C#
Control Structures/Statements
Lecture 7
Flow Control Structures

- The order in which statements are executed.
- There are four structures.
  1. Sequence Control Structure
  2. Selection Control Structure
     - Also referred to as branching (if and if-else)
  3. Case Control Structure (switch)
  4. Repetition Control Structure (loops)
1. Sequence Control Structure

- The order statements are placed (sequenced)
  ```csharp
  //input
  int intQty = int.Parse(Console.ReadLine());
  decimal decPrice = decimal.Parse(Console.ReadLine());
  //process
  decimal decSubtotal = intQty * decPrice;
  //output
  txtSubtotal.Text = decSubtotal.ToString("N");
  ```

- The only way to display subtotal, statements must be in this order.
Flowchart – Sequence Control

Begin

Input
price, qty

subtotal = price * qty

Output
subtotal

End
2. Selection Control (If)

- Use if statements to determine if a set of statements should be executed.

\[ \text{decSubtotal} = \text{intQty} \times \text{decPrice} \]

\[
\text{if} \ (\text{chkSalesTax} = = \text{true}) \\
\text{decSalesTax} = \text{decSubtotal} \times \text{cdecTAX\_RATE} ;
\]
Flowchart – If Statement

Subtotal = qty * price

If taxable

True

Sale tax = subtotal * Tax Rate

False

Output sales tax
Selection Control (If-Else)

- Use if else statements when there are several options to choose from.

```csharp
if (radNextDay.Checked == true)
    decShipping = cdecNEXT_DAY_SHIPPING_RATE;
else
    decShipping = cdecTHREE_DAY_SHIPPING_RATE;
```

- After if else statement, decShipping will have a value.
Flowchart – If-Else Statement

User selects Shipping

If Next Day

True

Shipping = Next Day

False

Shipping = Three Day

Output Shipping
Condition is an Expression

- Conditions evaluate to true or false.

```java
if (condition)
    true – 1 or more statements
else
    false – 1 or more statements
```
Block of code – Compound Statements

- Use braces to create a block of statements.
- Single statements do NOT require braces.

```c
if (intHours > 40)
{
    decRegularPay = 40 * decPayRate;
    decOvertimePay = (intHours - 40) *
        (decPayRate * cdec_OVERTIME_RATE);
}
else
{
    decRegularPay = intHours * decPayRate;
    decOvertimePay = 0;
}

decGrossPay = decRegularPay + decOvertimePay;
```
Incorrect if-else

```c
if (intHours > 40) {
    decRegularPay = 40 * decPayRate;
    decOvertimePay = (intHours - 40) * (decPayRate * cdec_OVERTIME_RATE);
}
else
    decRegularPay = intHours * decPayRate;
    decOvertimePay = 0;

decGrossPay = decRegularPay + decOvertimePay;
```

- `decOvertimePay` would always be set to zero, because there is a single statement for else since braces are not included.
Nested If Statements

First if has to be true in order to continue.

```csharp
if (intQuantity >= 1)
    if (decPrice >= 5.00M)
        decSubtotal = intQty * decPrice;
    else
        {
            MessageBox.Show("Invalid price entered");
            txtPrice.Focus();
            txtPrice.SelectAll();
        }
else
    {
        MessageBox.Show("Invalid quantity entered");
        txtQuantity.Focus();
        txtQuantity.SelectAll();
    }
```
Matching an Else to If

- How are else statements matched with an if?
  - Compiler works it’s way back. When an else is encountered, it looks back to find an if that has not been matched to an else.

- Why do we indent each level?
  - We indent to make programs easier to read. Indenting has no effect on how compiler matches an else to an if.
Selection Control (else if) Multiway Branching

- Compare each condition from top to bottom, block of code for 1\textsuperscript{st} true condition is executed, and then skip to the statement following the if-else statements.

```csharp
if (radNextDay.Checked == true)  
decShipping = cdecNEXT_DAY_SHIPPING_RATE;  
else if (radThreeDay.Checked == true)  
decShipping = cdecTHREE_DAY_SHIPPING_RATE;  
else  
decShipping = cdecGROUND_SHIPPING_RATE;
```
Relational Operators

= = Equal to
!= Not Equal to
< Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to

Relational Operators are used to form conditions, and conditions can involve constants, variables, numeric operators, and functions.
Assignment (=) vs Comparison (==)

- if (x = 12) // may generate syntax error.
- if (x == 12) // comparison

- The error message generated will have to do with an implicit data conversion to bool.
- If you get this error, check that you used 2 equal signs.
Logical Operators

- Compound or Complex Conditions can be formed using Logical Operators.
  - `&&` – And, both conditions must be true
  - `||` – Or, either condition must be true
  - `!` – Not, reverses the resulting condition
&& - And Logical Operator

- Both conditions must be true in order for the entire condition to be true.

```csharp
if (intQty > 0 && intQty < 51)
    decSubtotal = intQty * decPrice;
else
    MessageBox.Show("Enter a value between 1 and 50");
```

- What happens if qty = 25?
- What happens if qty = 60?
Use Parentheses

- Use parentheses to clarify or group conditions.
- All conditions must be enclosed with outer parentheses.

```c
if (((intQty > 0) && (intQty < 51)))
    decSubtotal = intQty * decPrice;
else
    MessageBox.Show("Enter a value between 1 and 50");
```
### Or Logical Operator

- Either condition must be true for the entire condition to be true.
- The pipe character is located above the Enter key; must shift to select it, and two of them are entered next to each other.

```csharp
if ((intQty < 1) || (intQty > 50))
    MessageBox.Show("Enter a value between 1 and 50");
else
    decSubtotal = intQty * decPrice;
```

- What happens if qty = 25?
- What happens if qty = 60?
Pipe Character - |

- Where is the pipe character on the keyboard?
  - On most keyboards
    - It is right above the Enter key
    - Shares the key with the back slash - \n    - Must hold the shift key go get it
    - Instead of a solid line, it is shown as a broken line

- For the Or operator,
  2 pipe characters must be entered - | |.

- For the And operator,
  2 ampersands characters must be entered - &&.
! – Not Logical Operator

- Not operator makes the condition the opposite of what it evaluated to...
- If condition is true, Not makes it false.
- If condition is false, Not makes it true.

```java
if ! (intQty == 50)
    true    intQty <> 50
else
    false   intQty = 50
```

- Confusing, try not to use Not.
Practice Exercises

count = 0  limit = 10  min = 5

if ((count == 0) && (limit < 20)) True
if (count == 0 && limit < 20) True
if (limit > 20 || count < 5) True
if ! (count == 12) True
if ((count == 10) || (limit > 5 && min < 10)) True
Comparing Boolean’s

- Booleans are variables that are equal to either True or False.
- Some properties are also True or False.
- The following condition will return True if the radio button for Next Day was checked:
  
  ```csharp
  if (radNextDay.Checked == true)
  ```

- The shortcut to the above condition is to leave `== true` off as follows:
  
  ```csharp
  if (radNextDay.Checked)
  ```

- We should usually express the desired condition (`== true`) to avoid bugs and confusion.
Comparing Strings

- String variables, properties, or literals can also be compared.
- Only (==) and (!=) can be used with strings.
- Can use CompareTo method – more on this later.
- Strings are compared from left to right.
- The characters’ binary value is used to determined which is greater or less than.
- This means that capitalization is considered.
- ASCII Code Table lists the binary values.
ToUpper and ToLower Methods

- Use ToUpper and ToLower methods of the String class to convert strings for comparisons.

```csharp
if txtState.Text.ToUpper() == “CA”
    decSalesTax = decSubtotal * cdecTAX_RATE;
```

- When we convert strings we only need to test for one condition, instead of every possible condition – ca, CA, Ca, cA.
3. Case Control Structure (switch)

- Switch statement is an efficient decision-making structure that simplifies choosing among several actions.
- Works good for a list of options (menus)
- It avoids complex nested If constructs.
- Just about all switch statements can be stated using an if block.
- Be sure to handle each possible case.
- Use default: to catch errors.
- A break statement is required at the end of each case.
switch Example

switch (strDiscType.ToUpper)
{
    case "E": //Employee
        decDiscountRate = .05;
        break;
    case "S": //Student
        decDiscountRate = .10;
        break;
    case "C": //Senior
        decDiscountRate = .15;
        break;
    default: //No discount
        decDiscountRate = .00;
        break;
}

Tahir Nawaz
Assistant Professor
UOL Sargodha Campus
Switch Expression

- The matching case is determined by the value of the expression.
- The expression can be a variable, property, operator expression, or method.

```java
switch (expression)
{
    case 1: ...
    case 2: ...
}
```
C# Loops Syntax

- Loop Statements

  - **while**

    ```csharp
    int i = 0;
    while ( i < 5 )
    {
      Console.WriteLine ( i );
      ++i;
    }
    ```

    The above loop repeats 5 times and prints the value of i.
    The output of above code would be like this:

    ```
    0
    1
    2
    3
    4
    ```
C# Syntax

- **for**
  ```csharp
  for ( int i = 0; i < 5; i++ )
  {
    Console.WriteLine ( i );
  }
  ```

  The above loop repeats 5 times just like the **while** loop and prints the value of i. The output of above code would be like this:

  - 0
  - 1
  - 2
  - 3
  - 4
C# Syntax

- **do ... while**
  
  ```csharp
  int i = 0;
  do
  {
    Console.WriteLine ( i );
    i++;
  }
  while ( i < 5 );
  
  The above loop is pretty much same as the **while** loop. The only difference is, the condition is checked only after executing the code inside the loop.
C# Syntax

- **foreach**

```csharp
string[] names = new string[] { "Ali", "Kamal", "Jamal", "Jameel" };
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

- foreach loop can be used to iterate through a collection like array, ArrayList etc.

The above code displays the following output:

- Ali
- Kamal
- Jamal
- Jameel
C# Syntax

- Conditional Operators

  if ... else

  This is the conditional operator, used to selectively execute portions of code, based on some conditions.

  ```csharp
  string name = "Kamal";
  if ( name == "Jameel" )
  {
    Console.WriteLine( "you are in 'if' block" );
  }
  else
  {
    Console.WriteLine( "you are in 'else' block" );
  }
  
  in the above case, it prints:

  you are in 'else' block
C# Syntax

- Flow Control Statements

**break**

'break' statement is used to break out of loops ('while', 'for', switch' etc).

```csharp
string[] names = new string[] { "Ali", "Kamal", "Jamal", "Jameel" };  
foreach (string name in names)
{
    Console.WriteLine(name);
    if (name == "Ali")
        break;
}
```

In the above sample, it iterates through the array of 4 items, but when it encounters the name "Ali", it exits the loop and will not continue in the loop anymore.

The output of above sample would be:

Jamal
Ali
C# Syntax

'C continue' statement is also used to in the loops ('while', 'for' etc). When executed, 'continue' statement will move the execution to the next iteration in the loop, without continuing the lines of code after the 'continue' inside the loop.

```csharp
string[] names = new string[] { "Ali", "Kamal", "Jamal", "Jameel" };
foreach (string name in names)
{
    if (name == "Ali")
        continue;
    Console.WriteLine(name);
}
```

In the above sample, when the value of name is "Ali", it executes the 'continue' which will change the execution to the next iteration, without executing the lines below it. So, it will not print the name, if the name is "Ali".
The output of above sample would be:

Ali
Jameel
Kamal
C# Syntax

- **switch**

  if you have to write several if...else conditions in your code, switch statement is a better way of doing it. The following sample is self explanatory:

```csharp
int i = 3;
switch (i)
{
    case 5:
        Console.WriteLine( "Value of i is : " + 5 );
        break;
    case 6:
        Console.WriteLine( "Value of i is : " + 6 );
        break;
    case 3:
        Console.WriteLine( "Value of i is : " + 3 );
        break;
    case 4:
        Console.WriteLine( "Value of i is : " + 4 );
        break;
    default:
        Console.WriteLine( "Value of i is : " + i );
        break;
}
```
Debugging Applications in C#

- One of the most important things in any programming language and environment is providing the ability to find errors, also called bugs, in your programs.
- In C#, there are different types of errors that can occur during the coding and execution of programs.
- The process of location and fixation of these bugs are known as “Debugging” process in application.
Origin of the term “Bug”?

- The most common story, though one that is disputed by some, is that in one of the original computers, the Mark I, the programmers were having a hard time with the system when they found out what the problem was.

- It was, literally, a bug i.e. a moth that was caught inside the system. From there on whenever a problem occurred

- So the programmers said there was a bug in the system.
What is Debugging?

- Debugging is how to find errors and mistakes in your code.
- Perhaps you forgot to declare a variable, and then you must fix the resulting errors and mistakes, if possible.
- If programmer didn’t have the ability to locate errors, he/she wouldn’t be able to create very stable applications.
- C# Express includes many tools for debugging your applications that make life much easier.
Types of Errors

- There are actually three main categories of errors that can occur in your applications.
  - **Syntax errors.** The most basic type error, a syntax error is an error in the language.
  - A missing curly bracket (used to begin and end code blocks) or an incorrectly formatted line.
  - It’s basically something that the compiler can’t understand.
  - These errors will pretty much always be displayed in the Error List window.
Types of Errors

- **Semantic/logical errors**
- These errors are harder to find, because they won’t get read as an error.
- Because they aren’t a mistake in the language, but rather something that means something other than what you intended, or are caused by data the user entered.
- These errors raised when programmer runs their programs, but you won’t necessarily know what is wrong or where it is in the code.
Types of Errors

- **Exception errors.** There are some errors that occur during runtime and those errors are uncontrollable and stop the application abruptly.
- These errors are called *exceptions*, and while they can be trapped and programmed for, because they occur at runtime in the “real world,” they will occur regardless of what you do to plan for them.
- Logical and semantic errors sometimes cause exceptions because the logic fails due to bad data or some other cause.
Exceptions

- An exception is any error condition or unexpected behavior encountered by an executing program.
- Exceptions can be raised because of a fault in your code, by unavailable operating system resources, or other unexpected circumstances.
- You can use Structured Error Handlers to recognize runtime errors as they occur in a program, suppress unwanted error messages, and adjust program conditions so that your application can regain control and continue running.
Exceptions in Windows Forms Application

- Run the application (double-click the exe file)
  - An Exception message box appears:
    - Click the Details button to check the details about the error
    - Click the Quit button to exit the application, or
    - Click the Continue button so that your application can regain control and run onward
In the .NET Framework, an exception is an object that inherits from the Exception Class.

- **Properties:**
  - **StackTrace**
    - It contains a stack trace that can be used to determine where an error occurred (includes the source file name and program line number)
  - **Message**
    - It provides details about the cause of an exception
## Exception Hierarchy

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception</td>
<td>Base Class</td>
</tr>
<tr>
<td>SystemException</td>
<td>Base class for all runtime-generated errors.</td>
</tr>
<tr>
<td>IOException</td>
<td>when an I/O error occurs.</td>
</tr>
<tr>
<td>FileNotFoundException</td>
<td>when an attempt to access a file that does not exist on disk fails.</td>
</tr>
<tr>
<td>IndexOutOfRangeException</td>
<td>when an array is indexed improperly.</td>
</tr>
<tr>
<td>ArithmeticException</td>
<td>Errors in an arithmetic, casting, or conversion operation</td>
</tr>
<tr>
<td>DivideByZeroException</td>
<td>when an attempt to divide an integral or decimal value by zero</td>
</tr>
<tr>
<td>NullReferenceException</td>
<td>when a null object is referenced.</td>
</tr>
</tbody>
</table>

The diagram below illustrates the exception hierarchy.
No Exception Handling

• An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates
  ◦ If an exception occurs, the exception is propagated back to the calling method, or the previous method
  ◦ If it also has no exception handler, the exception is propagated back to that method's caller, and so on
  ◦ If it fails to find a handler for the exception, an error message is displayed and the application is terminated

• Example:

```csharp
int[] a = {1, 2, 3};
int x = a[4]; // generates run-time error
Console.WriteLine(x);
```

• By placing exception handling code in your application, you can handle most of the errors users may encounter and enable the application to continue running
Try-catch block

- **Try block**
  - Place the code that might cause the exception in a try block
  - When an error happens, the .NET system ignores the rest of the code in the try block and jumps to the catch block

- **Catch block**
  - Specify the exception that you wish to catch or catch the general exception. (Since all exceptions are subclasses of the Exception class, you can catch all exceptions this way. In this case, exceptions of all types will be handled in the same way.)
  - **Execute** the code if the exception is thrown
  - **Skip** the code if no exception

- **Statements after the catch block**
  - **Execute** if either the exception is not thrown or if it is thrown

```csharp
try {
    int[] a = { 1, 2, 3 };
    x = a[4];
    Console.WriteLine("This will not be printed");
} catch (Exception ex){
    x = -1;
    Console.WriteLine(ex.Message);
}
Console.WriteLine("x=" + x + "");
```

Index was outside the bounds
...
x=-1
Handling Multiple Catch Clauses

The catch block is a series of statements beginning with the keyword `catch`, followed by an exception type and an action to be taken:

- Each catch block is an exception handler and handles the type of exception indicated by its argument.
- The runtime system invokes the exception handler when the handler is the first one matching the type of the exception thrown.
- It executes the statement inside the matched catch block (the other catch blocks are bypassed) and continues after the try-catch block.

```csharp
try {
    String s = "a";
    int x = Convert.ToInt32(s);
} catch (FormatException ex) {
    Console.WriteLine(ex.Message);
} catch (Exception ex){
    Console.WriteLine("General Exception");
}
Console.WriteLine("Finished");
```

Input string was not in a correct format.
Finished
Handling Multiple Catch (con’t)

- Exceptions are arranged in an inheritance hierarchy.
  - A catch specifying an Exception near the top of the hierarchy (a very general Exception) will match any Exception in the subtree.
  - Note: Exception subclasses (specific types of exception) must come before any of their superclasses (e.g., the general Exception) in the order of the catch clause. Otherwise, the compiler might issue an error.

```csharp
try {
    String s = "a";
    int x = Convert.ToInt32(s);
} catch (Exception ex) {
    Console.WriteLine(ex.Message);
} catch (FormatException ex) {
    Console.WriteLine("General Exception");
} 
Console.WriteLine("Finished");
```

Compile error

```csharp
try {
    String s = "a";
    int x = Convert.ToInt32(s);
} catch (ArithmeticException ex) {
    Console.WriteLine(ex.Message);
} catch (Exception ex) {
    Console.WriteLine("General Exception");
} 
Console.WriteLine("Finished");
```

General Exception

Finished
You can use nested try-catch blocks in your error handlers

- If an inner try statement does not have a matching catch statement for a particular exception, the next try statement’s catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted and the program terminates.

```csharp
int[] number = { 4, 8, 6, 32 };  
int[] denom = { 2, 0, 2, 4 };  
try {
    for (int i = 0; i < number.Length; i++) {
        try {
            Console.WriteLine(i + ":" + number[i] / denom[i]);
        } catch (ArithmeticException ex) {
            Console.WriteLine("Inner:Can't divide by ZERO");
        }
    }
} catch (Exception ex) {
    Console.WriteLine("Outer:No matching element found.");
}
```
Exception Propagation

- If it is not appropriate to handle the exception where it occurs, it can be handled at (propagated to) a higher level
  - The first method it finds that catches the exception will have its catch block executed. At this point the exception has been handled, and the propagation stops (no other catch blocks will be executed). Execution resumes normally after this catch block.

```csharp
try {
    Console.WriteLine("Starting calls...");
    PropagateException("123");          //this is fine
    PropagateException("a");           //format Exception
    PropagateException("3000000000");  //overflow
    PropagateException("234");         //doesn't happen
}
``` catch (Exception ex)
```csharp
{
    Console.WriteLine("General Exception"); try
    {
        Console.WriteLine("Entering subroutine...");
        int x = Convert.ToInt32(s);
        Console.WriteLine("x={0}",x);
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.WriteLine("Exiting subroutine...");
}
``` public static void PropagateException(string s)
```csharp
{
    Console.WriteLine("General Exception"); try
    {
        Console.WriteLine("Entering subroutine...");
        int x = Convert.ToInt32(s);
        Console.WriteLine("x={0}",x);
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.WriteLine("Exiting subroutine...");
}
```
The Finally block is optional.

It allows for cleanup of actions that occurred in the try block but may remain undone if an exception is caught.

Code within a finally block is **guaranteed to be executed** if any part of the associated try block is executed regardless of an exception being thrown or not.

```csharp
string s = "1";
try {
    int x = Convert.ToInt32(s);
} catch (Exception ex) {
    Console.WriteLine("General Exception");
} finally {
    Console.WriteLine("Finally");
}
Console.WriteLine("Finished");
```

```csharp
string s = "a";
try {
    int x = Convert.ToInt32(s);
} catch (Exception ex) {
    Console.WriteLine(ex.Message);
} finally {
    Console.WriteLine("Finally");
}
Console.WriteLine("Finished");
```

Input string was not in ...
Finally
Finished
Explicitly Throw Exceptions

- You can explicitly throw an exception using the **throw** statement.
- If the exception is unhandled, the program stops immediately after the throw statement and any subsequent statements are not executed.
- The throw statement requires a single argument:
  - The type of exception to be thrown.

```csharp
try {
    if ( i <= 0)
        throw new Exception();
    Console.WriteLine(i);
} catch (Exception ex){
    Console.WriteLine("General Exception");
} finally {
    Console.WriteLine("Finally");
}
```

General Exception
Finally
Finished
The checked keyword

- is used to control the overflow-checking context for integral-type arithmetic operations and conversions.
- Example:
  - if an expression produces a value that is outside the range of the destination type, by default, C# generates code that allows the calculation to silently overflow
  - i.e. you get the wrong answer

```csharp
int number = int.MaxValue;
Console.WriteLine(number);
number++;
Console.WriteLine(number);
```

- Use the `checked` keyword to turn on the integer arithmetic overflow checking (or `unchecked` to turn off)

```csharp
checked {
    number++;  
    Console.WriteLine(number);
}
```