LENGTH = 4
num x
x = 0
while x < LENGTH
  print "In quadrupleTheValues() method, value is ", vals[x]
  x = x + 1
endwhile
x = 0
while x < LENGTH
  vals[x] = vals[x] * 4
  x = x + 1
endwhile
x = 0
while x < LENGTH
  print "After change, value is ", vals[x]
  x = x + 1
endwhile
return

Figure 7-13 PassEntireArray program (continued)
Figure 7-14  Output of PassEntireArray program
Overloading Methods

• **Overloading**: supplying diverse meanings for a single identifier

• **Examples**:
  – Overloaded English words:
    • “break a window”
    • “break bread”
    • “take a break”
    • “break the bank”
  – Operators such as +

• **Overload a method**: write multiple methods with same name, different parameter lists
Figure 7-15 The `printBill()` method with a numeric parameter
Figure 7-16 The `printBill()` method with two numeric parameters

```java
void printBill(num bal, num discountRate)
    num newBal
    newBal = bal - (bal * discountRate)
    print “Thank you for your order"
    print “Please remit “, newBal
    return
```
Figure 7-17 The `printBill()` method with a numeric parameter and a string parameter
void printBill(num bal, num discountRate, string msg)

num newBal

newBal = bal - (bal * discountRate)

print “Thank you for your order”

print msg

print “Please remit “, newBal

return

void printBill(num bal, num discountRate, string msg)

num newBal

newBal = bal - (bal * discountRate)

print “Thank you for your order”

print msg

print “Please remit “, newBal

return

Figure 7-18 The printBill() method with two numeric parameters and a string parameter
Overloading Methods (continued)

- More natural for methods’ clients to use single method name for similar tasks
- Overloading never required
  - Could create multiple methods with unique identifiers
- Overloading methods does not reduce the work of programming
- Advantage: one appropriate name for all related tasks
Avoiding Ambiguous Methods

• When overloading a method, you risk creating **ambiguous methods**
• If compiler cannot determine which method to use, it generates an error message
• Methods may be valid individually, but not valid when together in the same program
• Provide different parameter lists for methods with the same name
• Illegal methods have identical names and identical parameter lists
Figure 7-19  Program that contains ambiguous method call
start
    num balance
    num discountInDollars
    get balance, discountInDollars
    printBill(balance, discountInDollars)
stop

void printBill(num bal, num discountRate)
    num newBal
    newBal = bal - (bal * discountRate)
    print "Thank you for your order"
    print "Please remit ", newBal
    return

void printBill(num bal, num discountInDollars)
    num newBal
    newBal = bal - discountInDollars
    print "Thank you for your order"
    print "Please remit ", newBal
    return

Figure 7-19 Program that contains ambiguous method call (continued)
Using Prewritten, Built-in Methods

• All modern programming languages contain prewritten methods
• Commonly available built-in methods include mathematical functions
• Use language documentation to determine what built-in methods are available
• To use a built-in method, you need:
  – What the method does in general
  – Method’s name
  – Method’s parameters
  – Method’s return type
Using an IPO Chart

- **IPO chart:**
  - Identifies and categorizes each item within a module as pertaining to input, processing, or output
  - Provides overview of processing steps in module

<table>
<thead>
<tr>
<th>Input</th>
<th>Processing</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>First value</td>
<td>If the first value is smaller than each of the other two, save it as the smallest value; otherwise if the second value is smaller than the third, save it as the smallest value; otherwise save the third value as the smallest value</td>
<td>Smallest value</td>
</tr>
<tr>
<td>Second value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Third value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7-20** IPO chart for the module that finds the smallest of three numeric values
Figure 7-21 Flowchart and pseudocode of `findSmallest()` module
Reducing Coupling and Increasing Cohesion

• Deciding how to break up a program into modules can be difficult

• Placing too many or too few instructions in a single module:
  – Decreases code readability
  – Reduces reliability

• Two rules for organizing program into modules
  – Reduce coupling
  – Increase cohesion
Reducing Coupling

• **Coupling**
  – Measure of the strength of the connection between two program modules
  – Expresses the extent to which information is exchanged by subroutines

• **Tight coupling**
  – When modules excessively depend on each other
  – Makes programs more prone to errors
  – Many data paths to keep track of
  – Many chances for bad data to be passed
  – Many chances that one module alters data needed by another
Reducing Coupling (continued)

• **Loose coupling**
  – Occurs when modules do not depend on other modules

• **Reduce coupling as much as possible to improve:**
  – Ease of writing the program
  – Program maintainability
  – Module reuse

• **Evaluating the amount of coupling**
  – Look at the intimacy between modules and the number of parameters passed between them
Reducing Coupling (continued)

• Tight coupling
  – Least intimate situation: modules have access to the same globally defined variables

• Loose coupling
  – Most intimate way to share data is pass a copy of variables between modules
  – Loosest (best) subroutines pass single arguments only
Increasing Cohesion

• Cohesion
  – Measure of how much a module’s statements serve to accomplish the module’s purpose
• Highly cohesive modules more reliable
• Functional cohesion
  – Occurs when all operations in a module contribute to the performance of a single task
  – Highest level of cohesion
Increasing Cohesion (continued)

- Procedural cohesion
  - Series of steps that use unrelated data
- **Dispatcher module**
  - If module only has procedural cohesion
  - Calls other modules to perform tasks
- **Logical cohesion**
  - Member module performs tasks based on a logical decision
Summary

• Method contains statements to carry out a task
• Variables and constants declared in a method only have scope in that method
• Calling method sends argument to called method
  – Multiple arguments: comma-separated list in header
• Method has a return type or void
• Single array element passed by value
  – Entire array passed by reference
Summary (continued)

• Overloaded method: same name, different parameter lists
• Ambiguous methods: same name, same parameter lists
• All modern programming languages have prewritten methods
• IPO chart: identifies and categorizes each item pertaining to input, processing, or output
• Strive to achieve loose coupling and high cohesion