Chapter Goals

• Describe the translation process and distinguish between assembly, compilation, interpretation, and execution

• Name four distinct programming paradigms and name a language characteristic of each

• Describe the following constructs: stream input and output, selection, looping, and subprograms

• Construct Boolean expressions and describe how they are used to alter the flow of control of an algorithm
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
Compilers

- **Compiler** A program that translates a high-level language program into machine code
- High-level languages provide a richer set of instructions that makes the programmer’s life even easier
Figure 8.1 Compilation process
**Interpreters**

- **Interpreter** A translating program that translates and executes the statements in sequence
  
  – Unlike an assembler or compiler which produce machine code as output, which is then executed in a separate step
  
  – An interpreter translates a statement and then immediately executes the statement
  
  – Interpreters can be viewed as *simulators*
Java

- Introduced in 1996 and swept the computing community by storm
- Portability was of primary importance
- Java is compiled into a standard machine language called **Bytecode**
- A software interpreter called the JVM (Java Virtual Machine) takes the Bytecode program and executes it
What is a paradigm?

A set of assumptions, concepts, values, and practices that constitute a way of viewing reality.
Figure 8.2
Portability provided by standardized languages versus interpretation by Bytecode
Programming Language Paradigms

(b) Java program
compiled into
Bytecode and run
on different systems

Figure 8.2
Portability provided by standardized languages versus interpretation by Bytecode
Programming Language Paradigms

• Imperative or procedural model
  – FORTRAN, COBOL, BASIC, C, Pascal, Ada, and C++

• Functional model
  – LISP, Scheme (a derivative of LISP), and ML
• Logic programming
  – PROLOG

• Object-oriented paradigm
  – SIMULA and Smalltalk
  – C++ is as an imperative language with some object-oriented features
  – Java is an object-oriented language with some imperative features
Functionality of Imperative Languages

- **Sequence** Executing statements in sequence until an instruction is encountered that changes this sequencing
- **Selection** Deciding which action to take
- **Iteration** (looping) Repeating an action

Both selection and iteration require the use of a Boolean expression
Boolean Expressions

- **Boolean expression** A sequence of identifiers, separated by compatible operators, that evaluates to *true* or *false*

- Boolean expression can be
  - A Boolean variable
  - An arithmetic expression followed by a relational operator followed by an arithmetic expression
  - A Boolean expression followed by a Boolean operator followed by a Boolean expression
Boolean Expressions

- **Variable**: A location in memory that is referenced by an identifier that contains a data value

  Thus, a Boolean variable is a location in memory that can contain either *true* or *false*
Boolean Expressions

• A relational operator between two arithmetic expressions is asking if the relationship exists between the two expressions.

• For example, $xValue < yValue$

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal to</td>
<td>= or ==</td>
</tr>
<tr>
<td>not equal to</td>
<td>&lt;&gt; or != or /=</td>
</tr>
<tr>
<td>less than or equal to</td>
<td>&lt;=</td>
</tr>
<tr>
<td>greater than or equal to</td>
<td>&gt;=</td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
</tr>
</tbody>
</table>
Strong Typing

- **Strong typing** The requirement that only a value of the proper type can be stored into a variable

- **Data type** A description of the set of values and the basic set of operations that can be applied to values of the type
Data Types

- Integer numbers
- Real numbers
- Characters
- Boolean values
- Strings
Integers

- The range varies depending upon how many bytes are assigned to represent an integer value.
- Some high-level languages provide several integer types of different sizes.
- Operations that can be applied to integers are the standard arithmetic and relational operations.
Reals

• Like the integer data type, the range varies depending on the number of bytes assigned to represent a real number

• Many high-level languages have two sizes of real numbers

• The operations that can be applied to real numbers are the same as those that can be applied to integer numbers
Characters

- It takes one byte to represent characters in the ASCII character set.
- Two bytes to represent characters in the Unicode character set.
- Our English alphabet is represented in ASCII, which is a subset of Unicode.
- Applying arithmetic operations to characters doesn’t make much sense

- Comparing characters does make sense, so the relational operators can be applied to characters

- The meaning of “less than” and “greater than” when applied to characters is “comes before” and “comes after” in the character set
The Boolean data type consists of two values: true and false

Not all high-level languages support the Boolean data type

If a language does not, then you can simulate Boolean values by saying that the Boolean value true is represented by 1 and false is represented by 0
Strings

• A string is a sequence of characters considered as one data value

• For example: “This is a string.”
  – Containing 17 characters: one uppercase letter, 12 lowercase letters, three blanks, and a period

• The operations defined on strings vary from language to language
  – They include concatenation of strings and comparison of strings in terms of lexicographic order
• **Declaration** A statement that associates an identifier with a variable, an action, or some other entity within the language that can be given a name so that the programmer can refer to that item by name.
## Declarations

<table>
<thead>
<tr>
<th>Language</th>
<th>Variable Declaration</th>
</tr>
</thead>
</table>
| Ada      | sum : Float := 0: --set up word with 0 as contents  
num1: Integer; --set up a two-byte block for num1  
num2: Integer; --set up a two-byte block for num2  
num3: INTEGER; --set up a two-byte block for num3  
...  
um1 := 1; |
| VB.NET   | Dim sum As Single = 0.0F ' set up word with 0 as contents  
Dim num1 As Integer ' set up a two-byte block for num1  
Dim num2 As Integer ' set up a two-byte block for num2  
Dim num3 As Integer ' set up a two-byte block for num3  
...  
um1 = 1 |
| C++/Java | float sum = 0.0; // set up word with 0 as contents  
int num1; // set up a block for num1  
int num2; // set up a block for num2  
int num3; // set up a block for num3  
...  
um1 = 1; |
• **Reserved word**  A word in a language that has special meaning

• **Case-sensitive**  Uppercase and lowercase letters are considered the same
Assignment statement

• **Assignment statement** An action statement (not a declaration) that says to evaluate the expression on the right-hand side of the symbol and store that value into the place named on the left-hand side.

• **Named constant** A location in memory, referenced by an identifier, that contains a data value that cannot be changed.
## Constant Declaration

<table>
<thead>
<tr>
<th></th>
<th>Ada</th>
<th>VB.NET</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Comma : constant Character := ',,';</td>
<td>Const WORD1 As Char = &quot;&quot;,&quot;c</td>
<td>const char COMMA = ',,';</td>
<td>final char COMMA = ',,';</td>
</tr>
<tr>
<td></td>
<td>Message : constant String := &quot;Hello&quot;;</td>
<td>Const MESSAGE As String = &quot;Hello&quot;</td>
<td>const string MESSAGE = &quot;Hello&quot;;</td>
<td>final String MESSAGE = &quot;Hello&quot;;</td>
</tr>
<tr>
<td></td>
<td>Tax_Rate : constant Float := 8.5;</td>
<td>Const TaxRate As Double = 8.5</td>
<td>const double TAX_RATE = 8.5;</td>
<td>final double TAX_RATE = 8.5;</td>
</tr>
</tbody>
</table>
Input/Output Structures

- In our pseudocode algorithms we have used the expressions *Read* and *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
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
## 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"); |
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”
case Statement

- 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**

- ‘+’ : Set answer to one + two
- ‘-’ : Set answer to one − two
- ‘*’ : Set answer to one * two
- ‘/’ : Set answer to one / two

Page 246
Looping Statements

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

• Let’s look at two distinct types of repetitions
**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;= 0)</td>
<td>Test event</td>
</tr>
<tr>
<td></td>
<td>Body of loop</td>
</tr>
<tr>
<td></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 $0$  
Set posCount to $0$  
While (posCount $\leq$ 10)  
Read a value  
If (value $> 0$)  
    Set posCount to posCount + 1  
    Set sum to sum + value  
...  
Initialize sum to zero  
Initialize event  
Test event  
Test to see if event should be updated  
Update event  
Add value into sum  
Statement(s) following loop
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.
(a) Subprogram A does its task and calling unit continues with next statement

Figure 8.5 Subprogram flow of control
Subprogram Statements

(b) Subprogram B does its task and returns a value that is added to 5 and stored in $x$

Figure 8.5 Subprogram flow of control
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)
<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)  
|           | {                      
|           | ...                    
|           | }                      |
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**
• 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

<table>
<thead>
<tr>
<th></th>
<th>1066</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1492</td>
</tr>
<tr>
<td>2</td>
<td>1668</td>
</tr>
<tr>
<td>3</td>
<td>1945</td>
</tr>
<tr>
<td>4</td>
<td>1972</td>
</tr>
<tr>
<td>5</td>
<td>1510</td>
</tr>
<tr>
<td>6</td>
<td>999</td>
</tr>
<tr>
<td>7</td>
<td>1001</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>2001</td>
</tr>
</tbody>
</table>

**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