Switch, Enumerations, Exception Handling, Recursion

Engineering 1D04,

Teaching Session 12

Switch Statement

Switch Statement

- Consider the if-else construct.
- A typical situation is to capture a keystroke and use it to determine some aspect of the application's behaviour.
- For example:
  - 'B' or 'b' means set bold font
  - 'C' or 'c' means copy selected text
  - 'I' or 'i' means set italics
  - 'M' or 'm' means move the selected text
  - 'Q' or 'q' means quit the application

how do we implement this?

Switch Statement Motivation

```
if (ch == 'B' || ch == 'b')
{
        //code to bold text
}
else if (ch == 'C' || ch == 'c')
{
        //code to copy text
}
else if (ch == 'I' || ch == 'i')
{
        //code to italicize text
}
else if (ch == 'M' || ch == 'm')
{
        //code to move text
}
else if (ch == 'Q' || ch == 'q')
{
        //code to quit
}
```

Switch Statement Motivation

```
if (ch == 'B' || ch == 'b')
{
        //code to bold text
}
else if (ch == 'C' || ch == 'c')
{
        //code to copy text
}
else if (ch == 'I' || ch == 'i')
{
        //code to italicize text
}
else if (ch == 'M' || ch == 'm')
{
        //code to move text
}
else if (ch == 'Q' || ch == 'q')
{
        //code to quit
}
```

And this can get cumbersome!

Also, what if you have many cases to consider - does the order of the else-if clauses matter?

The Switch Construct

```
switch (ch)
{
    case 'B':
    case 'b':
       //code to bold text
       break;
    case 'C':
    case 'c':
       //code to copy text
       break;
    case 'I':
    case 'i':
       //code to italicize text
       break;
    . . .
       break;
    case 'Q':
    case 'q':
       //code to quit
      break;
}
```

how do you think this works?

The Switch Construct

```
switch (ch)
{
    case 'B':
    case 'b':
       //code to bold text
       break;
    case 'C':
    case 'c':
       //code to copy text
       break;
    case 'I':
    case 'i':
       //code to italicize text
       break;
    . . .
       break;
    case 'Q':
    case 'q':
       //code to quit
       break;
}
```

variable that controls switch (variable must be “simple” - integer, character etc)

break - transfers control outside of the switch (works in loops also)

The Switch Construct

```
switch (ch)
{
    case 'B':
    case 'b':
        //code to bold text
        break;
    . . .
        break;
    case 'Q':
    case 'q':
        //code to quit
        break;
    default:
        //code to run if no case is entered
        break;
}
```

we can also use a default case to cope with the situation if no case has been specified with the current value of the switch variable

for example, what if ch = ‘s’?

The Switch Construct

- Surprisingly, as well as integer, characters and other sub-range variables, in C# strings can also be used as the switch variable.

The Switch Construct

- The *switch/case* construct has two main advantages over if-else-if:
  - It produces code that is much clearer to read and understand
  - In situations where we have many *cases*, the construct is implemented in a way that results in equal execution time for each *case*, independent of where it is in the construct
- *switch/case* cannot be used to replace general if-else constructs, just these simple "choice" constructs.

Enumerations

Enumerations

- In constructing an algorithm we may elect to work with a variable that takes on a few specific values.
- For example, say we have an algorithm that specifies different behaviour for different days of the week.
  - for mon, wed, and thu we have c = c + 20
  - for tue and fri we have c = c + 30
  - for sat and sun we have c = c + 15

how do we implement this?

Enumerations

- Without enumerations we use integers to refer to each day of the week - say
  - 0 ⇒ sun
  - 1 ⇒ mon
  - 2 ⇒ tue
  - 3 ⇒ wed
  - 4 ⇒ thu
  - 5 ⇒ fri
  - 6 ⇒ sat

so

```
int c, day;
. . .
if (day == 1 || day == 3 ||
    day == 4)
   c += 20;
else if (day == 2 || day == 5)
   c += 30;
else if (day == 0 || day == 6)
   c += 15;
```

Enumerations

- It would be a lot more straight forward if we could use variables of type DayOfWeek.
- This is what enumerations let us do.
- The keyword in C# is *enum*.
- We can declare an enum as follows

```
enum DayOfWeek {sun, mon, tue, wed, thu, fri, sat};
DayOfWeek day;
. . .
```

Enumerations

- So, we can use the enumeration as follows:

```
enum DayOfWeek {sun, mon, tue, wed, thu, fri, sat};
DayOfWeek day;
int c;
. . .
if (day == DayOfWeek.mon || day == DayOfWeek.wed ||
    day == DayOfWeek.thu)
   c += 20;
else if (day == DayOfWeek.tue || day == DayOfWeek.fri)
   c += 30;
else if (day == DayOfWeek.sun || day == DayOfWeek.sat)
   c += 15;
```

Enumerations

- The enum implementation in C# has some useful properties. We can access the name of the enum element as a string.
- Example:

```
enum DayOfWeek {sun, mon, tue, wed, thu, fri, sat};
DayOfWeek day;
int c;
. . .
MessageBox.Show(Convert.ToString(day));
```

Enumerations

- What does this do?

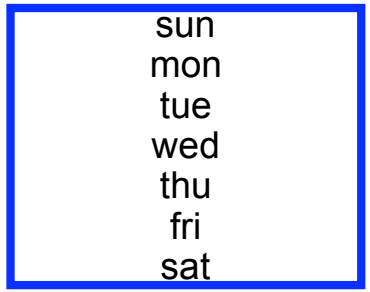
```
enum DayOfWeek {sun, mon, tue, wed, thu, fri, sat};
DayOfWeek d;
string s;

s = "";
for (d = DayOfWeek.sun; d <= DayOfWeek.sat; d++)
   s += d + "\n";
MessageBox.Show(s);
```

Enumerations

- What does this do?

shows a message box of the form:




```
enum DayOfWeek {sun, mon, tue, wed, thu, fri, sat};
DayOfWeek d;
string s;

s = "";
for (d = DayOfWeek.sun; d <= DayOfWeek.sat; d++)
   s += d + "\n";
MessageBox.Show(s);
```

starting value

ending value

increment value

this works because d can be converted to a string - so, as we saw early on, string concatenation does an automatic string conversion

Exception Handling

Exception Handling

- *Exception handling* is a crucial component of software design.
- An *exception* is a situation that arises in which something has “gone wrong”.
- Examples:
  - division by 0
  - array index out of bounds
  - opening a file that does not exist
  - converting a string to a number - and the string does not represent a number

Exception Handling

- If we do not “handle” these exceptions, they will cause run-time errors - commonly known as “crashes”.
- Handling the exception typically involves detecting that an exception has occurred, and then including code that warns the user about the problem but does not execute the code that would generate an error, or uses a default value or takes some other action to avoid an error.

Exception Handling

- Example:

```
if (a != 0) z = b / a;
else z = 0;
```

  - This avoids a crash caused by division by zero.

```
int a[] = new array[max];
if (j >= 0 && j < max)
   z = a[j];
else {
   z = a[0];
   MessageBox.Show("a out of bounds");
}
```

  - Avoids a crash caused by a array out of bounds.

Exception Handling

- In older languages we had to handle exceptions by testing for them and then having specific code written for each case.
- Modern languages have special constructs that help us handle exceptions.
- The construct in C# is

```
try
catch
finally
```

Exception Handling

- When an exception occurs, we say that an exception has been *thrown*.
- Exception handlers therefore *catch* exceptions that have been thrown.
- We can be specific about the exception that is caught, or we can catch general exceptions.

Why do you think it helps to be specific?

(We'll answer this later)

Exception Handling

- Example:

```
try
{
    z = b / a;
}
catch
{
    z = 0;
}
```

potential divide by zero

if there was a division by zero, an exception is thrown - and caught in here. This code then gets executed.

If there is no exception (any kind) then the code in the catch clause is not executed at all.

Exception Handling

- How is this different from the previous example?

```
try
{
   z = b / a;
}
catch (DivideByZeroException e)
{
   z = 0;
}
```

Exception Handling

- How is this different from the previous example?

```
try
{
   z = b / a;
}
catch (DivideByZeroException e)
{
   z = 0;
}
```

object of Exception class

This will catch only divide by zero exceptions.
Early one catches any exception.

Exception Handling

- But we can combine catch clauses:

```
try
{
   b = Convert.ToInt32(txtValue.Text);
   z = b / a;
}
catch (DivideByZeroException e)
{
   z = 0;
   MessageBox.Show("Divide by zero");
}
catch (FormatException e)
{
   MessageBox.Show("Text must be valid number");
}
```

Exception Handling

- But we can combine catch clauses:

```
try
{
   b = Convert.ToInt32(txtValue.Text);
   z = b / a;
}
catch (DivideByZeroException e)
{
   z = 0;
   MessageBox.Show("Divide by zero");
}
catch (FormatException e)
{
   MessageBox.Show("Text must be valid number");
}
```

This clause executed if divide by zero

This clause executed if format problem

Exception Handling

- We can use the Exception class objects:

```
try
{
   b = Convert.ToInt32(txtValue.Text);
   z = b / a;
}
catch (DivideByZeroException e)
{
   z = 0;
   MessageBox.Show(e.Message);
}
catch (FormatException e)
{
   MessageBox.Show(e.Message);
}
```

(can use any name here)

try it

Exception Handling

- The *finally* clause gets executed whatever happens.
- This is particularly useful for file I/O since we need to close files after opening and using them.

Exception Handling

- How does the try-catch-finally work?
  - statements in the *try* clause are executed
  - if no exceptions occur - execution continues with the code immediately following the final *catch* clause (could be a *finally* clause)
  - if an exception occurs - control is transferred to the applicable *catch* clause, and then to the code immediately following the final *catch* clause
  - if there is no *catch* clause there must be a *finally* clause

Exception Handling

- So, there are advantages to using specific exception classes
  - More specific feedback for the user
  - We can use built in methods/properties of the exception classes

Recursion

Recursion

- Another approach for iteration
- Most natural approach for some problems
- A recursive function (method) is one that calls itself
- N! = N*(N-1)*(N-2)*…*2*1
- e.g. 5!=5*4*3*2*1 = 5*4! = 5*4*3! = …
- N! =
  - 1 if N=0 or N=1
  - N*(N-1)! If N>1

Recursion Continued

- The idea is to reduce the instance of the problem to the same problem with "smaller" input
- A recursive function consists of two parts:
  - Base case – describes a simple case of the problem that can be solved in non-recursive form
  - Recursive part – the other cases of the problem can be reduced (by recursion) to problems that are closer to the base case

Factorial Function

```
int factorial(int n)
{
   if ((n==0)||(n==1))
   {
     return 1;
   }
   else
   {
     return n*factorial(n-1);
   }
}
```

Sum Example

```
//Return the sum of 1 to n
public int sum(int n)
{
   int result;
   if (n==1)
   {
     result = 1;
   }
   else
   {
     result = n + sum(n-1);
   }
   return result;
}
```

How would you write this summation using the "big sigma" notation?

Advice on Exam Preparation from your TAs Perspective

Questions?

- Data types?
- Expressions?
- Methods?
- Conditional statements?
- Iterative statements, recursion?
- Arrays?
- Object Oriented Programming?
- File I/O?
- Exception Handling?

Concluding Remarks

- Final exam will be multiple choice
  - **BRING YOUR OWN HB PENCILS**
- Good luck!
  - With your exams
  - With second term
  - With second year and beyond
- Have a great career!