

Classes

- we will continue to implement the `my_stream` class
- it will have public methods:
 - `getch`, `putch`, `expect`, `error` and `peek`
 - and an initialiser

slide 3
gaius

Version 1: Pseudo code for the `my_stream` class

- please note this is pseudo code - it will not compile
- used to convey ideas
 - it allows the design to iterate towards final code
 - taking small incremental steps

slide 4
gaius

Version 1: Pseudo code for the `my_stream` class

- ```
using System;

class my_stream
{
 string input;

 public char getch ()
 {
 char ch = input[0]; /* remember first element of the
 string */
 input = input[1:]; /* remove the first element of the
 string */
 return ch; /* return first element. */
 }
}
```

## Version 1: Pseudo code for the my\_stream class

- it assumes that the string `input` contains the complete input stream
- it allows the user of the class to examine each character
  - one at a time
  - `getch` will return the first character from the `input` and will then remove this character from `input`

## Version 1: Pseudo code for the my\_stream class

- ```
public char putch (char ch)
{
    input = ch + input;
    return ch;
}
```
- `putch` places `ch` onto the front of the `input` stream
- finally it returns the character, `ch`
- `putch` is designed to return the parameter, `ch`, to allow a more functional style of programming

Version 1: Pseudo code for the my_stream class

- this functional style of programming allows us to implement `peek` as follows:

```
public char peek ()
{
    return putch (getch ());
}
```

- `peek` non destructively returns a copy of the first character in the `input` stream
- the string `input` is unaltered by `peek`

Version 1: Pseudo code for the my_stream class

- ```
/*
 * my_stream - the initialiser method.
 */

public my_stream (string s)
{
 input = s;
}
```
- the initialiser which must be given a string
  - and is assigned to the private field `input` in `my_class`

## Version 1: Pseudo code for the my\_stream class

```
public void error (string s)
{
 Console.WriteLine ("error: {0} occurred before {1}", s, ...);
}
```

## Version 2: C# code for my\_stream

- placed into filename mystream.cs

```
using System;

class my_stream
{
 private string input;

 public char getch ()
 {
 char ch = input[0];

 input = input.Substring (1);
 return ch;
 }
}
```

## Version 2: C# code for my\_stream

```
public char putch (char ch)
{
 input = ch + input;
 return ch;
}
```

## Version 2: C# code for my\_stream

```
public void expect (char ch)
{
 if (ch == peep ()) {
 ch = getch ();
 return;
 }
 error ("expected {0} but seen {1} instead", ch, peep);
}
```

**Version 2: C# code for my\_stream**

```

public char peep ()
{
 return putch (getch ());
}

public void error (string s)
{
 Console.WriteLine ("error: {0} occurred before {1}", s);
}

```

**Version 2: C# code for my\_stream**

```

public void error (string format, char a, char b)
{
 Console.WriteLine (format, a, b);
}

/*
 * my_stream - the initialiser method.
 */

public my_stream (string s)
{
 input = s;
}
}

```

**Version 2: expr**

placed into filename expr.cs

```

using System;

class evaluate
{
 public int eval (string s)
 {
 my_stream t = new my_stream (s);

 return expression (t);
 }
}

```

**Version 2: expr**

```

/*
 * expression - return an integer by converting the in
 * The input stream is either:
 * a term or
 * a term-term or
 * a term+term
 */

```

**Version 2: expr**

```

int expression (my_stream s)
{
 int left = term (s);

 while ((s.peep () == '+')
 || s.peep () == '-')
 if (s.getch () == '+') {
 int right = term (s);
 left = left + right;
 }
 else {
 int right = term (s);
 left = left - right;
 }
 return left;
}

```

**Version 2: expr**

```

/*
 * term - return an integer by converting the input st
 * The input is either:
 * a factor or
 * a factor * factor or
 * a factor / factor
 */

```

**Version 2: expr**

```

int term (my_stream s)
{
 int left = factor (s);

 while (s.peep () == '*' ||
 s.peep () == '/')
 if (s.getch () == '*') {
 int right = factor (s);
 left = left * right;
 }
 else {
 int right = factor (s);
 left = left / right;
 }
 return left;
}

```

**Version 2: expr**

```

/*
 * factor - return an integer by converting the input
 * The input is either:
 * a number or
 * a (expression)
 */

```

**Version 2: expr**

```

int factor (my_stream s)
{
 if (s.peep () == '(') {
 s.expect ('(');
 int e = expression (s);
 s.expect (')');
 return e;
 }
 return number (s);
}

```

**Version 2: expr**

```

/*
 * number - return an integer by converting the digits
 * the input stream. The integer stops when
 * are no more digits in the stream.
 */

```

**Version 2: expr**

```

int number (my_stream s)
{
 int n = 0 ;

 while (Char.IsDigit (s.peep ()))
 n = n * 10 + (int)s.getch () - (int)'0';
 return n;
}

```

**Test code for expression and my\_stream**

- placed into file testexpr.cs

```

using System;

class testexpr
{
 public static void Main ()
 {
 evaluate e = new evaluate ();
 string q = "3*2-(3+4)";
 int i = e.eval (q);

 Console.WriteLine ("{0} = {1}", q, i);
 }
}

```

## Compiling the project

- ```
$ dmcs -out:calc.exe expr.cs mystream.cs testexpr.cs
```

- running the program

- ```
$ mono calc.exe
3*2-(3+4) = -1
```

## Conclusion

- careful design of classes mean that you can
  - reuse your code
  - isolate feature sets and therefore isolate bugs
  - minimise the interfaces between classes
    - called a loosely coupled system
- in practice it is good to keep as many fields within a class `private`
- we could reuse `my_stream` in future projects
- we could reuse `expr` in future projects