



# COMPSCI 280 S1 2011 Enterprise Software Development

Programming Fundamentals  
Operator Overloading & Indexers



## Agenda & Reading

- ▶ **Agenda:**
  - ▶ Operator Overloading
    - ▶ Arithmetic operators (binary)
    - ▶ unary -
    - ▶ Increment/decrement operators
    - ▶ Relational Operators
    - ▶ Overloading true and false
    - ▶ == & != operators
  - ▶ Indexers
    - ▶ Indexers on Arrays
    - ▶ Indexers on Strings
    - ▶ Indexers on other data type
- ▶ **Recommended Reading:**
  - ▶ Microsoft Visual C# 2008 Step by Step
    - ▶ Chapter 16, 21
- ▶ **Hands-On Lab:**
  - ▶ Lecture 10 Lab



## Operator Overloading

- ▶ **C# allows you to overload operators for use on your own classes.**
  - ▶ You use method-like syntax with a return type and parameters,
  - ▶ But the name of the method is the keyword operator together with the operator symbol
  - ▶ Rules
    - ▶ All operators must be public
    - ▶ All operators must be static
    - ▶ A binary operator has two explicit arguments and a unary operator has one explicit argument
      - For binary operators, at least one of the operands must be of the **same type** as its class
      - For unary operators, the operand must be of the same type as the class
    - ▶ Some operators naturally come in **pairs**. You must define them both. Example: ==, !=
    - ▶ Operator parameters must **NOT** use the ref or out modifier



## Arithmetic Operators

- ▶ **Overloading Arithmetic Operators:**

- ▶ +
- ▶ -

(11, 12, 13)

```
ThreeD a = new ThreeD(1, 2, 3);
ThreeD b = new ThreeD(10, 10, 10);
ThreeD c = new ThreeD();
c = a + b; // add a and b together
Console.WriteLine(c);
```

```
public class ThreeD {
    int x, y, z; // 3-D coordinates
    // constructors ...

    public static ThreeD operator +(ThreeD op1, ThreeD op2) {
        ThreeD result = new ThreeD();
        result.x = op1.x + op2.x; // These are integer additions
        result.y = op1.y + op2.y; // and the + retains its original
        result.z = op1.z + op2.z; // meaning relative to them.
        return result;
    }
    public static ThreeD operator -(ThreeD op1, ThreeD op2) {
        ThreeD result = new ThreeD();
        /* order: op1 is the left operand and op2 is the right. */
        result.x = op1.x - op2.x; // these are integer subtractions
        result.y = op1.y - op2.y;
        result.z = op1.z - op2.z;
        return result;
    }
}
```



## Unary Operator:

- ▶ unary – (negated fields of the operand)

```
public static ThreeD operator -(ThreeD op) {
    ThreeD result = new ThreeD();
    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;
    return result;
}
```

```
c = -a; // assign -a to c
Console.Write("Result of -a: ");
Console.WriteLine(c);
```

(-1, -2, -3)



## Increment & decrement

- ▶ You can declare your own version of the increment (++) and decrement (--) operators.
  - ▶ They must be public, static and unary.
  - ▶ They can be used in prefix and postfix forms

```
public static ThreeD operator ++(ThreeD op) {
    ThreeD result = new ThreeD();
    result.x = op.x + 1;
    result.y = op.y + 1;
    result.z = op.z + 1;
    return result;
}
```

```
Console.WriteLine(b++);
Console.WriteLine(b);
```

(10, 10, 10)  
(11, 11, 11)

```
public static ThreeD operator --(ThreeD op) {
    ThreeD result = new ThreeD();
    result.x = op.x - 1;
    result.y = op.y - 1;
    result.z = op.z - 1;
    return result;
}
```

```
Console.WriteLine(++a);
Console.WriteLine(a);
```

(2, 3, 4)  
(2, 3, 4)



## Binary Operators revisited

- ▶ For binary operators, at least one of the operands must be of the same type as its class

- ▶ operator + (ThreeD op1, ThreeD op2)
- ▶ operator + (ThreeD op1, int op2)
- ▶ operator + (int op1, ThreeD op2)

```
c = a + b; //
c = a + 10; //
c = 10 + b; //
```

```
public static ThreeD operator +(ThreeD op1, int op2){ // ThreeD + int
    ThreeD result = new ThreeD();
    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;
    return result;
}
public static ThreeD operator +(int op1, ThreeD op2){ // int + ThreeD
    ThreeD result = new ThreeD();
    result.x = op2.x + op1;
    result.y = op2.y + op1;
    result.z = op2.z + op1;
    return result;
}
```



## Relational Operators

- ▶ < (less than)
- ▶ > (greater than)
- ▶ Returns a **true** or **false** value
- ▶ Must overload them in **PAIRS**
- ▶ Example: compare ThreeD objects based on their distance from the origin.
  - ▶ op1 > op2 when the op1's distance from the origin is greater than the other

```
if (a < b)
    Console.WriteLine("a < b is true");
```

```
public static bool operator <(ThreeD op1, ThreeD op2) {
    if (Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) <
        Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
        return true;
    else
        return false;
}
public static bool operator >(ThreeD op1, ThreeD op2) {
    ...
}
```



## Overloading true and false

- ▶ The keyword true and false can be used as unary operators
- ▶ Once overloaded, you can use objects of that class to control the if, while, for and do-while statements, or in a ? expression.
- ▶ Must overload them in **PAIRS**
- ▶ Example:
  - ▶ A ThreeD object is true if at least one coordinate is non-zero;
  - ▶ If all coordinates are zero, then the object is FALSE.

```
public static bool operator true(ThreeD op) {
    if ((op.x != 0) || (op.y != 0) || (op.z != 0))
        return true; // at least one coordinate is non-zero
    else
        return false;
}

public static bool operator false(ThreeD op) {
    if ((op.x == 0) && (op.y == 0) && (op.z == 0))
        return true; // all coordinates are zero
    else
        return false;
}
```

```
if (c)
    Console.WriteLine("c is true.");
else
    Console.WriteLine("c is false.");
```

9

COMPSCI280

Handout10



## Determining Equality

- ▶ Two kinds of comparison for objects:
  - ▶ Identity and equality
- ▶ Object.ReferenceEquals
  - ▶ Used to compare identity. It compares the addresses of the objects in the memory to determine if they are the same object.
- ▶ Objects can also be compared for equality:
  - ▶ Use Object.Equals, equality operator (==) or inequality operator (!=)
  - ▶ The default Object.Equals calls object.ReferenceEquals
  - ▶ Guidelines:
    - ▶ You should override the equals method in your own class for value comparison
    - ▶ Do not throw an exception in the implementation of an Equals method.

```
MyPoint p1 = new MyPoint(1, 2);
MyPoint p2 = new MyPoint(1, 2);
if (Object.ReferenceEquals(p1, p2))
    Console.WriteLine("Identical");
else Console.WriteLine("Not Identical");
```

Not Identical

10

COMPSCI280

Handout10



## Value Types

- ▶ Implementing the Equality Operator (==) on Value Types
  - ▶ No default implementation of the equality operator (==) for value types
  - ▶ You should overload the **equality** operator (==) any time equality is meaningful
  - ▶ You should consider overriding the **Equals** method to gain increased performance

```
public override int GetHashCode(){
    return _h.GetHashCode();
}

public override bool Equals(object o){
    if (!(o is Hour)) {
        return false;
    }
    return this == (Hour)o;
}
```

Hour – structure type

no classes will be derived from Hour.  
Don't need to use GetType.  
Use the is operator

```
public static bool operator ==(Hour a, Hour b) {
    return a.h == b.h;
}

public static bool operator !=(Hour a, Hour b){
    return a.h != b.h;
}
```

11

COMPSCI280

Handout10



## Reference Types

- ▶ Implementing the Equality Operator (==) on Reference Types
  - ▶ default implementation of the equality operator (==) for reference type => check identity
  - ▶ You should consider overriding the **Equals** method on a reference type so that it compares the contents of two objects for equality

```
public override bool Equals(Object obj){
    if (obj == null || GetType() != obj.GetType())
        return false;
    MyPoint p = (MyPoint)obj;
    return (x == p.x) && (y == p.y);
}

public override int GetHashCode(){
    return x ^ y;
}
```

Check the object obj references an instance of the same type as this object

uses the GetType Method to determine whether the run-time types of the two objects are identical

MyPoint – class type

12

COMPSCI280

Handout10



## Indexers

- ▶ Indexers permit instances of a class or struct to be indexed in the same way as arrays.
  - ▶ Indexers are similar to properties except that their accessors take parameters.
    - ▶ A get accessor returns a value. A set accessor assigns a value.
  - ▶ The **this** keyword is used to define the indexers.
  - ▶ The value keyword is used to define the value being assigned by the set indexer.

```
class MyPointArray {
    private MyPoint[] ptArray = new MyPoint[10];
    public MyPoint this[int i] {
        get {
            return ptArray[i];
        }
        set {
            ptArray[i] = value;
        }
    }
}
```

```
MyPointArray ptArr = new MyPointArray();
ptArr[0] = new MyPoint();
ptArr[1] = new MyPoint(10, 20);
Console.WriteLine(ptArr[1].x);
```



## Overloading

- ▶ Indexers can be overloaded (array can't)
  - ▶ The signature of an indexer consists of the number and types of its formal parameters.
  - ▶ It does not include the indexer type or the names of the formal parameters.

```
public MyPoint this[double d] {
    get {
        int i = (int)d;
        if (i < 0 || i >= 10)
            return null;
        else
            return ptArray[i];
    }
    set {
        int i = (int)d;
        if (!(i < 0 || i >= 10))
            ptArray[i] = value;
    }
}
```

```
ptArr[1.4] = new MyPoint(4, 5);
Console.WriteLine(ptArr[1].x);
```



## Using non-numeric subscripts

- ▶ Indexers can use non-numeric subscripts (array can use only integer subscripts)

```
public MyPoint this[string s]
{
    get {
        int i = int.Parse(s);
        if (i < 0 || i >= 10) //check limit
            return null;
        else
            return ptArray[i];
    }
    set {
        int i = int.Parse(s);
        if (!(i < 0 || i >= 10))
            ptArray[i] = value;
    }
}
```

```
ptArr["2"] = new MyPoint(5, 6);
Console.WriteLine(ptArr[2].x);
```



## Indexers (con't)

- ▶ Indexers can't be used as ref or out parameters (but array can)

```
Method(ref ptArr[0]);
```

ERROR



## Indexers - ints

- ▶ Indexers don't have to operate on actual arrays

```
class PwrOfTwo {
    public int this[int index] {
        get {
            if ((index >= 0) && (index < 4)) return pwr(index);
            else return -1;
        }
    }

    int pwr(int p) {
        int result = 1;
        for (int i = 0; i < p; i++)
            result *= 2;
        return result;
    }
}
```

```
PwrOfTwo pwr = new PwrOfTwo();
Console.WriteLine("First 4 powers of 2: ");
for (int i = 0; i < 4; i++)
    Console.WriteLine(pwr[i] + " ");
```



## Indexers - strings

- ▶ Indexers can be used in strings

```
class DayCollection {
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    public int this[string day] {
        get {
            return (GetDay(day));
        }
    }

    private int GetDay(string testDay) {
        int i = 0;
        foreach (string day in days) {
            if (day == testDay) {
                return i;
            }
            i++;
        }
        return -1;
    }
}
```

```
DayCollection d = new DayCollection();
Console.WriteLine(d["Fri"]);
```



## Indexers - 2D array

- ▶ A 26 by 10 grid class that has an indexer with two parameters.

```
class Grid {
    const int NumRows = 26;
    const int NumCols = 10;
    int[,] cells = new int[NumRows, NumCols];

    public int this[char c, int col]{
        get{
            c = Char.ToUpper(c);
            ...
            return cells[c - 'A', col];
        }
        set{
            c = Char.ToUpper(c);
            ...
            cells[c - 'A', col] = value;
        }
    }
    ...
}
```

```
Grid g = new Grid();
g['b', 1] = 27;
```