- Lecture 3

  - Agenda
    - Questions regarding Chapter 3?
    - Follow up on Pi exercise
    - Relevant/complicated subjects from Chapter 4
    - Exercise: 2D random fractal terrains using midpoint displacement

  - Chapter 4
    - Parameter modifiers on Page 122
      - Add to params to modify behavior
      - Pass by reference vs. pass by value
      - Is assigned value within method visible to outside world?

      - None (pass by value)
        - The default behavior
        - The most restrictive being the default prevents classes of bugs
        - What is copied depends on if argument is value or reference type
      - out (pass by reference)
        - The caller is required to fill in a value or compiler error
        - Methods can return more than one value without creating new type
        - TryParse example
          ```
          uint p;
          bool b = uint.TryParse("1000", out p);
          ```
      - ref (pass by reference)
        - Like out but must be initialized before passed into methods
        - Why not just always use ref then? Expresses different intent
        - If a reference type is passed by reference (with ref), the callee may change the values of the object's state data, as well as the object it's referencing
        - If a reference type is passed by value (the default), the callee may change the values of the object's state data, but NOT the object it's referencing
      - params
        - Console.WriteLine example from MSDN

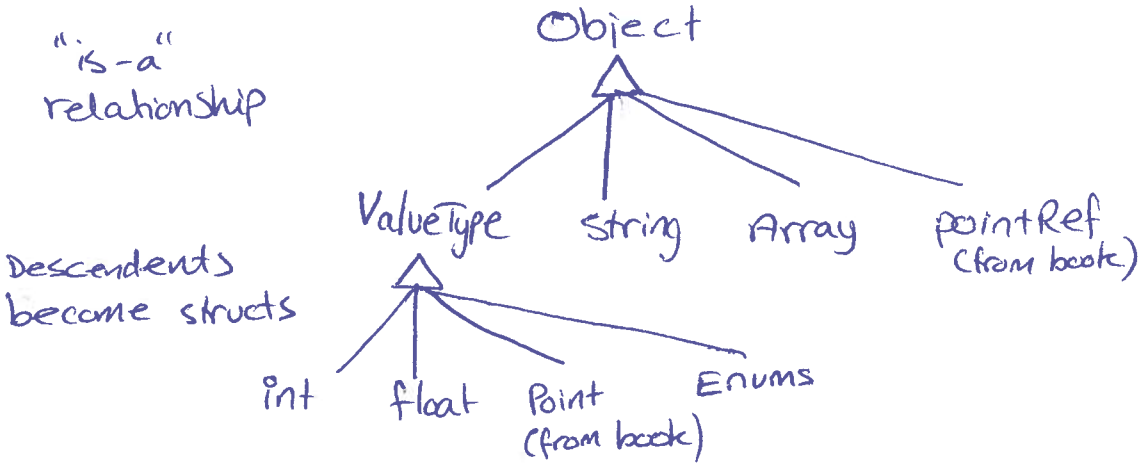
    - Array data structure
      - Container for data where you access elements by numeric index
      - Allocates a block of memory: size of data type * allocated size
      - Access elements by offset into block: index * size of data type
      - Once you allocate array, its size cannot change (know size in advance)

      - Array initialization syntax
        - ```
          // Indexing starts at zero, not one, and initialized to default value
          // 0 for values, null for references, false for bool
          int[] a = new int[3]; a[0] = 100;
          // short-hand population
          int[] a = new int[3] { 1, 2, 3 };
          // size is computed by compiler
          int[] a = new int[] { 1, 2, 3 };
          // size computed and short-hand population
          int[] a = { 1, 2, 3 };
          ```
        - a is really of type System.Array and that's where methods reside

    - Enum type
      - Without enum you'd create prefixed constants
      - By default enum "inherit" from Int32

    - Value types and reference types
      - Role of System.ValueType is to ensure derived types are stack allocated
      - On allocation, the runtime checks if type descends from ValueType
      - Stack allocated data is small, short-lived, needs fast alloc/dealloc
      - Ties into what happens with arguments on method calls
          ```

Inheritance tree for value and reference types



- Value type descendents are stored on stack
- All other objects stored on heap
- Value types typically small in size (bytes) and usually more often used than classes
- A stack is vally associated with every method call, whereas there's only one heap in the program.
- Heap is garbage collected

Simple method call with args passed by value

```
...
int a = 2;
int b = 5;
int c = Add(a, b);

int Add(int op1, int op2)
{
  // compiler setup code
  int op1 = a;
  int op2 = b;
  // end of setup code
  int sum = op1 + op2;
  return sum;
}
```

args passed by value, bit by bit

allocate on stack and setup local variables

deallocate on stack when method exits. It's safe to delete values stored on stack as methods can only be entered through new call, which creates a new stack entry

Stack allocation with nested scopes

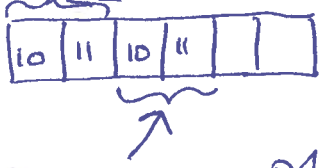
```
int SomeMethod(...)
{                    stack 1 alloc
  if(...)
  {                  stack 2 alloc
    if(...)
    {                stack 3 alloc
      for(...)
      {              stack 4 alloc
        ...
      } stack 4 dealloc
    } stack 3 dealloc
  } stack 2 dealloc
} stack 1 dealloc
```

Assuming that all local variables within each scope is a value type (a struct) its lifetime is very predictable. When a struct falls out of defining scope, it can be immidiately removed from memory

Point Value type passed as argument

Page 150 in book

```
Point p1 = new Point(10, 11);
```



stack

```
Point p2 = p1;
```

assignment causes shallow copy (bit by bit copying) and since "value" is steres on stack the "value" is copied.

p2 can be modified without affecting p1.

```
int SomeMethod(Point p)
{
    p = ...
    // real method code
}
```

stack alloc

stack dealloc

Point reference type passed as argument

~~Point p1 = n~~

```
PointRef p1 = new PointRef(10, 11);
```

stack

Heap

```
PointRef p2 = p1;
```

Shallow copying a reference copies it on the stack bit by bit, but since value isn't stored on stack, only a copy of the pointer is made.

```
// some program code as for
// SomeMethod as above, but with
// indirection to heap.
```

```
p2.x = 12;
```

would change the value of x in both p1 ~~and~~ and p2.

Solution: Implement deep copy upon assignment by implementing the ICloneable interface for PointRef.