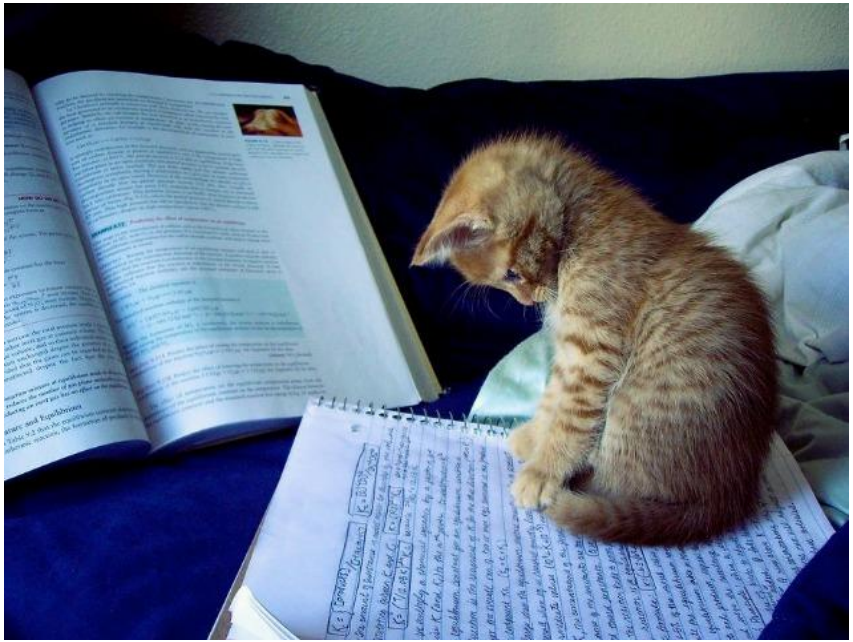
FINAL REVIEW

BME 121 2016

Jeff Luo



Variables and number types

- Variables
  Type varName;
- Assignment
  varName = value;
- Number types
  int and double
  int overflow (+/- 2M range limit)
- Arithmetic
  + - * / % ()
- Comparison
  == >= <= > < !=
- Special
  ++ --
- Compound Assignment
  += -= *= /= %=
- System.Math library
- System.Random class

Console, text types and string library

- Text data types
  string, char
- Combining strings
  +
- Placeholders and formatters
  {id,spaces:style}
  String.Format()
  Custom Formatters: https://msdn.microsoft.com/en-us/library/txafckwd(v=vs.110).aspx
- Escaped characters
  \n \r \t
- System.String library
- Console
  ReadLine, Write, WriteLine

Bool type and operations

- bool

  only stores true or false

- Logical operators:

  && || ! ^

- Converting values between types

type.Parse()

type.TryParse()

varName.ToString()

Decisions

```
if(  // test )
{

}
else if ( //test 2 )
{

}
else
{

}
```

```
test  ?  valueIfTrue : valueIfFalse ;
```

```
switch( //expression ) {
        case value1:

        break;
        case value2:

        break;
        default:

        break;
}
```

Loops

```
while( // test )
{

}

do
{


} while( // test )
```

```
for ( //start  ;  // end ;  // increment )
{

}

foreach ( type varName in collection )
{


}
```

Methods, pass by value and pass by reference

returnType MethodName(// parameters)
{

return *value*;
}

static methods:

must only call or use other static fields and methods, unless the method is used on some object

- Value types:
  int, double, string, char, bool
- Reference types:
  any class or interface
- Pass by value
  only value types
- Pass by reference
  any reference type
  value types if the parameter is declared as an **in**, **out**, or **ref** parameter

Recursion

- Recursion

- A method that calls a copy of itself to solve a sub-problem that is similar to the original problem.

Arrays and the Array class

- 1D

  ```
  type[] arrayVarName = new type[size];
  arrayVarName.Length
  ```

- 2D Rectangular

  ```
  type[,] arrayVarName = new type[rows, cols];
  arrayVarName.GetLength(0) // # of rows
  arrayVarName.GetLength(1) // # of cols
  ```

- 2D Jagged

  ```
  type[][] arrayVarName = new type[rows][];
  // in loop:
  arrayVarName[i] = new type[cols for row i];
  arrayVarName.Length  // # of rows
  arrayVarName[i].Length // # of cols for row i
  ```

Helper Methods and Procedural Design

- Break down the problem
  - Main Procedure → Main method
  - Sub Procedures → Helper methods

- Identify repeating code

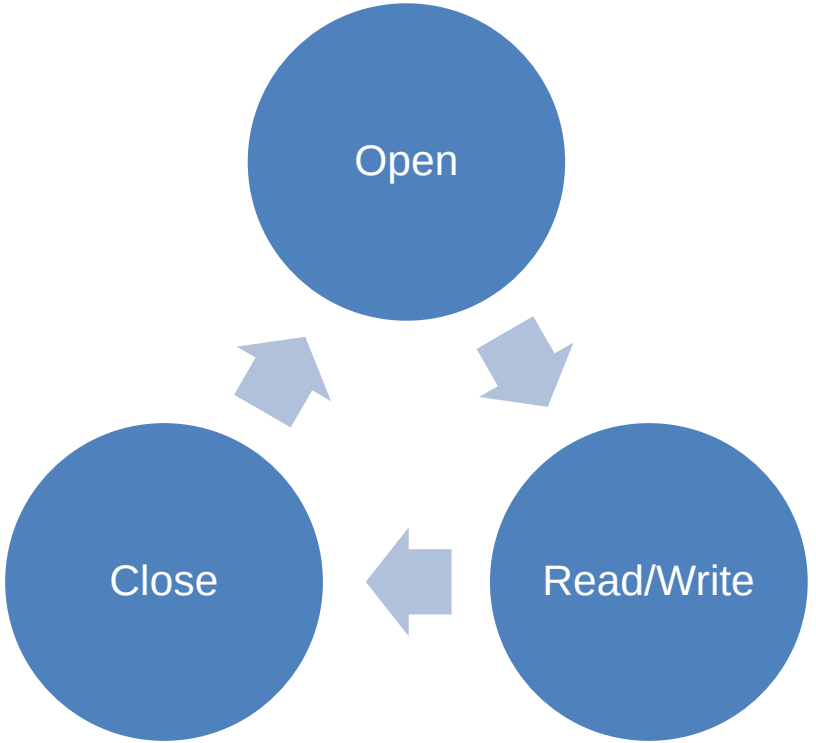
Classes and Objects

- Class has:
  - Constructor
  - Destructor (C++)
  - Fields
  - Properties
  - Methods
- An object is a copy of the class with values filled in for the fields

- All classes we define automatically inherit the System.Object class https://msdn.microsoft.com/en-us/library/system.object :
  - ToString()

File I/O

- FileStream
  - FileMode
  - FileAccess
  - Dispose()
- StreamReader
  - ReadLine()
  - EndOfStream
  - Dispose()
- StreamWriter
  - Write()
  - WriteLine()
  - Flush()
  - Dispose()



Exceptions, Try-Catch-Finally

- Exceptions
  - Crash the program immediately
  - Useful if code encounters unexpected input values (not from user)

        throw new Exception(“message”);

```
try {
      // code that might throw exceptions
}
catch (Exception e)
{
      // code that runs if an exception occurs
}
finally
{
      // code that will always run after the try block, even if exceptions are thrown
}
```

Testing

- Test Case
  - A pair of input and expected output for some method
- Test Set
  - A collection of test cases for a method, which together examines whether the method is correct (produces the expected output when given the specific inputs)

- A good test set has:
  - Standard cases → normal input values that are expected to be used with the method
  - Edge cases → unusual inputs, such as null, empty arrays, empty linked lists, …

Linked List

- A sequential data structure that holds a number of nodes

- LinkedList class:
  - Node head
  - Node tail
    - tail.Next always points to null
  - int count

- Node class:
  - *type* data
  - Node next

Linked List vs 1D Arrays

- Linked List
  - Flexibly grows and shrinks in size
  - Easy access to head and tail, expensive for all other nodes
  - Node reference costs memory, but provides the flexibility
  - Easy to add elements in the middle
  - Easy to merge or split linked lists (only few references to modify)
  - More efficient data structure for parallel programming
- Array
  - Must know the size before creation
  - Easy access to all elements via index
  - More memory efficient for the same # of data vs linked list

Binary Search

- Requires a sorted array (Not an efficient approach for linked list)
- Repeatedly compares a search value against the median, then determines based on that comparison whether to look into the left or right halves of the array

- When upper and lower indices crosses over, we conclude that the value doesn’t exist in the array

Insertion Sort

- Draws elements from the old data structure and inserts it into a position in the new data structure such that the new data structure maintains ascending (or descending) order
- Useful if you want to maintain a sorted data structure but the data comes in live

- Array implementation:
  - Using swaps and pushing data elements to the right
- Linked list implementation:
  - Using RemoveHead or RemoveTail in the old linked list and InsertInOrder in the new linked list

Selection Sort

- Repeatedly draws the smallest / largest element from the old data structure and appends / prepends it in the new data structure
- Useful if you have all of the data ahead of time

- Array implementation:
  - Using swaps
- Linked list implementation:
  - Using RemoveMin or RemoveMax in the old linked list and Append or Prepend in the new linked list

Inheritance

- Parent class defines fields and methods which are inherited by child class
- Child class can override definitions
  - Eg override ToString()
- Useful to categorize multiple related classes, and share common code
- In child classes:
  - this keyword → child class fields and methods
  - base keyword → parent class fields and methods

Interface

- A special kind of parent class where only abstract methods are defined (methods without any implementation). Interfaces do not define fields.
- Naming convention: IName

```
interface IName
{
        // methods
        type MethodOne();
        type MethodTwo();
}
```

Generics

- A generic class has type parameters:

YourClass< T >

- Types must be given when using the class

Polymorphism

- Technically 3 kinds
  - Method overloading – multiple versions of a method
  - Generic classes – classes that are specialized to specific types upon use
  - Inheritance – classes that specialize a parent class

- In industry, “Polymorphism” usually means inheritance and polymorphic method invocation

Polymorphic Method Invocation

- Parent class defines abstract methods that are concretely implemented in each child class

- Useful for defining shared methods but where the implementation must be customized by each child class
  - Eg: Shape.GetArea() → Circle.GetArea() and Square.GetArea()
  - Eg: Object.ToString() → YourClass.ToString()

Func and delegates

- Func is the full type of a method
- Func <T1, T2, …, Tn-1, Tn> is the type of a method with a method signature of:

  Tn MyMethod(T1 a, T2 b, …, Tn-1 z)

where Tn is the return type and T1 to Tn-1 are the types of the parameters.

- Useful for defining methods that allow for logical customization, which is supplied by a method that is passed to the Func parameter.

- Delegates
  - A way to define a method in-line as part of another method's call parameters (but the in-line method cannot be called anywhere else)


GOOD LUCK
ON YOUR EXAMS!
memegenerator.net