Chapter 8
(Part 2)

High Level Programming Languages
Chapter Goals

- Define the concepts of a data type and strong typing
- Explain the concept of a parameter and distinguish between value and reference parameters
- Describe two composite data-structuring mechanisms
- Name, describe, and give examples of the three essential ingredients of an object-oriented language
- . . . Some Hands-On
Input/Output Structures

• In our pseudocode algorithms we have used the expressions \textit{Read} and \textit{Write}

• High-level languages view input data as a stream of characters divided into lines
Input/Output Structures

• The **key to the processing is in the data type** that determines how characters are to be converted to a bit pattern (input) and how a bit pattern is to be converted to characters (output).

• We do not give examples of input/output statements because the **syntax is often quite complex** and differs so widely among high-level languages.
A Little Hands On
Hello World

<html>
<body>
<script type="text/javascript">
document.write("Hello World!")
</script>
</body>
</html>
An External JavaScript

<html>
<head>
<script src="xxx.js"></script>
</head>
<body>
</body>
</html>
Declaring Variables

You can create a variable with the `var` statement:

```javascript
var strname = some value
```

You can also create a variable without the `var` statement:

```javascript
strname = some value
```

You can assign a value to a variable like this:

```javascript
var strname = "Hello World!"
```

Or like this:

```javascript
strname = "Hello World!"
```
Control Statements

<script type="text/javascript">
//Write a "Good morning" greeting if
//the time is less than 10

var d=new Date();
var time=d.getHours();
if (time<10)
{
    document.write("<b>Good morning</b>"
}
</script>
Control Structures

- **Control structure** An instruction that determines the order in which other instructions in a program are executed.

- **Structured programming** A programming methodology in which each logical unit of a program should have just one entry and one exit.

- Sequence, selection statements, looping statements, and subprogram statements are control structures.
Selection Statements

- The **if** statement allows the program to **test the state of the program variables using a Boolean expression**

<table>
<thead>
<tr>
<th>Language</th>
<th>if Statement</th>
</tr>
</thead>
</table>
| Ada      | if Temperature > 75 then  
           |   Put(Item => "No jacket is necessary")  
           |   else  
           |   Put (Item => "A light jacket is appropriate");  
           | end if;  |
| VB.NET   | if (Temperature > 75) Then  
           |   MsgBox("No jacket is necessary")  
           | Else  
           |   MsgBox("A light jacket is appropriate")  
           | End if |
| C++      | if (temperature > 75)  
           |   cout << "No jacket is necessary";  
           | else  
           |   cout << "A light jacket is appropriate";  |
| Java     | if (temperature > 75)  
           |   System.out.print("No jacket is necessary");  
           | else  
           |   System.out.print("A light jacket is appropriate");  |
Selection Statements

Figure 8.3
Flow of control of if statement

true

Boolean expression

false

Zero or more statements in sequence

Zero or more statements in sequence

Rest of program or module
<html>
<body>
<script type="text/javascript">
//If the time is less than 10, you will get "Good morning,"
//Otherwise you will get a "Good day" greeting.

var d = new Date()
var time = d.getHours()

if (time < 10)
{
    document.write("Good morning!")
}
else
{
    document.write("Good day!")
}
</script>
</body>
</html>
## Selection Statements

<table>
<thead>
<tr>
<th>Language</th>
<th>if Statement</th>
</tr>
</thead>
</table>
| Ada      | if Temperature > 75 then  
          |   Put(Item => "No jacket is necessary")  
          | else  
          |   Put (Item => "A light jacket is appropriate");  
          | end if; |
| VB.NET   | if (Temperature > 75) Then  
          |   MsgBox("No jacket is necessary")  
          | Else  
          |   MsgBox("A light jacket is appropriate")  
          | End if |
| C++      | if (temperature > 75)  
          |   cout << "No jacket is necessary";  
          | else  
          |   cout << "A light jacket is appropriate"; |
| Java     | if (temperature > 75)  
          |   System.out.print("No jacket is necessary");  
          | else  
          |   System.out.print("A light jacket is appropriate"); |
Selection Statements

If (temperature > 90)
    Write “Texas weather: wear shorts”
Else If (temperature > 70)
    Write “Ideal weather: short sleeves are fine”
Else if (temperature > 50)
    Write “A little chilly: wear a light jacket”
Else If (temperature > 32)
    Write “Philadelphia weather: wear a heavy coat”
Else
    Write “Stay inside”
<html>
<body>
<script type="text/javascript">
var d = new Date()
var time = d.getHours()
if (time < 10)
{
    document.write("<b>Good morning!</b>")
}
else if (time>10 && time<16)
{
    document.write("<b>Good day!</b>")
}
else
{
    document.write("<b>Good evening!</b>")
}
</script>
</body>
</html>
For convenience, many high-level languages include a case (or switch) statement.

Allows us to make multiple-choice decisions easier, provided the choices are discrete.

CASE operator OF

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘+’</td>
<td>Set answer to one + two</td>
</tr>
<tr>
<td>‘-’</td>
<td>Set answer to one – two</td>
</tr>
<tr>
<td>‘*’</td>
<td>Set answer to one * two</td>
</tr>
<tr>
<td>‘/’</td>
<td>Set answer to one / two</td>
</tr>
</tbody>
</table>
<script type="text/javascript">
//You will receive a different greeting based
//on what day it is. Note that Sunday=0,
//Monday=1, Tuesday=2, etc.

var d=new Date()
theDay=d.getDay()
switch (theDay)
{
case 5:
    document.write("Finally Friday")
    break

case 6:
    document.write("Super Saturday")
    break

case 0:
    document.write("Sleepy Sunday")
    break

default:
    document.write("I'm looking forward to this weekend!")
}
</script>
Looping Statements

• The **while** statement is used to repeat a course of action

• Let’s look at two distinct types of repetitions
Looping Statements

- Count-controlled loops
  - Repeat a specified number of times
  - Use of a special variable called a loop control variable

Figure 8.4
Flow of control of while statement
## Looping Statements

- **Count-controlled loops**

<table>
<thead>
<tr>
<th>Language</th>
<th>Count-Controlled Loop with a while Statement</th>
</tr>
</thead>
</table>
| Ada       | `Count := 1; 
while Count <= Limit loop 
  ... 
  Count := Count + 1; 
end loop;` |
| VB.NET    | `Count = 1 
While (count <= limit) 
  ... 
  count = count + 1 
End While` |
| C++/Java  | `count = 1; 
while (count <= limit) 
{ 
  ... 
  count = count + 1; 
}` |
Looping Statements

- **Event-controlled loops**
  - The number of repetitions is controlled by an event that occurs within the body of the loop itself

<table>
<thead>
<tr>
<th>Read a value</th>
<th>Initialize event</th>
</tr>
</thead>
<tbody>
<tr>
<td>While (value &gt;= o)</td>
<td>Test event</td>
</tr>
<tr>
<td>...</td>
<td>Body of loop</td>
</tr>
<tr>
<td>Read a value</td>
<td>Update event</td>
</tr>
<tr>
<td>...</td>
<td>Statement(s) following loop</td>
</tr>
</tbody>
</table>
Looping Statements

- **Event-controlled** loops

  - Set sum to 0
  - Initialize sum to zero
  - Set posCount to 0
  - Initialize event
  - While (posCount <= 10)
    - Read a value
    - Test event
    - If (value > 0)
      - Test to see if event should be updated
      - Set posCount to posCount + 1
      - Update event
      - Set sum to sum + value
      - Add value into sum
  - ... Statement(s) following loop
Looping Statement

<html>
<body>
<script type="text/javascript">
var i=0
while (i<=10)
{
    document.write("The number is " + i)
document.write("<br />")
i=i+1
}
</script>
</body>
</html>
Looping Statement

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is 10
Subprogram Statements

- We can give a section of code a name and use that name as a statement in another part of the program.
- When the name is encountered, the processing in the other part of the program halts while the named code is executed.
Subprogram Statements

- There are times when the calling unit needs to give information to the subprogram to use in its processing.

- A parameter list is a list of the identifiers with which the subprogram is to work, along with the types of each identifier placed in parentheses beside the subprogram name.
Subprogram Statements

(a) Subprogram A does its task and calling unit continues with next statement

Figure 8.5 Subprogram flow of control
(b) Subprogram B does its task and returns a value that is added to 5 and stored in x.
Subprogram Statements

• **Parameters**  Identifiers listed in parentheses beside the subprogram declaration; sometimes they are called *formal parameters*

• **Arguments**   Identifiers listed in parentheses on the subprogram call; sometimes they are called *actual parameters*
Subprogram Statements

• **Value parameter** A parameter that expects a copy of its argument to be passed by the calling unit (put on the message board)

• **Reference parameter** A parameter that expects the address of its argument to be passed by the calling unit (put on the message board)
### Subprogram Declaration

<table>
<thead>
<tr>
<th>Language</th>
<th>Subprogram Declaration</th>
</tr>
</thead>
</table>
| **VB.NET** | Public Sub Example(ByVal one As Integer, ByVal two As Integer, ByRef three As Single)  
                 ...  
                 End Sub |
| **C++/Java** | void Example(int one; int two; float& three)  
                      {  
                      ...  
                      } |
Functions

```html
<html>
<head>
<script type="text/javascript">
function displaymessage() {
alert("Hello World!")
}
</script>
</head>

<body>
<form>
<input type="button" value="Click me!" onclick="displaymessage()">
</form>
</body>
</html>
```
Recursion

- **Recursion**  The ability of a subprogram to call itself

- Each recursive solution has **at least two cases**
  - **Base case**  The case to which we have an answer
  - **General case**  The case that expresses the solution in terms of a call to itself with a smaller version of the problem

- For example, the factorial of a number is defined as the number times the product of all the numbers between itself and 0:

  \[ N! = N \times (N - 1)! \]
Asynchronous Processing

- **Asynchronous processing** The concept that input and output can be accomplished through windows on the screen
  - *Clicking* has become a major form of input to the computer
  - Mouse clicking is not within the sequence of the program
  - A user can click a mouse at any time during the execution of a program
  - This type of processing is called **asynchronous**
Composite Data Types

- **Records**
  - A record is a named *heterogeneous* collection of items in which individual items are accessed by name
  - The *elements* in the collection *can be of various types*
## Composite Data Types

<table>
<thead>
<tr>
<th>Language</th>
<th>Record Type Declaration</th>
</tr>
</thead>
</table>
| Ada      | type Name_String is String (1..10);  
            type Employee_Type is  
                record  
                    Name : Name_String;  
                    Age : Integer range 0..100;  
                    Hourly_Wage : Float range 1.0..5000.0;  
                end record; |
| VB.NET   | Structure Employee  
            Dim Name As String  
            Dim Age As Integer  
            Dim HourlyWage As Single  
            End Structure |
| C++      | struct EmployeeType  
            {  
                string name;  
                int age;  
                float hourlyWage;  
            } |
## Composite Data Types

<table>
<thead>
<tr>
<th>Language</th>
<th>Record Variable Declaration and Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>An_Employee : Employee_Type;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>An_Employee.Name = &quot;Sarah Gale&quot;;</td>
</tr>
<tr>
<td></td>
<td>An_Employee.Age = 32;</td>
</tr>
<tr>
<td></td>
<td>An_Employee.Hourly_Wage = 95.00;</td>
</tr>
<tr>
<td>VB.NET</td>
<td>Dim AnEmployee As EmployeeType</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>AnEmployee.Name = &quot;Sarah Gale&quot;</td>
</tr>
<tr>
<td></td>
<td>AnEmployee.Age = 32</td>
</tr>
<tr>
<td></td>
<td>AnEmployee.HourlyWage 95.00</td>
</tr>
<tr>
<td>C++</td>
<td>EmployeeType anEmployee;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>anEmployee.name = &quot;Sarah Gale&quot;;</td>
</tr>
<tr>
<td></td>
<td>anEmployee.age = 32;</td>
</tr>
<tr>
<td></td>
<td>anEmployee.hourlyWage = 95.00;</td>
</tr>
</tbody>
</table>
Arrays

- An **array** is a named collection of **homogeneous** items in which individual items are accessed by their place within the collection.
  - The **place** within the collection is called an **index**.

<table>
<thead>
<tr>
<th>Language</th>
<th>Array Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>type Index_Range is range 1..10;</td>
</tr>
<tr>
<td></td>
<td>type Ten_Things is array (Index_Range) of Integer;</td>
</tr>
<tr>
<td>VB.NET</td>
<td>Dim TenThings(10) As Integer</td>
</tr>
<tr>
<td>C++/Java</td>
<td>int tenThings[10];</td>
</tr>
</tbody>
</table>
Arrays

Figure 8.8
Array variable tenThings accessed from 0..9
Functionality of Object-Oriented Languages

- Encapsulation
- Inheritance
- Polymorphism
Encapsulation

- **Encapsulation**  A language feature that enforces information hiding

- **Class**  A language construct that is a pattern for an object and provides a mechanism for encapsulating the properties and actions of the object class

- **Instantiate**  Create an object from a class
Inheritance

- **Inheritance**  A construct that *fosters reuse* by allowing an application to take an already-tested class and *derive a class* from it that *inherits the properties* the application needs.

- **Polymorphism**  The ability of a language to have *duplicate method names* in an inheritance hierarchy and to *apply the method* that is *appropriate for the object* to which the method is applied.
Inheritance

Inheritance and polymorphism combined allow the programmer to build useful hierarchies of classes that can be reused in different applications.

Figure 8.9
Mapping of problem into solution
Homework

- Read Chapter Eight, Sections 8.3 – 8.4

- “PLAY” with JavaScript
  http://www.w3schools.com/js/js_howto.asp

- Do some of the hands-on examples in class
Mid-Term

- Due Back: **Tonight**
- Sorry For The Web Site Outage!!!
No Class

- There Will Be **No Class On Monday, 10/30**

- **Next Class Is Wednesday, 11/1**
Have A Great Weekend